Jihočeská univerzita v Českých Budějovicích

Přírodovědecká fakulta

Návrh a optimalizace procesu automatizovaného integračního testování

Bakalářská práce

Adrian Czarnomski

Vedoucí práce: Mgr. Vít Barabáš

Garantka práce: Ing. Marta Vohnoutová

České Budějovice 2020

Jihočeská univerzita v Českých Budějovicích
Přírodovědecká fakulta

ZADÁVACÍ PROTOKOL BAKALÁŘSKÉ PRÁCE

Student: Adrian Czarnomski
(jméno, příjmení, tituly)

Obor – zaměření studia: Aplikovaná informatika

Katedra: Ústav aplikované informatiky

Školitel: Mgr. Vít Barabáš, ENGEL strojírenská spol. s.r.o. Českobudějovická 314, 382 41 Kaplice, +420 730 529 478, vit.barabas@engel.at
(jméno, příjmení, tituly, u externího š. název a adresa pracoviště, telefon, fax, e-mail)

Garant z PřF: Ing. Marta Vohnoutová
(jméno, příjmení, tituly, katedra – jen v případě externího školitele)

Školitel – specialista, konzultant:
(jméno, příjmení, tituly, u externího š. název a adresa pracoviště, telefon, fax, e-mail)

Téma bakalářské práce:
Návrh a optimalizace procesu automatizovaného integračního testování

Cíle práce:

Cílem práce je vývoj nástrojů pro zavedení automatického integračního testování ve společnosti Engel strojírenská spol. s.r.o.
Práce bude sestávat z návrhu a implementace toolsetu realizující integrační testování pro firemní účely a vytvoření několika vzorových použití.
Zároveň řešitel provede analýzu současného procesu testování a nasazování software, na jejichž základě stanoví vhodnou metodiku pro implementaci automatizovaných integračních testů.

Základní doporučená literatura:

Financování práce: .....

Vedoucí práce: Vít Barabáš ..... podpis: .....

U externích vedoucích fakultní garant práce: Marta Vohnoutová podpis: .....

Garant oboru bak. studia (nepožaduje se u zaměření, příprava na mag. studium biologie)

..... podpis: .....

Vedoucí katedry ..... podpis .....

Případný souhlas vedoucího ústavu AV ..... podpis: .....

V Českých Budějovicích dne .....

Převzal/a dne 28.2.2019 ..... podpis: Czarnomski

Bibliografické údaje

Czarnomski, A., 2020: Návrh a optimalizace procesu automatizovaného integračního testování. [Design and optimization of automated integration testing process. Bc. Thesis, in Czech] – 48 p., Faculty of Science, University of South Bohemia, České Budějovice, Czech Republic.

Anotace

Bakalářská práce se zabývá analýzou stávajícího procesu manuálního integračního testování, používaném v denním provozu ve firmě Engel strojírenská spol. s.r.o., a vytvořením vlastního řešení v podobě nástroje pro automatizaci tohoto procesu.

Klíčová slova

Automatické testy, integrační testy, funkční testy, tester, automatizace testování, testovací scénáře, Selenium WebDriver, JUnit framework, Java

Annotation

The bachelor thesis deals with the analysis of the current process of manual integration testing used in daily operation in the company Engel strojírenská spol. s.r.o., and creating own solution in the form of a tool to automate this process.

Key words

Automatic tests, integration tests, functional tests, tester, testing automatization, test scenarios, Selenium WebDriver, JUnit framework, Java

Prohlášení

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

V Českých Budějovicích, dne 22. května 2020 Adrian Czarnomski

Poděkování

Tímto bych rád poděkoval Mgr. Vítu Barabášovi za odborné vedení praktické části práce, věcné připomínky a užitečné rady. Zároveň bych chtěl poděkovat Ing. Martě Vohnoutové za cenná doporučení a vstřícný přístup.

Obsah

1. Úvod

Trendem současného vývoje softwaru je kladení důrazu firem na odladění aplikací, aby byl počet chyb co nejvíce zredukován již v průběhu vývoje. Programy jsou objemnější a komplexnější, a proto je obtížné takový software udržet validní, obzvláště pokud na něm pracuje více vývojářů najednou a veškerá sebemenší změna kódu může napáchat mnoho nežádoucích a neočekávaných chyb. Z tohoto důvodu je potřeba udržovat software funkční a mít nad chybami do značné míry přehled. Jediným způsobem, jak si být jisti, že tohoto stavu docílíme, je testování.

Integrační testy patří mezi celou řadu dalších typů testů, přičemž se zaměřují konkrétně na integritu mezi jednotlivými částmi softwaru, operačním systémem či mezi aplikací a hardwarem daného zařízení.

Bakalářská práce vznikla souběžně v kooperaci se studijním kolegou Bc. Michalem Kuchtou, jenž v rámci své diplomové práce vytvořil nástroj pro vytváření testovacích scénářů a software pro spouštění virtualizovaných běhových prostředí pro mnou navržený testovací nástroj.

1.1 Cíle práce

Cílem práce je provést analýzu stávajícího procesu testování softwaru prováděném na softwarovém oddělení firmy Engel strojírenská spol. s.r.o. Výsledkem analýzy je pak návrh vlastního řešení v podobě nástroje pro zavedení automatických integračních testů. Účelem nástroje je zvýšit efektivitu testování softwaru na koncových zařízeních společnosti.

2. Testování softwaru

Testování softwaru je soubor mnoha činností, jež mají zkontrolovat, zda veškerá funkcionalita softwaru odpovídá očekávanému stavu dle požadavků koncového uživatele. Testování má odhalit a identifikovat případné chyby v aplikaci, nebo nesoulad mezi očekávanými a skutečnými výsledky testovaného subjektu. [2] Vývoj softwaru se bez pravidelného testování neobejde, neboť odhalování chyb a jejich oprava v průběhu vývoje šetří časové, finanční i lidské zdroje, které by v případě netestovaného softwaru způsobily s odstupem času mnoho problémů a ztrátou kontroly nad danou situací.

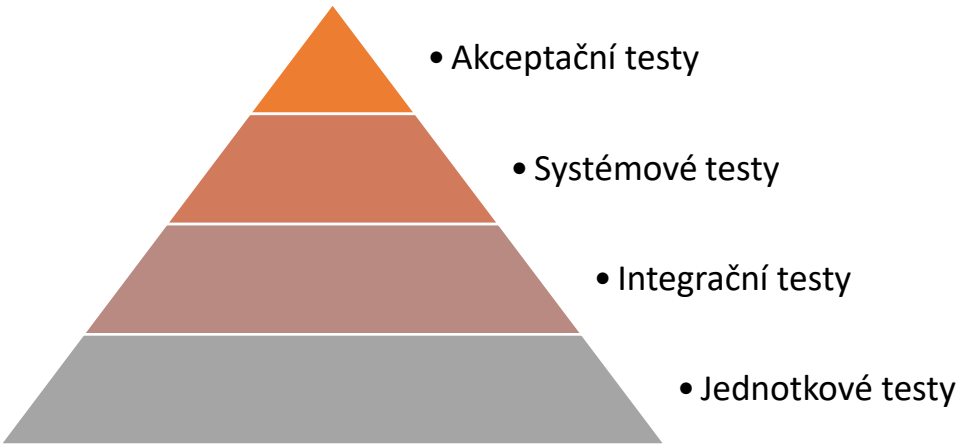
Ať už za jakékoliv situace dojde k nalezení závady během testování, je potřeba ji patřičně oznámit vývojářskému týmu, za jaké situace k chybě došlo a které kroky jí v rámci testování předcházely. Tímto principem dokážeme vývojáři či celému vývojovému týmu předat cenné informace, které následně napomohou k jednodušší identifikaci a odstranění nežádoucí vady v kódu.

Vykonávání testů má za úkol v případě manuálního testování tester, v případě automatického testování testovací softwarový nástroj. V obou případech je zapotřebí oznámit výsledek testu vývojářům formou reportu. Tester tak může učinit za pomocí test management tool[1], automatický testovací nástroj pak používá reporty prezentovány různými způsoby (záleží na použitém nástroji), nejčastěji však v samotném prostředí nástroje, pomocí emailu s výsledky v PDF[2] či jiném formátu, ve webové stránce nebo v prostém textu. [5]

[1] Nástroj pro agilní správu testů a řízení incidentů
[2] Portable Document Format – souborový formát vytvořený firmou Adobe

2.1 Úrovně testů

Úrovně testů jsou definicí různých typů testování v životním cyklu vývoje softwaru. Jinými slovy, během vývoje je potřeba v daný moment testovat určitou hloubku aplikace. Čím více aplikace „roste“, tím více je potřeba nad určitou úrovní složitosti vykonat různý typ testů. Je všeobecně určeno, že nejnižší úroveň, jenž představují jednotkové testy, obsahuje nejvíce testů, kdežto každá vyšší úroveň jich obsahuje méně. Pro snazší představu byl použit pyramidový graf, doplněný o jednotlivé úrovně testování.



Obrázek 2.1 - Pyramidové znázornění úrovní testů

Při testování je nutné postupovat od nejnižší vrstvy po tu nejvyšší. Pokud by totiž došlo v nižší úrovni testování k chybě, dojde k ní s největší pravděpodobností i ve vyšších vrstvách. Proto je při vývoji komplexnějšího softwaru nutné začít s testováním nižších úrovní testů již v prvopočátcích a postupně s rostoucí logikou testovat úrovně vyšší. [2]

2.1.1 Unit testování

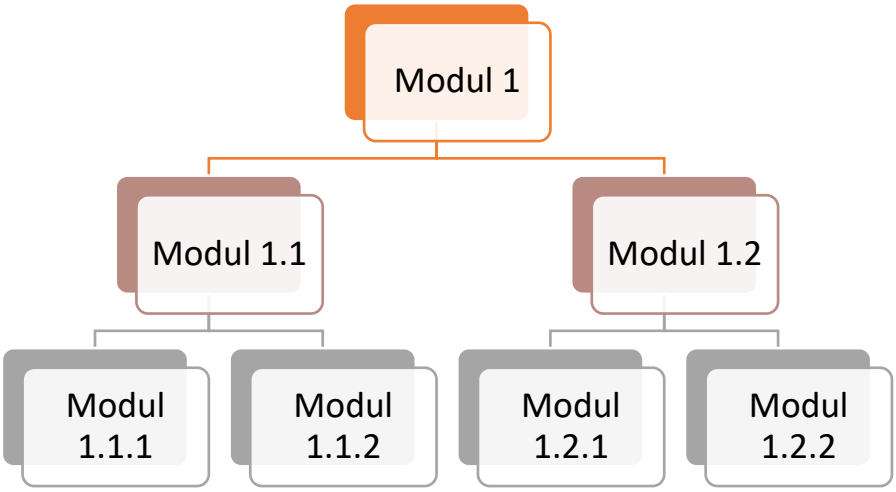
Unit testy nebo také jednotkové testy či testování komponent, jsou první etapou v testovacím procesu. Jednotkou se rozumí nejzákladnější testovatelná část softwaru představující v rámci OOP[3] třídu nebo metodu. Principem této úrovně je provést porovnání předpokládaného výstupu dané jednotky s jejím skutečným výstupem, čímž je docíleno odhalení chyb na nejzákladnější úrovni aplikace, které by bylo v dalších etapách testování obtížné najít.

