Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Bakalářská práce

Automatizace testování a kvalita softwaru

Jakub Váňa

© 2020 ČZU v Praze

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Jakub Váňa

Systémové inženýrství a informatika
Informatika

Název práce

Automatizace testování a kvalita softwaru

Název anglicky

Test automation and software quality

Cíle práce

Hlavním cílem bakalářské práce bude vybrat vhodný proces automatického testování softwaru a následně ověřit jeho nasazení ve střední firmě.

Dílčí cíle bakalářské práce:
- analýza literárních zdrojů
- návrh konkrétního postupu testování a stanovení hypotéz
- experimentální ověření a měření navrženého postupu

Metodika

Teoretická část práce je založena na studiu odborné literatury týkající se způsobů automatizovaného testování softwaru a možností měřitelnosti, vyhodnocování kvality softwaru. Poznatky z analýzy budou sloužit jako výchozí bod pro praktickou část.

V rámci praktické části bude vypracován a ověřen business-case pro vytvoření a údržbu automatických testů ve střední firmě zabývající se vývojem softwaru, implementace sady testů na základě navrženého modelu a nastavení možných faktorů pro měřitelné sledování vývoje kvality softwaru. Na základě výsledků testování bude stanovena vhodnost zvoleného postupu a formulovány závěry práce.

Doporučený rozsah práce

35-50

Klíčová slova

Vývoj softwaru, automatizace testování, kvalita softwaru

Doporučené zdroje informací

BUREŠ, M. – RENDA, M. – DOLEŽEL, M. – Efektivní testování softwaru, 2016. ISBN 978-80-247-5594-6.
ELFRIEDE, D. – Implementing Automated Software Testing, 2009. ISBN 0321580516.
FEWSTER, M. – Software Test Automation, 1999. ISBN 9780201331400.
LEWIS, W E. – VEERAPILLAI, G. *Software testing and continuous quality improvement.* Boca Raton: Auerbach Publications, 2005. ISBN 0849325242.
ROUDENSKÝ, P. – Kvalita softwaru TEORIE A PRAXE, 2018. ISBN 978-80-7402-322-4.

Předběžný termín obhajoby

2020/21 LS – PEF

Vedoucí práce

Ing. Jan Pavlík

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 29. 7. 2020

Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 19. 10. 2020

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 08. 11. 2020

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Automatizace testování a kvalita softwaru" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 21.02.2021 ____________________________

Poděkování

Rád bych touto cestou poděkoval Ing. Janu Pavlíkovi za velmi vstřícnou pomoc při tvorbě a zpracování této bakalářské práce.

Automatizace testování a kvalita softwaru

Abstrakt

Tato bakalářská práce se zabývá procesem automatizace testování softwaru, možnostmi, které následně automatizace přináší a kvalitou softwaru, která je testováním do výrazné míry ovlivněna.

Hlavním cílem práce je na základě analýzy vybrat vhodný proces automatického testování softwaru a následně ověřit tento proces nasazením ve střední firmě. Na základě zpracované analýzy v teoretické části je definována strategie pro koncept automatických testů. Poté jsou vytvořeny automatické testy pro konkrétní softwarový produkt. Jsou zde také popsány aspekty kvality softwaru a modely, které tyto aspekty definují. Dále je součástí práce vypracovaný business-case pro vytvoření a údržbu těchto automatických testů. Business-case je primárně určený k obhajobě nákladů spojených právě s implementací a údržbou automatických testů.

Úspěšnost tohoto procesu bude vyhodnocena na základě měření aspektů kvality softwaru definovaných v teoretické části práce.

Klíčová slova: vývoj softwaru, kvalita softwaru, testování, automatické testy, automatizace testování, měření kvality, klíčové ukazatele výkonnosti

Test automation and software quality

Abstract

This bachelor thesis describes the process of automating software testing, the possibilities that automation brings and software quality, which is affected by the testing.

The main goal of the work is based on the analysis to select a suitable process to automatically test the software and then verify this process by deploying in a medium-sized company. Based on the analysis in the theoretical part, a strategy for the concept of automatic tests is defined. Then automatic tests are created for a specific software product. Bachelor thesis also describes aspects of software quality and the models that define these aspects. Furthermore, a part of the work is developed the business case for the creation and maintenance of these automatic tests. The business case is primarily intended to defend the costs associated with the implementation and maintenance of automated tests.