Unit testy jsou vytvářeny většinou samotnými vývojáři, kteří danou komponentu implementovali, a znají tak její očekávanou funkcionalitu. Pro realizaci těchto testů jsou používány frameworky[4], které proces testování jednotek automatizují a vytváří finální výsledky těchto provedených testů. [2]

2.1.2 Integrační testování

Integrační testy jsou další úrovní v testovacím procesu. Každý větší softwarový projekt se typicky skládá z různých logických částí/jednotek (třídy, moduly), které jsou kódovány různými programátory. Pokud jedna z těchto částí na svém vstupu využívá výstup jiné části, nazýváme tento vztah jako interakci. Cílem integračních testů je testování skupin představující jednotlivé jednotky seskupené skrze vzájemné interakce. Tester má v této epoše testování na výběr z mnoha metodik, přičemž nejznámějšími a nejpoužívanějšími jsou testování integrace zdola-nahoru a shora-dolů. [2]

[3] Objektově orientované programování
[4] Sada tematicky zaměřených knihoven pro usnadnění konkrétního procesu



Obrázek 2.2 - Hierarchie modulů v integraci

Testování integrace zdola-nahoru je pro začátek doporučovanou technikou, neboť se nejprve testují mezi sebou ty nejzákladnější skupiny jednotek a pokračuje se čím dál většími shluky. Výhodou této metodiky je možnost paralelizovat[5] testování nižších skupin jednotek a časově zefektivnit proces testování.

Metoda shora-dolů je používána především v případě, kdy nejsou veškeré skupiny nižší úrovně stále integrovány. V takovém případě je nutné ještě neexistující moduly nahradit zástupnými, které nazýváme stub. Stub jsou jinými slovy dvojníci budoucích, ještě neimplementovaných skupin či jednotek, jejichž funkcionalitu dokážou napodobit tak, že na volání reagují předpřipravenými výsledky nebo logikou. Díky tomu je možné velice rychle vytvořit první prototyp aplikace simulující konečnou funkcionalitu, neboť vytvořit stub je časově méně náročnější, než kdybychom vytvářeli kompletně funkční modul. Samotné testování tak probíhá z vyšších logických celků postupně k těm nižším. [6]

[5] Rozdělit a souběžně provádět vícero činností stejného typu

2.1.3 Systémové testy

Po ověření integrace přichází na řadu systémové testy, které jsou posledním stádiem testování během vývoje softwaru. Produkt se testuje jako funkční celek z pohledu zákazníka, a proto lze systémové testy zařadit mezi black box testování, popsané v oddílu 2.2.9 Testování black/white/gray boxu. Testovací tým provádí mechanismy, jež by odhalily nesplnění technických, obchodních, funkčních a nefunkčních požadavků na software. Z pohledu testera se jedná o nejvíce prováděné testy, protože je provádí opakovaně. Pokud je totiž nalezena sebemenší chyba či nesrovnalost v produktu, je testerem vše důkladně zdokumentováno a je poslán požadavek zpět na autora funkcionality, aby danou chybu opravil. Pokud se tak stane, je opět opakován testovací průběh, jež vedl k téže chybě. Přetestování se však neděje pouze při nalezení chyb, ale i po úpravách funkcionalit. Cílem je, aby software i po úpravě reagoval na vstupní hodnoty tak, jak je po něm vyžadováno, stejně tak by mělo dojít i k selhání, pokud programu předložíme nesprávná vstupní data. Tento princip se nazývá testy splněním a selháním, o kterých se dále píše v kategoriích testů.

Často je tato úroveň testů spjata s integračním testováním, celý proces pak probíhá najednou. V takovém případě se tato vrstva nazývá systémové integrační testování. [1]

Tester v rámci systémového testování používá testovací scénáře obsahující jednotlivé testovací kroky. Scénáře se pak mohou seskupovat do testovacích sad, které určují, jakým způsobem a do jaké hloubky se bude software testovat. Testovacím scénářům se v této práci věnuje podkapitola 3.3 Test management tool.

2.1.4 Akceptační testování

Akceptační testování, někdy označované také jako uživatelské akceptační testování, je poslední úrovní v testovacím procesu, než bude finální produkt uveden zákazníkem v praxi. Za předpokladu, že veškeré předešlé úrovně

testování proběhly validně, je předán software zákazníkovi, který s pomocí vlastního testovacího týmu provádí své testování nad produktem, dle vlastních testovacích procedur. Pokud je během procesu nalezena chyba či odhalena nežádoucí funkcionalita, je veškerý průběh, který k tomuto stavu vedl, patřičně zdokumentován a zaslán zpět dodavateli produktu. Ten se pak následně snaží chybu opravit v co možná nejkratším čase. Následně aplikace prochází opět všemi úrovněmi testování, dokud nebude výsledek pro zákazníka uspokojivý. [1]

2.2 Kategorie a typy testů

Předchozí kapitola se věnovala úrovním testování, nikoliv však samotným typům a kategoriím testů. Jednotlivé druhy testů rozdělujeme do kategorií na základě jejich zaměření, složitosti, či kým a jak jsou vykonány. [1]

2.2.1 Assembly testování

Assembly testy jsou nejzákladnějším typem testů. Jsou prováděny samotnými vývojáři po implementování změn v kódu. Je běžnou praxí, když si programátor pozve na testování jiného programátora a tyto testy pak provádějí společně. Software je ověřován na úrovni zdrojového kódu – vývojáři prochází kód řádek po řádku a snaží se odhalit chyby v nově implementované části. Díky tomu lze objevit chyby, které programátor prvotně neodhalil, například z nepozornosti. [1]

2.2.2 Exploratory testy

Exploratory testy jsou podskupinou manuálních testů. Jedná se o testy prováděné nikoliv podle předpřipraveného testovacího scénáře, avšak zcela volným průzkumem testovaného softwaru. V tomto případě neexistují předepsané metody, jak takové testy provádět, neboť sám tester musí zvolit takový způsob, který odpovídá jeho získaným zkušenostem a znalostem testovaného subjektu. Často se tak řídí především svojí intuicí. [8]

2.2.3 Manuální a automatizované testy

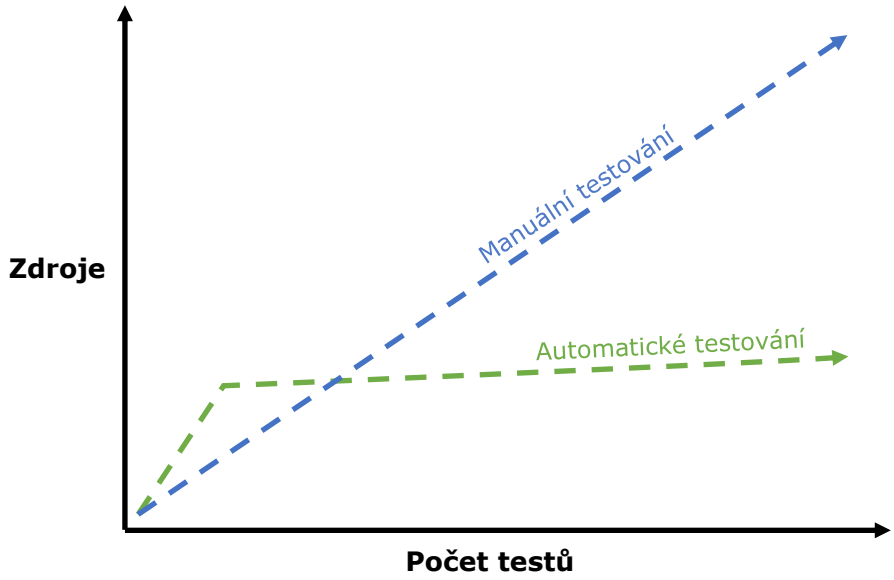
Jedná se o rozdělení testů na základě realizace – jakým způsobem jsou vykonávány. Manuální testování je prováděno kvalifikovaným testovacím týmem, kdežto automatizované testování je prováděno testovacími nástroji.

Manuální testování

Manuální testování zahrnuje testy, které jsou ručně prováděny testovacím týmem podle předem stanovených testovacích scénářů. Jedná se o běžně zaváděný typ v prvopočátku vývoje softwaru. Je realizováno bez použití jakýchkoliv testovacích nástrojů, které nejsou pro analyzování funkčnosti aplikace zapotřebí. V dnešní době je čím dál více nahrazováno automatizovaným testováním, neboť manuální prováděný testů přináší několik nevýhod, např.: vytížení testovacích prostředků i nad triviálním testem, či nutnost disponovat více testery pro paralelizaci testování. Přesto má však manuální testování neocenitelnou výhodu v možnosti provádět exploratory testy.

Automatizované testování

Automatizované testování je proces, kdy jsou testy prováděny automaticky, bez přímé účasti testovacího týmu. Jsou prováděny nástroji, či skriptovací jazyky, používané testovacím týmem. Krom automatického provádění testů lze celý proces paralelizovat – vykonat více testů najednou na různých běhových prostředích ve stejný čas. Díky tomu je možné uvolnit pracovní kapacity testerů pro vykonání jiných testů.



Obrázek 2.3 - Náročnost jednotlivých metodik vůči počtu testů

Pro zavedení automatizace testování je nejdříve potřeba vybrat vhodný testovací nástroj, kterým bude automatizace prováděna. Dále je nutné vytvořit testovací scénáře, dle kterých bude nástroj testovat daný software. Implementace je v začátcích velmi náročná, neboť se musí řádně prozkoumat již zavedený software. To stojí mnoho lidských, finančních i časových zdrojů, jedná se však o investici pro zefektivnění testovacího procesu.

Obrázek 2.3 znázorňuje vytížení firemních zdrojů (zaměstnanci, čas, finance) manuálním a automatickým testováním vůči počtu testů.

2.2.4 Statické a dynamické testy

Statické testování je typ testování, během kterého není zapotřebí běh kódu, ani funkční prototyp aplikace. Většinou se provádí ručně v rámci manuálního testování nebo za pomoci frameworků v případě automatického testování. Zahrnuje kontrolu kódu vůči požadavkům v návrhových dokumentech, procházení kódu a jeho analýzu vůči chybným vzorům či potencionálním chybám, které by v budoucím vývoji mohly nastat.

Dynamické testování naopak vyžaduje spuštěný funkční prototyp, kterému předkládá určité vstupy a ověřuje očekávané výstupy programu. Tento princip je použit ve všech úrovních vývojového cyklu softwaru, jak z pohledu white box testování, tak i black box, které jsou podrobněji rozebrány v oddílu 2.2.9 Testování black/white/gray boxu. [7]

2.2.5 Testy splněním a selháním

Testy splněním a selháním, či také negativní a pozitivní testy, jsou kategorií testů, které předkládají softwaru množinu dat a kontrolují jeho reakci.