The success of this process will be evaluated by measuring aspects of software quality defined in the theoretical part of the work.

Keywords: software development, software quality, testing, automatic tests, test automation, quality measuring, key performance indicators

Obsah

1 Úvod ... 11

2 Cíl práce a metodika ... 13

3 Teoretická východiska ... 14
3.1 Testování softwaru a základní pojmy ... 14
3.2 Automatizace testování ... 15
3.2.1 Testovací prostředí ... 15
3.2.2 Strategie automatického testování ... 15
3.2.3 Návrh architektury ... 17
3.3 Typy automatických testů ... 17
3.3.1 Jednotkové testy ... 17
3.3.2 Testy uživatelského rozhraní ... 19
3.3.3 Integrační testy ... 20
3.4 Pokrytí kódu jednotkovými testy ... 21
3.5 Efektivní využití automatických testů ... 22
3.6 Náklady spojené s automatizací ... 23
3.6.1 Vývoj testů ... 23
3.6.2 Údržba testů ... 23
3.7 Nejčastější problémy spojené s automatizací ... 24
3.8 Kvalita softwaru ... 24
3.8.1 Předpoklady pro kvalitní software ... 24
3.8.2 Modely kvality ... 24
3.8.3 Normy pro oblast kvality softwaru ... 27
3.8.4 Kvalita kódu dle SOLID ... 27
3.8.5 Měřitelnost kvality softwaru ... 29

4 Vlastní práce ... 30
4.1 Výběr strategie ... 30
4.1.1 Koncepce testů ... 30
4.1.2 Požadavky ... 30
4.2 Business-case pro automatické testy ... 30
4.3 Stanovení hypotéz ... 32
4.4 Implementace testů ... 32
4.5 Prostředí spouštění automatických testů ... 32
4.5.1 Zásahy nepovolaných osob ... 33
4.5.2 Změny v operačním systému ... 33
4.5.3 Vyskakování oken během průběhu testů ... 33
4.6 Běh testů ... 34
4.7 Údržba automatických testů ... 34

4.8 Sledování počtu defektů 35
4.8.1 Analýza již evidovaných defektů v systému 35
4.8.2 Grafy s vývojem počtu defektů 36
4.8.3 Graf se zohledněnou závažností 37
4.8.4 Graf se zohledněným obchodním dopadem 38
5 Výsledky 39
5.1 Ověření platnosti odhadů 39
5.1.1 Odhady pro implementaci 39
5.1.2 Odhady pro údržbu 39
5.2 Počet nalezených chyb automatickými testy 39
5.3 Ověření hypotéz 40
5.4 Stanovení metriky pro sledování kvality 40
6 Závěr 41
7 Seznam použitých zdrojů 42

Seznam obrázků

Obrázek 1 – cyklus vývoje softwaru rozdělený do jednotlivých fází [6] 14
Obrázek 2 – jednotlivé kroky při plánování strategie [7] 16
Obrázek 3 – pyramida struktury automatických testů [8] 17
Obrázek 4 – implementace jednoduchého jednotkového testu 18
Obrázek 5 – vizualizace pokrytí kódu [11] 22
Obrázek 6 – křivka nákladů pro manuální i automatické testování [9] 23
Obrázek 7 – definice aspektů kvality softwaru [10] 25
Obrázek 8 – SOLID akronym [12] 28
Obrázek 9 – graf s počtem všech evidovaných defektů 36
Obrázek 10 – graf s počtem vydaných defektů 36
Obrázek 11 – graf znázorňující vývoj defektů z pohledu závažnosti 37
Obrázek 12 – graf znázorňující vývoj defektů z pohledu obchodních dopadů 38

Seznam tabulek

Tabulka 1 – odhadované náklady na vývoj automatických testů 31
Tabulka 2 – odhadované náklady na údržbu automatických testů 31
Tabulka 3 – odhadované náklady na manuální testování 31
Tabulka 4 – reálné náklady na implementaci 32
Tabulka 5 – náklady na modifikace testů v období 3 měsíců 34
Tabulka 6 – porovnání původních odhadů vůči reálným nákladům 39

Seznam použitých zkratek

ISTQB – International Software Testing Qualifications Board, nezisková organizace působící na mezinárodní úrovni jako certifikační rada v oblasti testování softwaru

UI – User Interface, uživatelské rozhraní

UX – User Experience, uživatelská zkušenost

SOA – Service Oriented Architecture, obecný návrhový architektonický vzor z oblasti vývoje softwaru
MD – Man Day, pracovní den jedné osoby (8 pracovních hodin)
USB – Universal Seriál Bus, rozhraní pro připojení periferií k počítači
IT – Information Technology, informační technologie

1 Úvod

Testování softwaru je v dnešní době stále více rozebíranou problematikou. Hlavním důvodem je nárůst počtu digitálních zařízení, které jsou takřka všude kolem nás. Každé toto zařízení má svůj účel a očekáváme od něj určité chování. Abychom byli konkurenceschopní v době, kdy již existuje celá řada softwarů, musíme zajistit, že námi vyvinuté zařízení bude fungovat dle definovaných požadavků, budeme se na něj moci spolehnout a jeho uživatelé ho budou s oblibou používat. Kvalitu produktu a jeho funkčnost je tedy nezbytně nutné během procesu vývoje vhodně navrženým způsobem validovat.

Jak toho ale co nejefektivněji dosáhnout? Je to jednoznačně testování, které tvoří jednu z hlavních částí vývojového cyklu softwaru. Testování umožňuje získat informace o stavu produktu a dává určitou jistotu ještě před tím, než dojde k uvolnění daného zařízení/softwaru.

Pokud jsme však schopni tento proces automatizovat, dostáváme úplně nové možnosti, jak k této problematice přistoupit. Jednou z výhod může být například výrazně rychlejší odhalování defektů v softwaru vzniklých vývojem nebo ještě efektivnější možnost upozornit na chybu už přímo vývojáře a zamezit tak šíření chyby do dalších verzí softwaru. Beze sporu je v případě efektivního využití procesů automatického testování další výhodou velká úspora času. Automatickým testováním jsme také schopni dosáhnout minimalizace takzvané slepoty, která může u testera vznikat častým opakováním stejných manuálních testovacích scénářů. Všechny tyto výhody ale samozřejmě nejsou zadarmo. S automatickým testováním je spojena řada nákladů na vývoj, a hlavně také na jejich údržbu. V některých společnostech může být náročné automatizované testování vůbec obhájit už jen po vyčíslení nákladů na vývoj automatických testů. Pokud je však naším cílem budovat robustní a kvalitní software bez alespoň minimálního zastoupení automatických testů už se neobejdeme, obzvlášť u systémů, kterým narůstá poměrně razantně komplexita.

Testování se výrazně projevuje do následné kvality softwaru, respektive do některých z aspektů kvality softwaru, proto je velice důležité k němu zodpovědně přistupovat. Problematické mnohokrát bývá už v počátku dokázat kvalitu produktu vůbec definovat. Na kvalitu lze nahlížet z různých pohledů a každý uživatel si pod tímto pojmem může představit trochu jiné aspekty. Pro některé uživatele může být vhodnější, že celý systém bude fungovat svižně a občas se někde projeví důsledek neodhaleného defektu, pro jiné uživatele toto může být naprosto neakceptovatelné. Mohou naopak vyžadovat vysokou spolehlivost a rychlost pro ně nemusí být klíčová.

Celá problematika bude tedy rozebrána podrobněji s tím, že v praktické části proběhne experimentální ověření funkčnosti některých základních popsaných principů. Zaměříme se také na to, jaké možnosti nám automatické testy přináší a jaké jsou s nimi spojené případné komplikace.

2 Cíl práce a metodika

Hlavním cílem bakalářské práce je vybrat vhodný proces automatického testování softwaru a následně ověřit jeho nasazení v praxi. Dílčími cíli jsou analýza literárních zdrojů, návrh konkrétního postupu testování, stanovení hypotéz a experimentální ověření efektivity navrženého postupu.