V případě pozitivních testů je na vstupu aplikace předložená validní množina dat, u které se očekává, že ji program zpracuje bez chyb a dojde tak ke splnění testu. Příkladem může být vložení věku do formulářové kolonky „Věk:“. V takové situaci je patrné, že kolonka přijme pouze množinu přirozených čísel[6], nikoliv písmena, interpunkční znaménka atd.

Negativní testy oproti tomu předkládají na vstup úmyslně nevalidní množinu dat, kterou program není schopen korektně zpracovat. Cílem testu je ověřit, jestli aplikace odmítne náš vstup. Pokud k tomu nedojde, test selhal. [1]

2.2.6 Regresní testy

Regresní testy ověřují, zda software, který byl již dříve implementován, je stále stejně funkční, jako tomu bylo v době jeho vzniku. Regrese v podstatě znamená stav, kdy úpravou stávajícího kódu vzniknou nové, neočekávané chyby. Pokud je software stále aktualizován – vylepšování funkcionality, optimalizace, úpravy konfigurací atd., musí být zajištěno, že se jeho funkčnost nezmění a bude na vnější podněty reagovat stále stejně. Nejznámějším zástupcem regresních testů jsou jednotkové testy, které regresní testování provádí na té nejzákladnější úrovni softwaru. [9]

[6] Kladná celá čísla

2.2.7 Smoke testy

Kategorie smoke testů je použita na konci úrovně integračního testování. Smoke testy se provádějí před funkčním testováním, zaměřují se především na hlavní funkce programu a ověřují, zdali je aplikace připravena pro další testování. Vzhledem k tomu, že oblast testování smoke testy je malá a testují se pouze ty části softwaru, u kterých se nepředpokládá změna funkcionality, jsou samotné testy většinou automatizovány. [1]

2.2.8 Funkční a nefunkční testy

Funkční testování je typem testů ověřující funkcionalitu v souladu se specifikací požadavku. To je prováděno pomocí black box testování, neboť se neověřuje zdrojový kód softwaru. Testování zahrnuje kontrolu uživatelského rozhraní, databáze, zabezpečení a funkčnost testované aplikace. Lze jej provádět jak automatizovaně, tak i manuálně, a to na úrovních integračního, systémového i akceptačního testování.

Nefunkční testy se zaměřují na nefunkční aspekty vyvíjeného softwaru, mezi které lze zařadit výkon, použitelnost, spolehlivost a další vlastnosti aplikace. Jsou navrženy tak, aby testovaly připravenost systému podle nefunkčních parametrů, kterými se funkční testy nezabývají.

Příkladem takového nefunkčního testu by mohlo být ověření, kolik lidí se může současně přihlásit do softwaru.

S pomocí tohoto testování lze ověřit i připravenost aplikace, zdali a jak bude aplikace fungovat pod větší zátěží, či zda v určitých situacích zbytečně nezatěžuje procesor a paměť běhového zařízení. [1]

2.2.9 Testování black/white/gray boxu

Tento typ klasifikování testování je založen na úrovni znalostí a přístupu testera do vnitřní struktury testovaného softwaru, od čehož se odvíjí rozdílné procesy testování. [4]

Přístup testera je rozdělen do tří barevných spekter, černé – tester nemá o vnitřní struktuře aplikace žádnou znalost, bílé – tester je plně obeznámen o vnitřní struktuře, a šedé – kombinace bílé a černé kategorie, tester má částečné znalosti vnitřní struktury. [3]

Black box testování

V rámci testování black boxu, tester nemá povědomí o vnitřní struktuře kódu, ani o funkcionalitě jednotlivých procesů. Zaměřuje se především na funkcionalitu aplikace jako celku. Testování probíhá z pohledu uživatele, tedy tím způsobem, že je softwaru předložen jeden nebo vícero vstupů, na jejichž základě se kontrolují očekávané výstupy. O průběhu jejich utváření, či lépe řečeno generování, nemá tester přehled. [3]

White box testování

Tester je plně obeznámen o struktuře a funkcionalitě softwaru. Testuje daný software z pohledu vývojáře, v tomto případě jsou jeho znalosti kódu na podobné či stejné úrovni. Má přehled o procesech generující výstupy v reakci na předložené vstupy, proto je v jeho zájmu kontrolovat nejen vygenerované výstupy, ale i samotný průběh jejich utváření. [3]

Gray box testování

Gray box testování je kombinace výše uvedených kategorií. Tester kontroluje kromě obsahu výstupu i procesy, kterými je generován na základě vstupů. Zná tedy roli softwaru, jeho funkcionalitu a do jisté míry i užité algoritmy, avšak už mu není plně známa implementace. [3]

3. Analýza testování ve firmě

Společnost Engel používá pro současný proces integračního a systémového testování testery, kteří je provádí manuálně, bez znalosti vnitřní struktury kódu

nebo modulů, ze kterých se výsledný software skládá. Proto můžeme konstatovat, že se jedná o kategorii – black box testování.

Úkolem testera je řídit se předem danými testovacími scénáři (*test cases*), jenž se skládají z jednotlivých kroků (*test steps*), které tester prochází, a na jejich základě vyhodnocuje stav aplikace. Pro testera se tak jedná o určitý návod, jak software testovat, aniž by musel plně chápat vnitřní implementaci funkcionalit. Testovací scénáře většinou vytváří samotný vývojář, který funkcionalitu vytvořil. Udává tak postup, jakým chce, aby tester jeho práci otestoval.

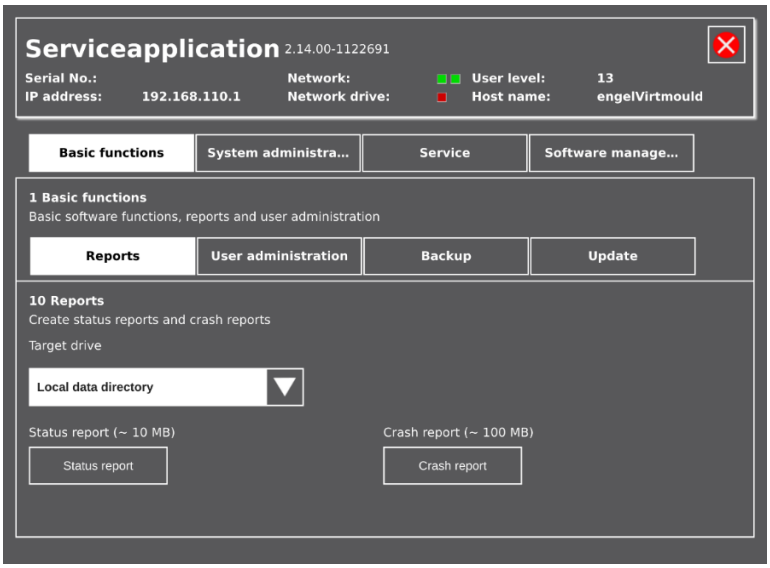
Testovací scénáře jsou tedy vstupem pro testera, jenž používá jednotlivé testovací kroky jako vstup pro testovaný software a očekává pro každý krok i patřičný výstup, který vede k validnímu splnění daného kroku. Výstupem celého testování je pak vyhodnocení, zdali byl test úspěšný či neúspěšný. V případě neúspěšného testu je vývojáři předložena zpráva o testovacím průběhu a o vzniklé chybě, aby mohla být v co možná nejkratším čase opravena.

3.1 Service aplikace

Service aplikace, nebo také servisní aplikace, je hlavním testovacím subjektem v rámci této bakalářské práce. Jedná se o softwarový produkt firmy Engel, komplexní obslužný software provozovaný na specifických linuxových distribucích, které běží na firmou daných hardwarových zařízeních firmy či virtualizovaném nástroji. Účelem tohoto softwaru je zprostředkovat ovládací prostředí mezi operačním systémem, instalací dodatečného softwaru, a uživatelem, jenž daný produkt používá. Produktem se rozumí výrobky společnosti v podobě automatizovaných vstřikovacích strojů a lisů a dodatečných zařízení v podobě robotů.

Uživatelem aplikace je ve většině případů servisní technik ze společnosti Engel, který v rámci softwarové podpory zákazníků obstarává běh, instalaci softwaru, aktualizace či servisní prohlídku výrobních strojů. Pro veškeré tyto činnosti

využívá právě zmíněnou servisní aplikaci, která představuje prostředníka mezi systémem stroje a jeho prostředky, a samotným technikem. Testeři strojů taktéž používají Service aplikaci k zavedení aplikačních balíčků, vlastního testovacího softwaru a dalších doplňků, které umožňují testovat hardware strojů.



Obrázek 3.1 - Ukázka uživatelského prostředí Service aplikace

3.1.1 Klíčové funkce

Neboť Service aplikace je ve své podstatě velmi komplexní software, který zprostředkovává mnoho funkcí najednou, byly zde uvedeny ty nejzákladnější a zároveň nejpoužívanější funkce:

- Nahrazení linuxového CLI[7] vlastním GUI[8] s podporou dotykového ovládání
- Správa linuxových balíčků a repozitářů
- Reportování aktuálního stavu aplikace
- Vytváření a obnovování záloh
- Spouštění dalších aplikací

[7] Command Line Interface – textové uživatelské prostředí
[8] Graphic User Interface – grafické uživatelské prostředí

- Konfigurace síťových připojení
- Správa data a času
- Monitorování běhu zařízení
- Konfigurace a spouštění dalších panelů

3.2 Běhová prostředí

Jak již bylo v předchozí části zmíněno, Service aplikace běží na několika linuxových distribucích, založených na systému Debian, které firma zvolila na základě svých potřeb. Díky tomu je snadné spustit aplikaci na kterémkoliv zařízení, neboť Linux podporuje celou řadu různého hardwaru a je tak snadné na něj software implementovat, především na zařízení používaná firmou.

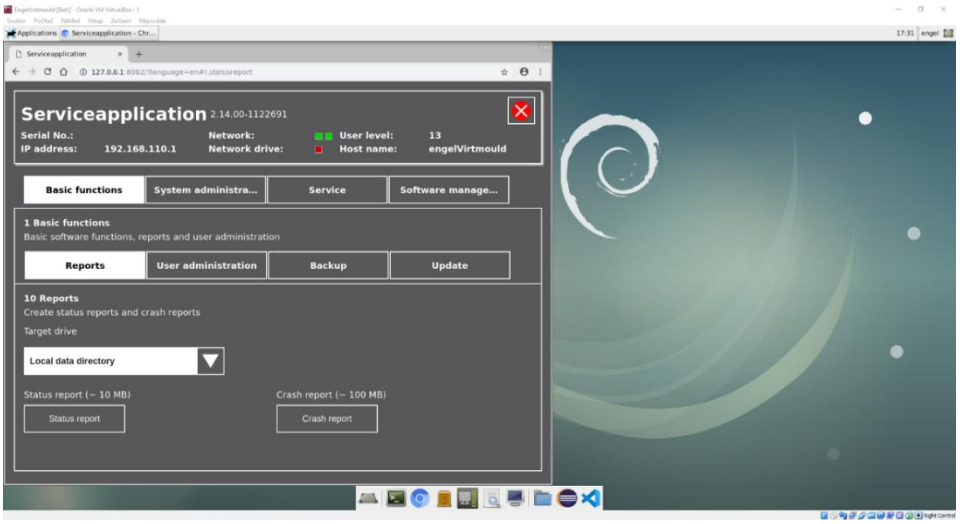
3.2.1 Virtmould

Virtmould je označení souboru komplexních nástrojů, skládající se převážně z VirtualBoxu[9], který virtualizuje linuxový operační systém Debian, na němž je předinstalovaná Service aplikace. Virtmould je dalším softwarovým produktem, který je prodáván stávajícím zákazníkům, jež disponují výrobními stroji firmy Engel.