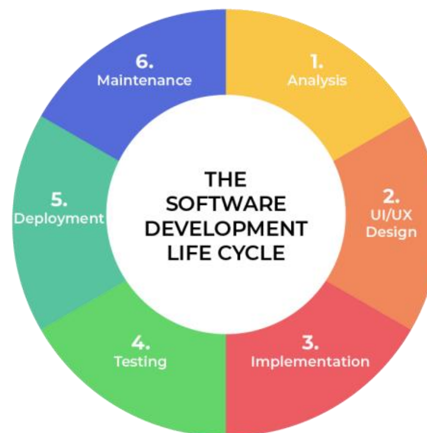
Teoretická část práce je založena na studiu odborné literatury týkající se způsobů automatizovaného testování softwaru a možností měřitelnosti, vyhodnocování kvality softwaru. Poznatky z analýzy budou sloužit jako výchozí bod pro praktickou část.

V rámci praktické části bude vypracován a ověřen business-case pro vytvoření a údržbu automatických testů ve střední firmě zabývající se vývojem softwaru, implementace sady testů na základě navrženého modelu a nastavení možných faktorů pro měřitelné sledování vývoje kvality softwaru. Na základě výsledků testování bude stanovena vhodnost zvoleného postupu a formulovány závěry práce.

3 Teoretická východiska

3.1 Testování softwaru a základní pojmy

Testování softwaru je jednou z klíčových fází cyklu vývoje softwaru. Hlavním cílem tohoto procesu je ověřování očekávaného chování a získávání informací o stavu softwarového produktu.



Obrázek 1 – cyklus vývoje softwaru rozdělený do jednotlivých fází [6]

V praxi velice často dochází k záměně některých základních pojmů. V rámci této kapitoly bych rád popsal rozdíl mezi třemi základními pojmy pro oblast testování a to chybu, defekt a selhání. Chybu způsobí přímo člověk, který se podílí na určité části vývojového procesu. Může to být vývojář, který při implementaci nedohlédne nějaký konkrétní scénář. Následkem této chyby je potom defekt, což už je přímo definovatelný problém, který není v souladu s očekávaným chováním dané funkcionality. Pokud se uživatel pohybuje v oblasti, kde očekává určité chování a systém zde má nalezený defekt, může jako jeho projev nastat selhání. [1]

Ideálním stavem je samozřejmě během procesu testování odhalit všechny defekty, které se v daném produktu mohou vyskytovat. Dnes již skoro žádný software nelze považovat za bezchybný. Aplikace jsou velice komplexní a zkrátka dohlédnout všechny možné scénáře, a ještě je všechny vhodným způsobem otestovat, není většinou možné. Vždy je snahou se tomuto ideálu co nejvíce přiblížit a vytěžit tak co nejvíce informací o stavu daného produktu. Informace mohou zároveň sloužit například k posuzování kvality softwaru, protože defekty softwaru jednoznačně kvalitu ovlivňují.

3.2 Automatizace testování

Automatizace testování softwaru se postupně stává stále významnější částí procesu vývoje softwaru. Jedním z důvodů, proč se automatizace testování začíná stále více využívat je, že začínají vznikat velice komplexní systémy, které kvůli jejich rozsahu je velice náročné testovat manuálně a bez automatických testů je zcela nemožné efektivně udržovat software v dostatečné kvalitě.

3.2.1 Testovací prostředí

Testovací prostředí je obecně dle ISTQB definováno jako prostředí obsahující hardware, software, simulátory, nástroje a další vybavení potřebné pro vykonávání testů. [1] Jedná se tedy o základ, nad kterým celý systém automatických testů běží, z čehož zároveň vyplývá, že je to i nutným předpokladem pro veškeré další úvahy implementace systému automatických testů. Je velice důležité hned v prvotní fázi návrhu architektury automatizovaného testování zohlednit, co všechno k následnému běhu testů bude potřeba a zvážit veškerá potencionálně možná rizika, která následně mohou běh testů ohrožovat. Testovací prostředí se může výrazně lišit v závislosti na typu vyvíjeného softwaru. Je třeba si uvědomit, že testovací prostředí by mělo být plně odstíněné od běžného vývoje a mělo by být co možná nejvíce izolované. Izolované z toho pohledu, že například k počítači, který chceme využít ke spouštění automatických testů, nebudou mít přístup osoby, které nemají se správou automatických testů nic společného.