Účelem tohoto softwaru je virtualizovat konkrétní stroj, aby měl zákazník možnost otestovat si a odladit sekvenci výrobních procesů či nastavit konfiguraci stroje, aniž by musel zmíněné postupy provádět na fyzickém stroji. To ušetří mnoho času a financí, protože každý stroj před spuštěním výroby musí projít technickou prohlídkou a kalibrací výrobního postupu s ostatními stroji (pokud je stroj součástí linky), a to by v případě objevení chyby ve výrobní sekvenci vedlo k opakovaným prohlídkám i kalibracím.

[9] Multiplatformní virtualizační nástroj od společnosti Oracle

Mimo jiné je Virtmould používán vývojáři z vývojového oddělení pro tvorbu a vývoj dalších softwarových produktů, např. zmiňovaná Service aplikace. Využívá se tak výhod, kdy si vývojář může průběžně testovat právě vyvíjený či opravovaný kód na běhovém prostředí podobnému tomu, které běží na koncových strojích.



Obrázek 3.2 - Prostředí Virtmould se spuštěnou Service aplikací

3.2.2 HW Panely

Pro ovládání výrobních strojů jsou používány takzvané panely. Jsou to počítačová zařízení neboli terminály, na kterých běží jedna ze zmíněných linuxových distribucí, která poskytuje běhové prostředí pro Service aplikaci. Firma Engel si panely nechává vyrábět dvěma firmami – Keba a Kontron, od niž panely převzaly označení.

Keba a Kontron jsou po vzhledové stránce takřka identická zařízení. Hlavní rozdílem je hardwarová konfigurace – každý panel disponuje jinou základní deskou, procesorem a kapacitou operační paměti. Dalším rozdílem je použití operačního systému. Panel Kontron používá linuxovou distribuci Debian, kdežto Keba používá vlastní upravenou distribuci založenou na Debianu s názvem Kebian.

Obrázek 3.3 - Přední strana panelu

Ač se to na první pohled nezdá, během testování jsou rozdíly obou zařízení značně patrná. To je důvodem, proč je v některých případech pro každý panel vytvořen vlastní, specifický testovací scénář.

3.2.3 Handheld[10] zařízení

Handheld zařízení, nazývané ve firemním prostředí jako Standalone, je oproti panelům menším ručním zařízením. Hlavní aplikace a operační systém Kebian neběží, na rozdíl od větších panelů, přímo na samotném zařízení ale

[10] Přenosné, či kapesní zařízení

na embedded systému[11], nazývaném Cube, zabudovaném ve výrobním stroji. Připojují se k výrobním strojům v kombinaci s hlavními panely jako doplňkové ovládání, které je možné přenášet na krátké vzdálenosti okolo stroje.

Po vzhledové stránce je Standalone podobný tabletům, nicméně disponuje robustnějším tělem pro ergonomičtější uchopení. Narozdíl od tabletu neobsahuje vlastní akumulátor, a proto je k němu připojen napájecí a datový kabel pro napájení a komunikaci s výrobním strojem. V průběhu vývoje produktové řady CC300 (označení rodiny lisovacích a vstřikovacích strojů) jsou nyní dostupné dva modely standalone zařízení s rozdílným vybavením.

C70

Jedná se o zařízení vyráběné na zakázku Rakouskou firmou Keba. Po hardwarové stránce disponuje C70 rezistivním displejem o velikosti 7 palců. Webovým prohlížečem byl zvolen IceWeasel, odnož Mozilla Firefox, v čemž se zásadně liší od ostatních běhových prostředí, kde byl zvoleno Chromium. V dnešní době se již ustupuje od zavádění tohoto zařízení a spíše se přiklání k modernějšímu nástupci C10.

Obrázek 3.4 - Přenosný panel Keba C70

[11] Specializované vestavěné počítačové zařízení

C10

C10 je v porovnání s C70 podstatně novějším zařízením. Vyrábí jej další Rakouská firma Sigmatek. Namísto 7palcového rezistivního displeje je vybaveno 10palcovým kapacitním displejem a výchozím prohlížečem bylo zvoleno Chromium, stejně jako na ostatních zařízeních. Tak jako v případě C70, i zde běží Service aplikace na zařízení Cube.

Obrázek 3.5 - Přenosný panel Sigmatek C10

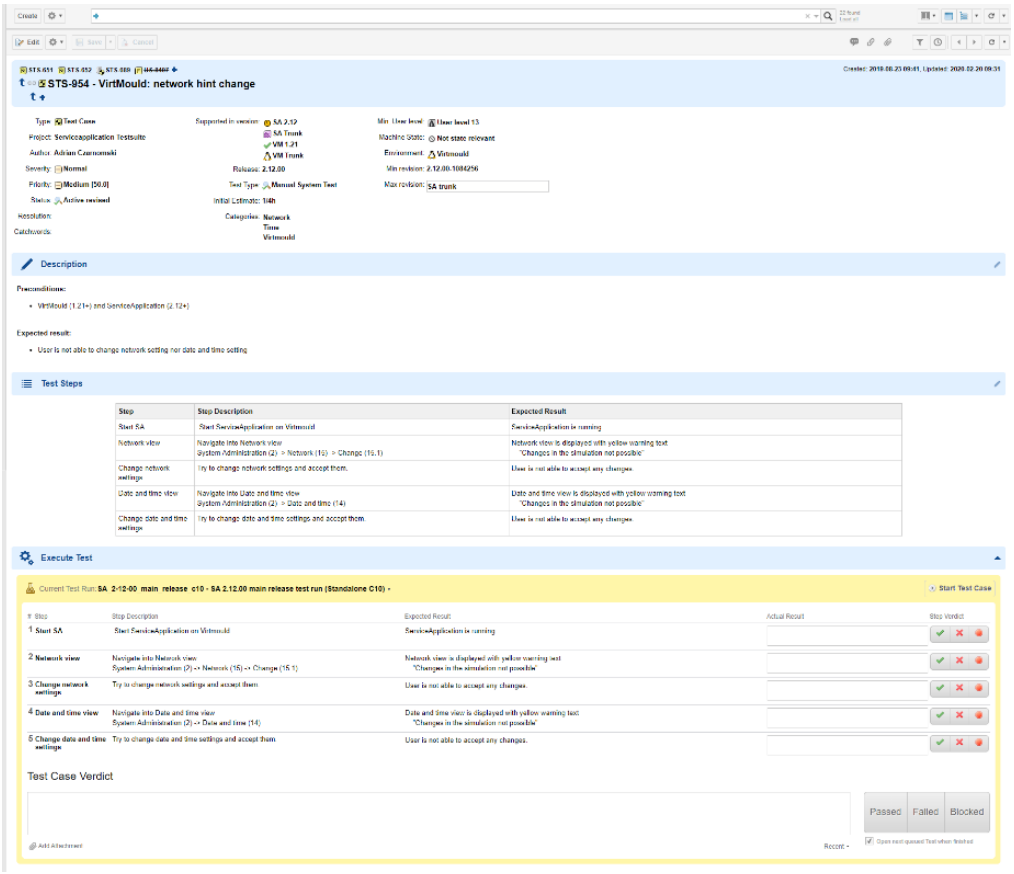
3.3 Test management tool

Test management tool je nástroj, hojně používán testovacím týmem pro organizaci testovacích scénářů, testovacích sad, správu defektů a vazeb mezi nimi. Dalším účelem nástroje je správa testování jednotlivých testovacích sad a scénářů, zaznamenání jejich průběhu a uložení výsledků testování a jejich prezentace. Testerům je tak zajištěn přehled nad samotným testovacím procesem, který tak může být paralelizován mezi vícero testery na různých běhových prostředí.

Ve firmě Engel je jako test management tool používán software Polarion od společnosti Siemens. Jedná se o komerční řešení pro střední a velké podniky, které v sobě kombinuje i nástroje pro agilní vývoj softwaru, správu vývojových

úkolů a nalezených defektů, plánování týmových schůzek, nástroj pro evidenci docházky a mnoho dalšího. Ocenitelnou výhodou je i možnost doplnit chybějící komponenty a funkce rozšířeními.

Jak již bylo zmíněno, Polarion je ve společnosti Engel používán pro mnoho účelů, proto se kapitola dále věnuje již pouze nástrojům určených pro testery.



Obrázek 3.6 - Prostředí testovacího scénáře v nástroji Polarion

3.3.1 Test case

Test case je jedním z work itemů[12], které Polarion umožňuje vytvářet pod jednotlivými projekty. Je to dokumentová stránka skládající se ze několika panelů, přičemž ty nejdůležitější panely jsou – hlavička, popisek, tabulka

[12] Pracovní položka – dokumentační stránka v nástroji Polarion

s testovacími kroky a seznamem pro záznam průběhu jednotlivých kroků test casu. Zpravidla je pokaždé vytváří tester, někdy ve spolupráci s vývojářem, pokud tester nezná veškerá testovací úskalí nové/změněné implementace.

Hlavička v dokumentu obsahuje základní informace, dle kterých je nutné připravit vhodná prostředí pro test. Pro testera jsou nedůležitější tyto informace:

- Podporované verze běhových prostředí
- Minimální úroveň přihlášeného uživatele v Service aplikaci
- Minimální a maximální verze Service aplikace

Mimo jiné se v hlavičce nachází i autor test casu, odhadovaný čas běhu testu, klíčová slova, priorita testu, kategorie testu a další méně podstatné informace o daném test casu.

Description obsahuje krátký srozumitelný popisek k testovacímu scénáři. Kromě popisku může obsahovat i prerekvizity, pro které není v hlavičce políčko, či informace o podobě požadovaného celkového výstupu testu.