Proč je tato izolace důležitá? Hlavním důvodem je zajištění stability testovacího prostředí. Cílovým stavem je minimalizovat dopady nestability testovacího prostředí na výsledky automatických testů. Jenom nečekaný výpadek proudu v době, kdy nemáme osobní přístup k testovacímu prostředí nám může zkomplikovat práci a ovlivnit výsledky automatických testů.

3.2.2 Strategie automatického testování

Zvolení vhodné strategie testování je dalším z klíčových prvků pro výslednou efektivitu testovacího procesu, a to ať už se jedná o manuální nebo automatické testování. Strategie testování by měla vycházet z funkčních a nefunkčních požadavků na danou funkcionalitu, případně na celý produkt. Pro otestování stanovených požadavků je tedy vhodné specifikovat v rámci procesu plánování akceptační kritéria pro nové funkčnosti.

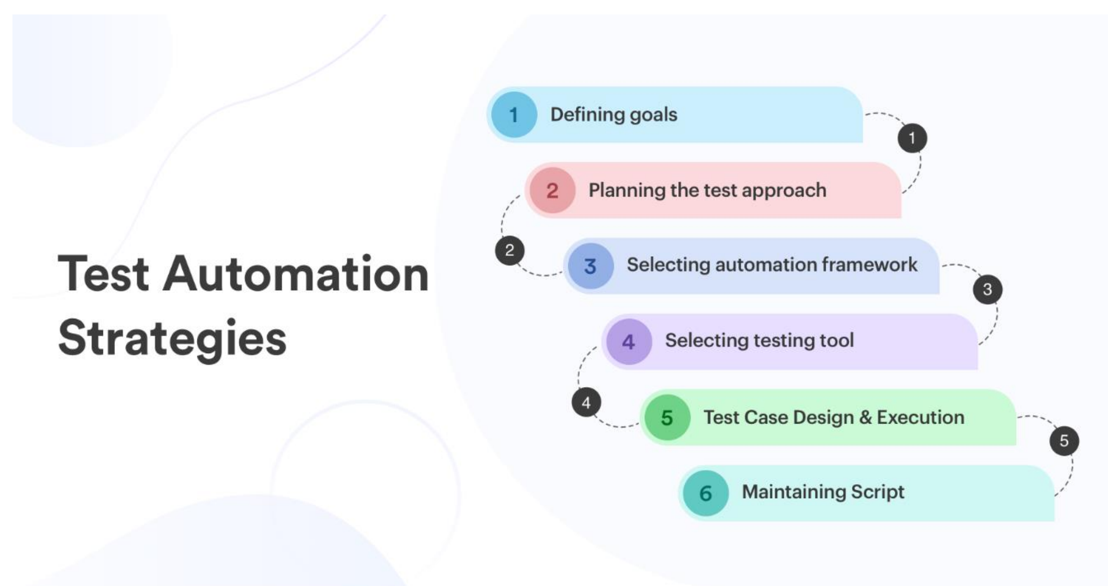
Funkční požadavky řeší konkrétní problémy, kde je výsledkem určitý výstup (např. konfigurace vstupů a výstupů na zařízení ovládaném skrze softwarovou aplikaci). Nefunkční

požadavky potom určují, jakým způsobem toho bude dosaženo, popřípadě jaké důsledky se mohou projevit. Typicky se jedná třeba o rychlost, přístupnost nebo zabezpečení. Nefunkční požadavky dále můžeme rozdělit do dvou základních kategorií. Do první kategorie můžeme zahrnout takové nefunkční požadavky, které mají dopady na koncového uživatele a do druhé naopak nefunkční požadavky neovlivňující uživatele. Mezi základní nefunkční požadavky, které ovlivňují uživatele, patří například rychlost, stabilita, design (UI/UX). Mezi nefunkční požadavky neovlivňující uživatele můžeme zahrnout například škálovatelnost softwaru nebo jeho udržovatelnost, což je činnost, kterou se primárně zabývají IT architekti.