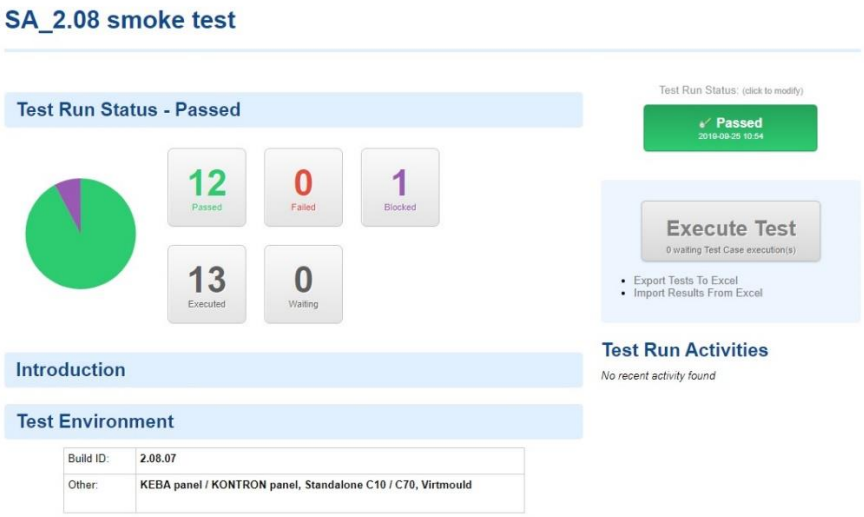
Test steps je panel, jež disponuje tabulkou s jednotlivými testovacími kroky. Tabulka má vždy tři sloupce – *Step* (název kroku), *Step description* (popis kroku) a *Expected result* (očekávaný výsledek po kroku).

Execute Test je seznam kroků, odpovídající jednotlivým řádkům Test steps tabulky. Tester má možnost si vybrat Test run, pod kterým chce test case spustit. Poté následuje testování po jednotlivých krocích. Každý krok je vždy rozdělen podle pěti sloupců – název kroku, jeho popis, očekávaný výsledek, textové políčko pro zadání výsledku kroku a sloupeček s tlačítky. Tlačítka označují, jak skončil průběh daného testovacího kroku. Tester má na výběr ze tří voleb – *Passed* (testovací krok skončil dle očekávání), *Failed* (testovací krok selhal, nicméně můžeme pokračovat v průběhu testovacího scénáře), *Blocked* (testovací krok selhal tak, že není možné dále pokračovat).

3.3.2 Test run

Test run je sadou několika test casů. V prostředí Polarion se jedná o stránku se seznamem jednotlivých scénářů a panelem dashboard, zobrazující celkový průběh testů – počty hotových testů podle výsledku (*Passed, Failed, Blocked*), celkový počet hotových testů (*Executed*) a počet stále neprovedených testů (*Waiting*).

Účelem test runu je seskupit testovací scénáře, aby byly testovány jen určité funkcionality pro daný run. Díky tomu lze seskupit testy používané před releasem[13] nové verze aplikace, smoke testy či testy určené jen pro specifické verze aplikace nebo rozdílná běhová prostředí.



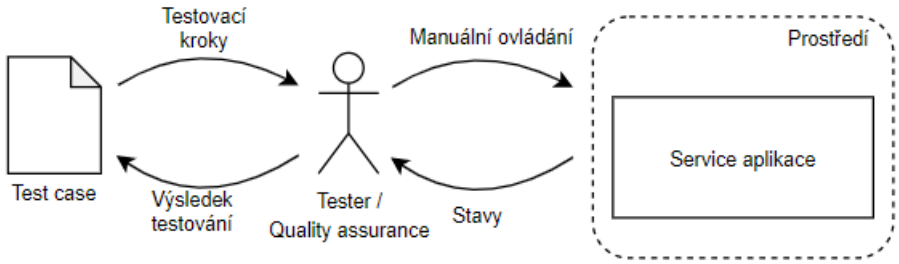
Obrázek 3.7 - Panel Dashboard zobrazující výsledek test runu

3.4 Průběh testování

Tester se v průběhu testování zaměřuje především na své vstupy, které předává testovanému softwaru, a na jejich základě získaných výstupů, které aplikace generuje, viz. Obrázek 3.8. Jedná se o jednoduchý princip, který je však ztížen

[13] Uvolnění pro další zpracování, či zveřejnění

rozsahem jednotlivých testů a komplexitou celkového softwaru, navíc ne vždy je výstup jednoznačný a při nepozornosti může dojít k nevalidnímu splnění testu.



Obrázek 3.8 - Proces dosavadního manuálního testování

Testy jsou z valné většiny prováděny dle testovacích scénářů, uložených v nástroji Polarion. Tester jich využívá především v rámci testování nových implementací, opravených defektů, či validaci stávající funkcionality.

3.4.1 Testování nové implementace

Poté, co vývojáři dokončí implementaci svého kódu, provedou Assembly testy a jednotkové testování. Poté vytvoří testovací scénář a předají jej společně s dokončenou úpravou kódu testovacímu týmu. Testeři na základě testovacích scénářů provádí veškeré kroky, přičemž netestují pouze pozitivní průběh – do aplikace je zadán validní vstup, ale i negativní průběh – je zadán nevalidní vstup. To napomáhá otestovat stabilitu testované části, která by v případě nevalidního vstupu neměla skončit pádem softwaru, ale pouze s upozorněním.

Pokud dojde k chybě, musí být vývojáři patřičně oznámena i s průběhem, který k ní vedl. Vývojář je pak povinen chybu v nejbližším čase napravit a opravenou aplikaci opět předat k testování.

Stejný princip je uplatňován i momentě kdy je testován opravený defekt.

3.4.2 Validace stávající funkcionality

Testeři vykonávají i testy nad stávající funkcionalitou, za účelem ověření, že vše stále funguje. K takovému testování dochází většinou během releasu některé verze Service aplikace nebo při vykonávání smoke testů.

Testeři proto mají pro své potřeby předpřipravené test runy, testující ty části aplikace, které se musí před vydáním zvalidovat. Během změn v kódu totiž mohlo dojít k jejich znefunkčnění.

Průběh je opět stejný jako v testování nové implementace, nicméně v tomto případě se většinou opakují stále ty samé testy.

3.5 Závěr analýzy

Z výše uvedených informací lze usoudit, že ačkoliv úroveň testování na softwarovém oddělení firmy Engel je převážně závislá na manuálním testování, najde se zde prostor pro zavedení automatických testů, a to především v takových situacích, kdy dochází k opakovanému provádění stejných testů. Toto přetestování nejenže vyžaduje plnou investici pracovního času testera nebo zpomaluje celkový průběh test runu, ale především zhoršuje kvalitu softwaru, neboť se tester nemůže věnovat testování právě dokončovaných implementací nových funkcionalit kódu, během kterých by mohl včasně objevit a zpět reportovat nalezené chyby vývojářům.

Zároveň, pokud tester musí testovat více zařízení najednou, sníží počet volných zařízení, která využívají i vývojáři pro jejich vlastní potřeby testování svých implementačních změn.

Další problém se vyskytuje v momentě, kdy je potřeba provést testovací scénář na více verzích Service aplikace. To znamená, že tester bude muset nejen provést ten samý test nad více druhy běhových prostředí, ale i nad různými verzemi aplikace.

Pokud shrneme celkový výsledek současného manuálního testování, lze konstatovat, že provádět testování nad různými verzemi testovaného softwaru a typy jeho běhových prostředí je náročnou operací, jak na testovací prostředky, tak na samotný testovací tým. V praktické části této práce je navrhnuto zlepšení dosavadního testování a praktická aplikace vlastního řešení v podobě testovacího nástroje pro automatizované integrační testování.

4. Postup vlastního řešení

Při návrhu vlastního řešení bylo přihlédnuto k principům používaných v současném procesu testování. Z analýzy současného stavu vyplynulo, že tester se v rámci výstupů testované aplikace zaměřuje převážně na tyto části:

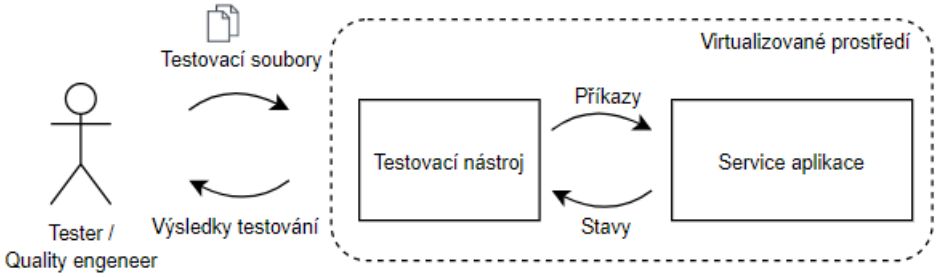
- Zprávy generované do logovacího souboru
- Reakce UI na ovládání testerem
- Obsah vygenerovaných souborů
- Systémová nastavení (síť, datum a čas)
- Správná autorizace přihlášeného uživatele

Veškeré výše uvedené výstupy Service aplikace musel tester doposud kontrolovat manuálně bez pomoci automatizovaného nástroje. To byl zásadní problém, který bylo nutno vyřešit – ušetřit prostředky testera na neustálé manuální procházení testovacími scénáři a opakováním té samé činnosti.

Tato část bakalářské práce se zabývá návrhem a implementací výstupního testovacího nástroje, navrženého pro účely společnosti Engel. Vzhledem k tomu, že nástroj byl navržen pro interní používání ve firmě, nebyly k této bakalářské práci přiloženy zdrojové kódy toolsetu. Namísto toho jsou uvedeny některé příklady kódu.

4.1 Návrh testovacího nástroje

Na základě analýzy současného testování byl navržen softwarový nástroj, který by řešil dosavadní problém častého přetestování již testovaných funkcionalit Service aplikace. Testovací nástroj bylo nutné navrhnout tak, aby byl schopen na základě předdefinovaných testovacích scénářů automaticky provádět jednotlivé testovací kroky, simulovat interakci testera s testovaným softwarem, odhalovat chyby na základě nevalidních výstupů a celkový výsledek testování přehledně prezentovat.



Obrázek 4.1 - Proces navrhnutého automatizovaného testování

V procesu současného testování je tester prostředníkem mezi testovacím scénářem a testovanou aplikací – Obrázek 3.8. Aby bylo docíleno stavu, kdy tester nevykonává jednotlivé kroky testovacích scénářů, nahradil ho v tomto procesu testovací toolset. Ten běží ve stejném běhovém prostředí, jako testovaná Service aplikace. Nástroj přijímá jako svůj vstup testovací soubory získané od testera, taktéž navrženy v rámci této bakalářské práce. Testovací soubory nahrazují v původním procesu manuálního testování jednotlivé testovací scénáře, a skládají se převážně z testovacích příkazů. Nástroj na základě jednotlivých kroků testů provádí interakce nad testovanou aplikací, což jsou jeho výstupy, a zároveň přijímá výstupní stavy Service aplikace. Ty jsou toolsetem následně zpracovány při generování finálního výsledku testování, který je na konci celého procesu prezentován.

Pro vizuální představu navrženého procesu byl přiložen Obrázek 4.1.

4.2 Použité technologie

4.2.1 Java

Pro realizaci výsledného programu byl použit programovací jazyk Java ve verzi 8. Jazyk a verze byly zvoleny na základě používaného JRE[14] ve všech běhových prostředí, ve kterých je program spouštěn.

GSON

GSON je knihovna vytvořená společností Google, určená pro snadnou serializaci a deserializaci objektů v jazyce Java. Knihovna pro serializaci používá standardizovaný a otevřený souborový formát JSON.

JCommander

Další použitou knihovnou je JCommander, který je určen pro snadné extrahování jednotlivých parametrů, zadaných ve spouštěcím příkazu aplikace. Vývojář se tak nemusí starat o složité parsování, pouze označí jednotlivé třídní proměnné, ke kterým chce přiřadit hodnotu parametru. K označení je používána anotace *@Parameter,* přijímající hodnotu *name* pro nastavení názvu parametru. Lze nastavit i mnoho dalších hodnot, jako *description* (popisek parametru), *required* (zda je parametr vyžadován) atd.

JUnit framework

JUnit je testovací framework, určený pro programovací jazyk Java. Jeho účelem je zprostředkovat automatické jednotkové testování pro vývojáře. Vyhodnocování testů probíhá automaticky, na základě porovnávání očekávaných výstupů jednotlivých částí softwaru s jejich předpokládanými výstupy. Výhodou frameworku je plná automatizace celého procesu unit testů

[14] Java Runtime Environment

a také plná podpora v různých vývojových prostředích, například Eclipse, Netbeans, Jetbrains a mnohé další.

4.2.2 Selenium WebDriver

Selenium WebDriver je rozhraní určené k testování webových aplikací. Jeho princip je založen na ovládání webové stránky za pomocí testovacích skriptů, které jsou napsány v mnoha podporovaných jazycích, například C#, Java, Python, PHP atd.

WebDriver pro realizaci testování používá webový prohlížeč, ve kterém jsou jednotlivé kroky testu prováděny nad testovanou webovou stránkou. Mezi prohlížeče, které jsou s tímto nástrojem kompatibilní, patří například Chrome, Firefox, Internet Explorer či Safari.

Selenium používá k vyhledání jednotlivých webových komponent na stránce různé typy identifikátorů – ID elementu, třídy, HTML značky, XPath, CSS selektory či vnořený text elementu. Díky tomu lze snadno lokalizovat jednotlivé komponenty a provádět nad nimi různé akce – kliknutí, vyplnění textem, Drag and Drop, stisk klávesy a jiné.

Nástroj Selenium WebDriver byl vybrán na základě uživatelského prostředí testované Service aplikace, které běží právě ve webovém prohlížeči. Oproti konkurenčnímu řešení, v podobě nástroje Cypress, bylo Selenium vybráno i na základě W3C[15], které jej začalo doporučovat pro automatizaci prohlížečů. [10]

4.2.3 Eclipse

Pro tvorbu testovacího nástroje bylo zvoleno vývojové prostředí Eclipse IDE ve verzi 4.12. Eclipse je zdarma dostupné, open source řešení pro vývojáře

[15] World Wide Web Consortium

nativních i webových aplikací, s širokou škálou podporovaných jazyků a rozšiřujících pluginů.

4.3 Automatický test case

Testovací scénáře byly navrženy jako soubory ve formátu Java. Jedná se o soubor se zdrojovým kódem testovací třídy. Až teprve během procesu testování je třída testovacím nástrojem zkompilována a spuštěna. Pokud by tester disponoval pouze předem zkompilovanou testovací třídou, nebyl by schopen si přečíst její zdrojový kód, neboť by byla přeložena do byte kódu[16].

4.3.1 Struktura

Jak již bylo zmíněno, test case je třída napsaná v programovacím jazyce Java a vychází ze struktury jednotkových testů JUnit. Musí tak splňovat syntaxi daného jazyka, ale i předdefinovanou upravenou strukturu JUnit testů, aby byl výsledný testovací soubor správně zpracován toolsetem.

[16] Nativní kód - instrukční sada jazyka pro Java Virtual Machine

```
import com.engel.cc300.integrationTesting.core.TestCase.*;
import com.engel.cc300.integrationTesting.core.WebUI.*;
import com.engel.cc300.integrationTesting.core.System.*;
import com.engel.cc300.integrationTesting.core.TestClassLoader.TestcaseRunner;

@TestCaseInfo(
   Author="Adrian Czarnomski",
   ReleaseVersion="2.14.00",
   MinRevision="2.08.00",
   MaxRevision="2.14.00",
   SupportedInRevision={2.14, 2.12, 2.10, 2.08},
   MinUserLevel=13,
   Environment={Environment.ALL_CC300_FAMILY},
   MachineState={MachineState.ALL_OPERATIONAL_STATE}
)

@TestCaseDescription(
/**
* Ukázkový kód
*/
)

@RunWith(TestcaseRunner.class)
public class SampleTest extends AbsTestCase {

   @Step(1)
   public void startBrowser(){
    webUITesting.launchBrowser(Browser.CHROME);
   }

   @Step(2)
   public void clickSomeWhere(){
    webUITesting.clickOn("#closeButton");
   }

   @Step(3)
   public void takeScreenshot(){
        webUITesting.takeScreenshot();
   }

}
```

Zdrojový kód 4.1 - Struktura kódu testovacího souboru

Ve vzorové ukázce Zdrojový kód 4.1 si lze povšimnout, že součástí testovacího scénáře je i sada importů. Ty autorovi testu poskytují třídy, pomocí kterých lze volat jednotlivé funkce pro ovládání testované aplikace. Dále třída obsahuje autorem libovolně nazvané testovací metody, představující kroky testovacího scénáře.

4.3.2 Anotace

Nedílnou součástí testovacích tříd jsou anotace, navržené pro účely toolsetu, bez kterých by nebylo možné provést testování. V této části je popis jednotlivých vlastních anotací, jež byly vytvořeny pro testovací scénáře.

TestCaseInfo

TestCaseInfo je anotace pro konfiguraci testovací třídy. Odpovídá hlavičce testovacího scénáře v současně používaném test management nástroji Polarion.

Autor je schopen díky této anotaci určit, pro jaké běhové prostředí je test validní, pro které verze testované aplikace, nebo s jakou úrovní práv má být testování provedeno.

TestCaseDescription

Stejně jako TestCaseInfo, i anotace TestCaseDescription je odvozena od současných manuálních test casů. Pro provedení testu není podmíněná, slouží pouze k zapsání autorova komentáře – popis a účel testu.

Step

Step je anotace, kterou autor testu přiřazuje k jednotlivým testovacím krokům neboli metodám. Bez ní by nebylo možné danou metodu automaticky spustit, neboť by ji nástroj ignoroval. Zároveň anotace označuje i pořadí, v jakém mají být jednotlivé testovací kroky spouštěny. To autorovi testu umožňuje měnit pořadí jednotlivých kroků, aniž by musel zdlouhavě přesouvat celé bloky metod v rámci třídy. Pořadí se určuje číslicí – indexem ve vstupním parametru anotace, dle kterého se testovací kroky před spuštěním testu vzestupně seřadí.

4.3.3 Tvorba testovacích scénářů

Autor má možnost si vybrat ze dvou postupů tvorby testovacích scénářů. Během vývoje toolsetu se předpokládalo, že testovací scénáře budou psány pouze manuálně. S postupem času se však naskytla příležitost pro vývoj nového nástroje, který by umožňoval autorovi vytvářet testy jednodušším způsobem. V této části jsou popsány oba přístupy.

Manuální psaní testovacích scénářů

V případě zvolení manuálního psaní testovacích scénářů je nutné, aby autor disponoval alespoň základními znalostmi programovacího jazyka Java. Kromě jazyka je nutné, aby znal i účel jednotlivých importovaných balíčků, které poskytují třídy a metody pro ovládání a testování Service aplikace či běhového prostředí. Další nutností je disponovat vývojovým prostředím, ve kterém bude test napsán.

Aby autor nemusel psát celou testovací třídu od základu, je v projektu obsažena vzorová třída, kterou může libovolně editovat. Ta již obsahuje potřebné importy, rozšíření třídy a anotace.

Testovací třída se skládá z testovacích metod, které představují jednotlivé test stepy. Hlavička metody by měla být vždy označena specifikátorem přístupu *public* a návratovým typem *void*, dle požadavků JUnit frameworku. Název metody je libovolný, avšak měl by být krátký a dostatečně srozumitelný. Tělo metody by pak mělo obsahovat volání tříd a metod, poskytované testovacímu scénáři za pomocí importů. V neposlední řadě musí být každý testovací krok označen anotací *Step* s indexem označující pořadí, v jakém má být metoda spuštěna.

Nástroj pro tvorbu testovacích scénářů

Tento nástroj poskytuje autorovi testů webové uživatelské rozhraní, ve kterém si pomocí formulářů může snadno vytvořit vlastní testovací soubor, bez sebemenší znalosti programování. Nástroj byl navržen tak, aby výstupní testovací třída splňovala předdefinovanou strukturu, pro její validní zpracování automatickým testovacím toolsetem.

Zmiňovaný nástroj pro tvorbu testovacích scénářů není součástí této práce, neboť se jedná o součást diplomové práce kolegy Bc. Michala Kuchty – *Návrh,*

implementace a integrace nástroje pro vytváření automatických integračních testů, ve které jsou obsaženy podrobnější informace.

4.4 Implementace toolsetu

V této části jsou popsány důležité základní funkcionality navrhnutého testovacího nástroje.

4.4.1 Core

Toolset byl již v počátcích vývoje rozdělen do několika balíčků, aby se oddělily jednotlivé třídy podle jejich zaměření. Základním a jediným nadřazeným balíčkem je *Core*. Obsahuje hlavní spouštěcí třídu *Program*, třídu pro načítání vstupních uživatelských parametrů *PropertiesLoader* a singletone třídu *Property*, která slouží pro uložení načtených parametrů a poskytování jejich hodnot dalším třídám.

Atribut	**Účel atributu**
--test-case-folder	Cesta k adresáři s testovacími soubory.
--results-folder	Cesta k adresáři pro uložení výsledků testování
--drivers-folder	Cesta k adresáři s vlastními webdrivery
--url	Nastavení výchozí url adresy webového prohlížeče
--port	Nastavení výchozího portu url adresy
--sa-log-file	Cesta k logovacímu souboru Service aplikace

Tabulka 4.1 - Seznam atributů spouštěcího příkazu a jejich účel

4.4.2 Dynamické načítání, kompilace a spouštění

Jelikož testovací soubory jsou zpracovány testovacím nástrojem až za jeho běhu, bylo potřebné zajistit jejich dynamické načítání, kompilaci a spouštění. Pro tyto účely byl proto vytvořen balíček *TestClassLoader*, obsahující třídy určené pro jednotlivé kroky.

Loader

Loader je třída, která načítá nezkompilované a neprovedené testovací soubory ze vstupního adresáře. Testovací soubory jsou při jejich načítání rozpoznány koncovkou *.java*. Třída dále prochází obsah podadresářů s již zkompilovanými a provedenými test casy, aby je odfiltrovala a nedocházelo tak k jejich opakovanému načtení. Načtené a odfiltrované soubory jsou nadále předány třídě *Compiler*.