Poté, co máme jasně specifikované, na co se při testování primárně soustředit, měli bychom se zaměřit na výběr vhodného typu automatických testů a případně nejlépe odpovídajícího testovacího frameworku.

Dále můžeme přistoupit k definici jednotlivých testovacích scénářů, které budeme chtít automatizovat. Zde je nejdůležitější zaměřit se na to, které testovací scénáře se nám vyplatí nejvíce automatizovat. Jak to ale správně vyhodnotit? Obecně si automatickými testy chceme usnadnit práci, takže kritériem číslo jedna může být počet opakování daného testovacího scénáře (jak často je nutné manuálně scénář procházet). Čím vyšší bude počet opakování daného scénáře, tím více času lze po automatizaci ušetřit. Dalším důležitým prvkem je u každého scénáře, který plánujeme automatizovat zvážit, zdali to z nějakého důvodu není příliš náročné, respektive vyhodnotit jeho návratnost. Pokud bychom došli k závěru, že návratnost automatizace daného scénáře bude až například po uzavření projektu, nemusí se nám vždy automatizace vyplatit, i přestože je nutné ho častokrát manuálně provádět.

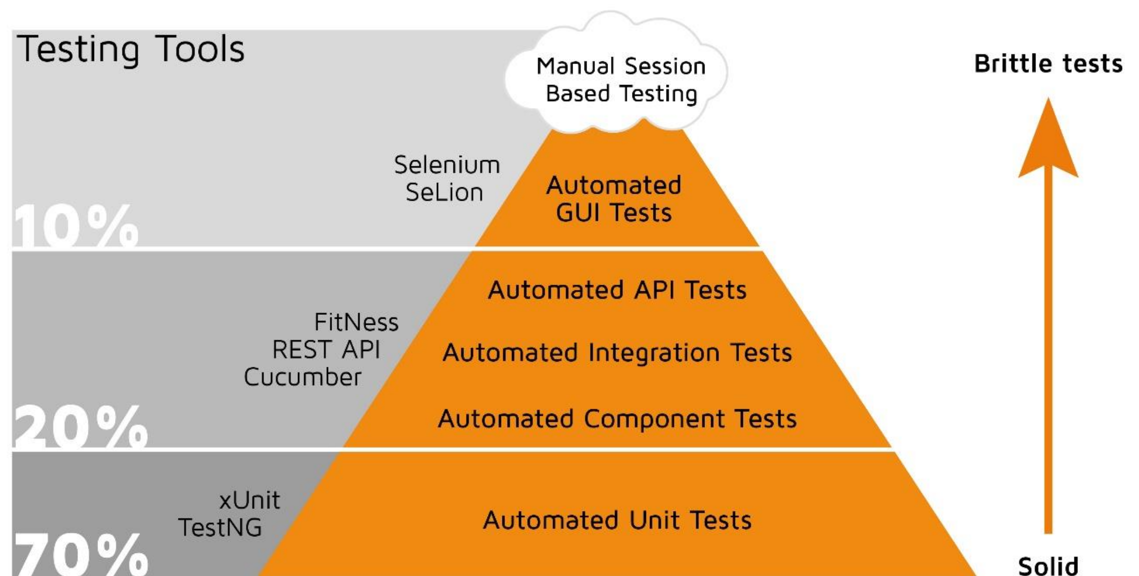
Posledním krokem v tomto celém cyklu je samotné spuštění definovaných testů.



Obrázek 2 – jednotlivé kroky při plánování strategie [7]

3.2.3 Návrh architektury

Jak tedy využít automatické testování, aby pro nás mělo co možná největší přínos? Důležitým výchozím bodem pro návrh vhodné architektury je uvědomit si, pro jaký typ softwaru chceme automatické testy nasadit, co vlastně bude testováno, jak testy budeme vyhodnocovat a na základě této úvahy vhodně navrhnout architekturu celého systému automatických testů. Od navržené architektury se poté odvíjí veškeré výhody, popřípadě nevýhody spojené s automatickým testováním a jeho údržbou. Určitě bude velký rozdíl v návrhu koncepce automatických testů pro software například v leteckém průmyslu a pro informační systém používaný v menší společnosti. Budou se pravděpodobně velice lišit požadavky jak na samotné testy, tak na testovací prostředí. Podcenění těchto úvah může vést k tomu, že se celý systém po určité době stane spíše zátěží než benefitem, což je samozřejmě neakceptovatelné a je potřeba se na to v počátcích zaměřit a dobře promyslet strategii testování.