Compiler

Aby byly jednotlivé testovací soubory spustitelné, je potřeba je zkompilovat – zpracovat do byte kódu jazyka Java. K tomuto účelu slouží třída Compiler, které jsou předány načtené testovací soubory a následně zkompilovány za pomocí systémového kompilátoru *JavaCompiler*, získaného na základě třídy *ToolProvider*.[11]

Zkompilované soubory jsou následně uloženy do podadresáře *compiled* s koncovkou *.class*. Odtud jsou dále načteny již jako instance za pomocí metody *newInstance* poskytovanou třídou *URLClassLoader*.[12]

Pokud testovací třída obsahuje chybu v syntaxi, nedojde k její kompilaci, a tedy ani ke spuštění. Výsledek testu je pak označen jako Failed a do chybových hlášek je zapsán důvod chyby, získaný z instance systémového kompilátoru.

Executor

Exekutor je třída, starající se o spouštění jednotlivých instancí testovacích tříd. Pro běh testů byla použita třída *JUnitCore* z frameworku JUnit, která umožňuje skrze programovací kód spouštět jednotlivé metody v test casu a vracet výsledky jejich provedení, čehož je využito pro vyhodnocení jednotlivých testů na konci procesu testování. Výsledky testů jsou průběžně ukládány do instancí třídy *TestResult*, která je popsána v následující kapitole.

4.4.3 Report

Balíček Report obsahuje třídy, které slouží k zaznamenávání výsledků test casů.

Pro ukládání výsledků jednotlivých testovacích scénářů slouží instance třídy *TestResult*. Objekty této třídy jsou vytvářeny a přiřazovány k testům před jejich spuštěním. Teprve po dokončení průběhu testovací třídy jsou její výsledky zaznamenány do přiděleného TestResultu.

Objekt TestResult disponuje seznamy varovných a chybových hlášení, které se mohly naskytnou během procesu testování. Dále obsahuje informace s názvem testovacího scénáře, jak dlouho trvalo provedení testu, či zda byl test dokončen úspěšně, neúspěšně, nebo byl přeskočen.

Jelikož instance TestResult je vždy výsledkem jen jednoho testu, bylo nutné výsledky všech testů seskupit a doplnit dodatečné informace v rámci test runu. K tomu slouží třída *ReportMaker*. Objekt této třídy průběžně ukládá do svého ArrayListu jednotlivé výsledky test casů.

Po dokončení všech testovacích tříd je v instanci ReportMakeru zavolána metoda *generateFile*, která tuto instanci za pomoci frameworku GSON serializuje do souborového formátu JSON, jež je následně uložen do výstupního adresáře nástroje. Na vzorovém příkladu Zdrojový kód 4.2 je zobrazena struktura tohoto souboru.

```
{
    "Tests info": {
        "Passed": 2,
        "Skipped": 0,
        "Failed": 0
    },
    "Test results": [
        {
            "Test Case": "StatusReportTest",
            "Status": "PASSED",
            "Started at": "Mar 20, 2020 12:10:18 AM",
            "Ended at": "Mar 20, 2020 12:15:25 AM",
            "Time(ms)": 307271.0,
            "Warning messages": [],
            "Error messages": [],
            "Screenshots": [
                "screenshot7947601072181987149.jpg"
            ]
        },
        {
            "Test Case": "CreateNewUserTest",
            "Status": "PASSED",
            "Started at": "Mar 20, 2020 12:15:25 AM",
            "Ended at": "Mar 20, 2020 12:17:26 AM",
            "Time(ms)": 121345.0,
            "Warning messages": [],
            "Error messages": [],
            "Screenshots": []
        }
    ],
    "Total time(ms)": 428616.0
}
```


Zdrojový kód 4.2 - Struktura reportu s výsledky dvou test casů

4.4.4 WebUI

WebUI je balíček obsahující třídu *WebUITesting*, která autorovi testu poskytuje metody pro práci s webovým rozhraním Service aplikace. Funkcionalita je založena na frameworku *Selenium WebDriver*, který slouží pro ovládání webových prezentací z programovacího jazyka. Účelem třídy WebUITesting bylo vytvořit metody, které plně nahradí testerovu interakci s UI.

Mezi nejdůležitější poskytované metody patří:

- *launchBrowser* – otevření URL adresy ve webovém prohlížeči
- *clickOn* – kliknutí na prvek ve webové stránce
- *isVisible* – ověření viditelnosti prvku na stránce
- *takeScreenshot* – pořízení a uložení snímku obrazovky

```
public void clickOn(String id) {
    try {
        if (!id.contains("#")) {
            driverWait.until(
               ExpectedConditions.elementToBeClickable(By.xpath(id)));
            driver.findElement(By.xpath(id)).click();
        } else {
            driverWait.until(
               ExpectedConditions.elementToBeClickable(By.id(id)));
            driver.findElement(By.id(id)).click();
        }
    } catch (NoSuchElementException | TimeoutException |
               NullPointerException e) {
        logger.log(Level.WARNING, e.toString());
    }
    logger.log(Level.INFO, "Element {0} has been pushed.", id);
}
```

Zdrojový kód 4.3 - Metoda pro kliknutí na prvek stránky

Pro určení, se kterým prvkem stránky má být provedena interakce, byly zvoleny dva typy selektorů. Prvním z nich je jedinečné ID elementu, které je však v Service aplikaci málo používané. Proto byl jako druhý selektor zvolen XPath.

4.4.5 System

Tento balíček byl vytvořen pro interakci s běhovým prostředím Service aplikace a jeho kontrolu. Skládá se ze tříd, které provádí terminálové příkazy nad linuxovým operačním systémem nebo manipulaci se soubory.

SystemUtilities

SystemUtilities je třída, která umožňuje ověřit nainstalované systémové balíčky a jejich verze. To slouží především ke kontrole, zda Service aplikace a běhové prostředí odpovídá konfiguraci testovacího scénáře.

Volaný příkaz a kontrola terminálového výstupu jsou vždy prováděny na základě volané metody, proto tak není potřeba, aby autor jednotlivé příkazy znal.

Pro testera je nejpodstatnější metoda *getPackage*, která podle názvu vyhledává konkrétní systémový balíček a ověří, zda je nainstalován a v jaké verzi.

FileManager

Pro manipulaci se soubory v rámci systému slouží třída FileManager, využívající rozhraní jazyka Java – *java.io*. Autor testu tak může ověřovat a editovat obsah souborů, kontrolovat obsah adresářů, přesouvat a mazat soubory nebo vytvářet nové s libovolným obsahem. Jedná se o ty samé akce, doposud prováděné testery manuálně.

ConnectionControl

Poslední zahrnutou třídou v balíčku je ConnectionControl. Jejím účelem bylo umožnit v test casu konfigurovat síťová rozhraní systému, jako nastavení IP adresy a masky podsítě adaptéru, kontrolu připojení, vypnutí a zapnutí konkrétního adaptéru či ověření jeho konfigurace. Autor opět nepotřebuje znát linuxové příkazy pro konfiguraci sítě, neboť ty za něj provádí jednotlivé metody na základě vstupních parametrů.

4.5 Použití testovacího toolsetu

4.5.1 Požadavky pro testování

Než tester bude moct spustit automatické integrační testování, je potřebné splnit požadavky ze strany navrženého testovacího procesu.

Tester musí disponovat *virtualizovaným běhovým prostředím*, pro které chce testování vykonat. Prostředí jsou již poskytována firmou. Virtuální stroj musí zároveň disponovat *předinstalovanou Service aplikací* v požadované verzi, která má být testována. Dále je potřeba nahrát do systému *testovací toolset* společně s adresářem obsahující vybrané *test casy* pro plánovaný test run.

4.5.2 Spuštění testování

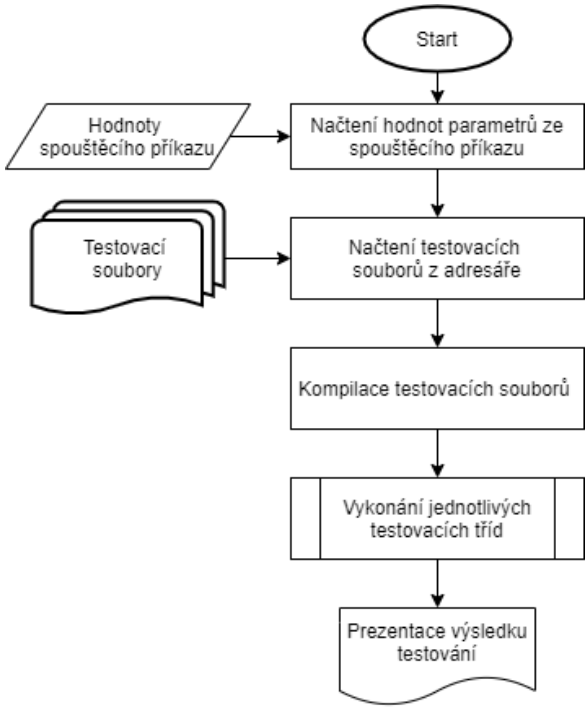
Pokud jsou požadavky pro testování splněny, je před spuštěním nutné testovací nástroj správně nakonfigurovat. Vyžaduje se, aby byl specifikován vstupní

adresář s testovacími soubory a výstupní adresář pro výsledky testování, jako je tomu v ukázce Zdrojový kód 4.4. Dále je možné volitelně nakonfigurovat i další parametry, které jsou zmíněny v oddíle 4.4.1 Core.

```
java -jar core.jar --test-case-folder "/var/lib/engel/testCases"
      --results-folder "/var/lib/engel/results"
```

Zdrojový kód 4.4 - Vzorový příklad pro spuštění testovacího nástroje

Po zadání příkazu je spuštěn testovací nástroj. Testovací průběh byl pro zpřehlednění znázorněn ve flow diagramu Obrázek 4.2.



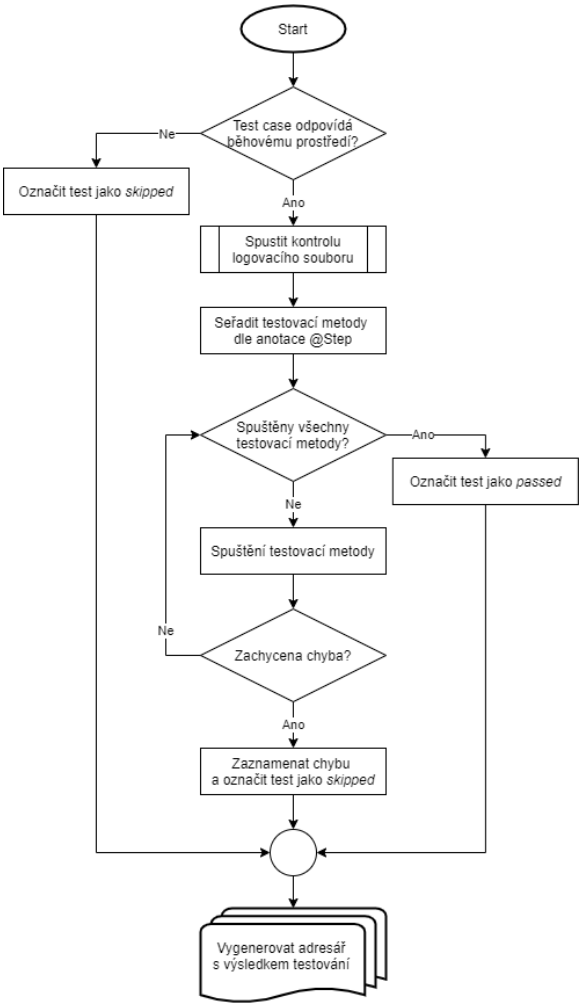
Obrázek 4.2 - Vývojový diagram průběhu spuštění toolsetu

Průběh spuštění

Po spuštění nástroje se provede počáteční inicializace hodnot ze vstupních parametrů spouštěcího příkazu. Následuje načtení testovacích souborů ze vstupního adresáře vybraného uživatelem. Načtené soubory se dále zkompilují, přičemž je provedena i validace syntaxe jazyka. Každá zkompilovaná testovací

třída je pak individuálně spuštěna v průběhu testování. Samotný postup testování je již plně v režii test casů, na základě vlastního scénáře, viz. Obrázek 4.3.

Vyhodnocení testovacího scénáře probíhá až po jeho dokončení. V momentě, kdy jsou hotovy všechny testy v test runu, je ze všech výsledků vygenerován výstupní reportovací soubor obsahující informace o průběhu a výsledku daného test runu.

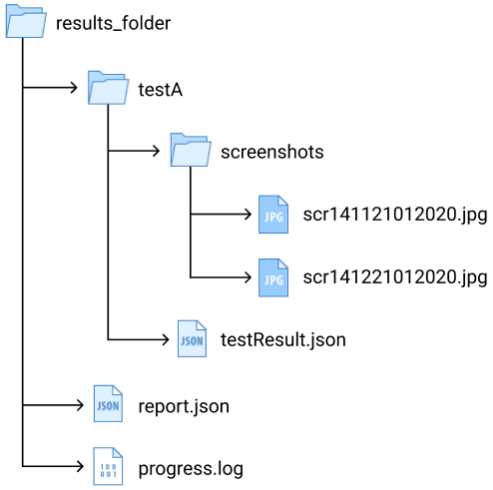


Obrázek 4.3 - Flow diagram průběhu test casu

4.5.3 Výstup testování

Výstupní adresář je složkou, která slouží pro ukládání vygenerovaných výstupů během testování. Obrázek 4.4 znázorňuje vnitřní strukturu adresáře obsahující položky:

- adresář pro jednotlivé test casy, zahrnující soubor s výsledkem pro daný test, a adresář pro pořízené snímky obrazovky
- reportovací soubor představující konečný výsledek test runu
- logovací soubor toolsetu, jehož obsahem jsou zprávy spuštěných procesů během testování



Obrázek 4.4 - Struktura výstupního adresáře s výsledky testování

4.6 Prezentace výsledku testování

Výsledek testování je zapsán jako textový soubor ve formátu JSON. Za účelem přehledné prezentace dat v grafické podobě byla navržena webová stránka prezentující data z reportu ve webovém prohlížeči. Aby byla jednotlivá data

z reportovacího souboru zobrazena, byl použit scriptovací webový jazyk JavaScript, který podporuje zmiňovaný JSON formát.

Po dokončení a vyhodnocení testování mnou navrženým testovacím toolsetem, je stránka (běžící lokálně) zobrazena v nástroji pro spouštění virtualizovaných běhových prostředí, vytvořeném kolegou Michalem Kuchtou v rámci jeho diplomové práce.

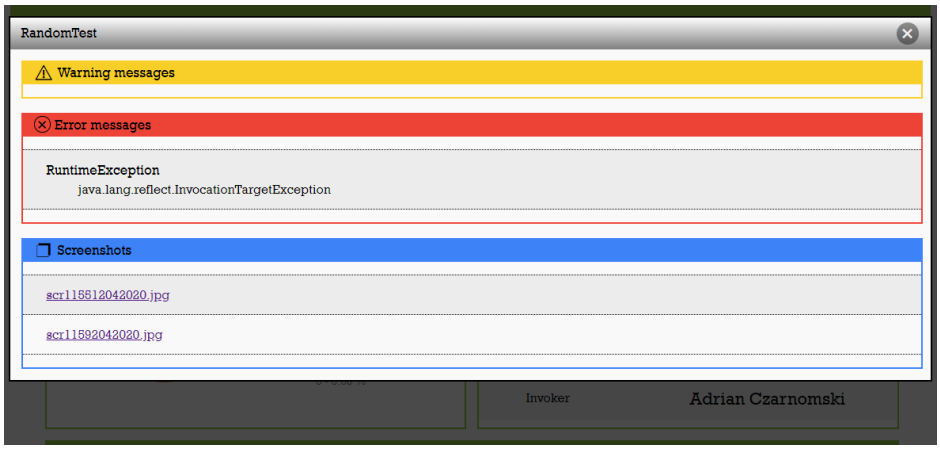
4.6.1 Vzhled stránky

Stránka se skládá ze čtyř základních panelů:

- *Summary testcases* zobrazující tabulku s výsledky jednotlivých test casů
- *Test run overview*, jež obsahuje graf znázorňující poměry výsledných stavů testů
- *Test run summary* pro prezentaci základních informací test runu
- *Progress log* – textový panel pro výpis logovacího souboru toolsetu

Obrázek 4.5 - Grafická prezentace výsledků testování

Po rozkliknutí jednotlivého testu v tabulce je zobrazeno modální okno obsahující výpisy jednotlivých varovných a chybových hlášek a seznam snímků obrazovky, pořízených v daném testu.



Obrázek 4.6 - Modální okno zobrazující výpis testu

5. Zhodnocení

Nástroj byl poskytnut hlavnímu testerovi na softwarovém oddělení, jenž výsledný produkt zhodnotil na základě ověření funkcionality během interního automatického testování. Pro tento účel bylo vytvořeno celkem pět automatizovaných test casů, které byly použity na požadovaných běhových prostředích. Vzhledem k tomu, že testovací proces a jeho výsledky dopadly dle očekávání, byl nástroj testerem označen jako žádoucí prostředek pro plánovanou automatizaci testování.

Zároveň proběhlo i otestování funkčnosti nástroje v rámci diplomové práce kolegy Michala Kuchty, jenž testovací toolset použil během vlastního testování svého navrhnutého softwaru.

5.1 Možnosti dalšího rozvoje

I přestože je nástroj již v dosavadní podobě plánován pro zavedení do procesu automatického testování, naskytla se již během vývoje toolsetu několik uplatnitelných vylepšení.

Jedna z budoucích funkcionalit je umožnění test casům restartovat běhové prostředí a automaticky pokračovat po znovunačtení v místě, kde došlo k přerušení testovacího procesu.

Dalším plánovaným vylepšením je přesun testovacího nástroje, běhových prostředí a testované Service aplikace na vzdálený server. Záměrem je uvolnit požadavky na testera – disponovat podmíněnými nástroji pro automatické testování. Zároveň tím společnost získá řešení, umožňující provádět testování na dálku, například pro jiná vzdálená oddělení.

6. Závěr

Výsledkem bakalářské práce bylo vytvoření testovacího nástroje, který byl na míru vytvořen pro potřeby testovacího týmu na softwarovém oddělení ve firmě Engel strojírenská spol. s.r.o.

Byla provedena analýza současného procesu provádění testů, na jejímž základě byly do nástroje implementovány funkcionality umožňující automatizovat základní akce manuálně prováděné testerem.

Součástí práce je i vlastní návrh struktury nových testovacích scénářů, zpracovávaných výstupním toolsetem. Toolset umožňuje testovacímu týmu uvolnit své pracovní nasazení nad často opakujícími se integračními testy, které tak mohou být automaticky provedeny bez přímé účasti testera.

Dále byl navržen výstup neboli výsledný report procesu testování, jehož součástí je i grafická prezentace ve webovém prohlížeči, taktéž vytvořena v rámci této práce.

Nástroj pro automatické integrační testování je v současné době ve zkušebním procesu, kdy testovací tým vytváří testovací soubory a ověřuje praktičnost řešení. Na základě úspěšnosti testovacího nástroje lze očekávat jeho následné zavedení do ostrého procesu automatizace testování ve společnosti.

Seznam použité literatury

[1] HLAVA, Tomáš. Druhy, typy a kategorie testů. In: Testování softwaru [online]. 21.8.2011 [cit. 2020-01-04]. Dostupné z: http://testovanisoftwaru.cz/category/metodika-testovani/druhy-typy-a-kategorie-testu/

[2] HLAVA, Tomáš. Fáze a úrovně provádění testů. In: Testování softwaru [online]. 21.8.2011 [cit. 2020-01-06]. Dostupné z: http://testovanisoftwaru.cz/tag/integracni-testovani/

[3] SAUNOIS, Lucie. Black box, grey box, white box testing: what differences? In: NBS System [online]. Paříž: NBS System, 2016 [cit. 2020-01-14]. Dostupné z: https://www.nbs-system.com/en/blog/black-box-grey-box-white-box-testing-what-differences/

[4] KELLY, Mike. Integration testing: Is it black box or white box testing? In: TechTarget: SearchSoftwareQuality [online]. 11.11.2008 [cit. 2020-01-14]. Dostupné z: https://searchsoftwarequality.techtarget.com/answer/Integration-testing-Is-it-black-box-or-white-box-testing

[5] KOKAB, Usman. What is Software Testing? Itechnism [online]. 2019 [cit. 2020-02-18]. Dostupné z: https://itechnism.com/tutorials/what-is-software-testing-and-why-it-is-important/

[6] Integration Testing: What is, Types, Top Down & Bottom Up Example. Guru99 [online]. [cit. 2020-02-20]. Dostupné z: https://www.guru99.com/integration-testing.html

[7] Static Testing vs Dynamic Testing: What's the Difference? Guru99 [online]. [cit. 2020-02-22]. Dostupné z: https://www.guru99.com/static-dynamic-testing.html

[8] KITNER, Radek. Exploratory testy. Kitner [online]. [cit. 2020-04-03]. https://kitner.cz/slovnik/exploratory-testy/

[9] KITNER, Radek. Regresní testování. Kitner [online]. [cit. 2020-04-12]. Dostupné z: https://kitner.cz/testovani_softwaru/regresni-testovani/

[10] Selenium's WebDriver is now a W3C Recommendation! Software Freedom Conservancy [online]. [cit. 2020-04-17]. Dostupné z: https://sfconservancy.org/news/2018/may/31/seleniumW3C/

[11] Class ToolProvider. In: Oracle: Java Platform Standard Edition 7 Documentation [online]. [cit. 2020-04-20]. Dostupné z: https://docs.oracle.com/javase/7/docs/api/javax/tools/ToolProvider.html

[12] Class URLClassLoader. In: Oracle: Java Platform Standard Edition 7 Documentation [online]. [cit. 2020-04-20]. Dostupné z: https://docs.oracle.com/javase/7/docs/api/java/net/URLClassLoader.html

Seznam obrázků

Seznam ukázkových kódů

Seznam tabulek