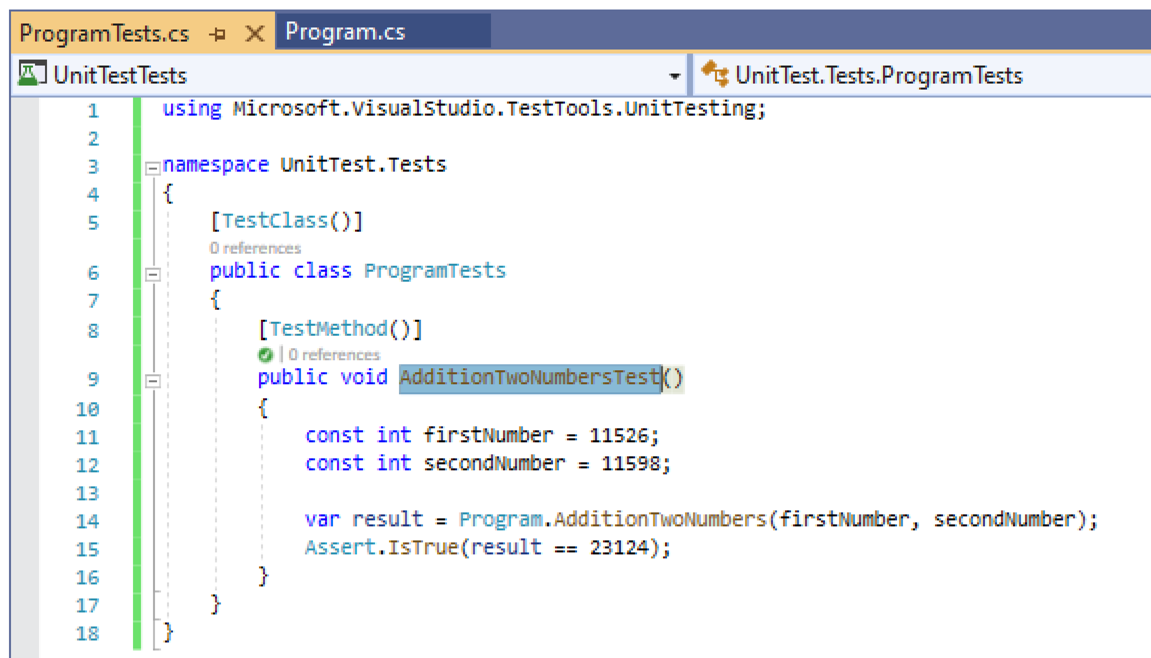
Obrázek 3 – pyramida struktury automatických testů [8]

3.3 Typy automatických testů

3.3.1 Jednotkové testy

Asi nejznámějším a nejvíce rozšířeným typem automatických testů jsou takzvané jednotkové testy (známé hlavně pod pojmem Unit Testy). Jednotkové testy validují přímo chování kódu tím, že očekávají určité výstupy na základě definovaných vstupních dat. Testy jsou psané ve stejném programovacím jazyce jako daná aplikace, která je testována. Konkrétním příkladem může být implementace testů pro jednoduchou kalkulačku, která bude

umět čtyři základní matematické operace – sčítání, odčítání, násobení a dělení. V tomto případě pro nás představuje každá matematická operace jednu konkrétní funkčnost, takže bude vhodné implementovat ke každé matematické operaci jeden jednotkový test, čímž vlastně zajistíme pokrytí všech hlavních funkčností naší aplikace. Pro ilustraci mohou být vstupní data reprezentována polem číslic, kde první dvě číslice budou argumenty a třetí číslice bude očekávaný výsledek. Samotný test následně zavolá metodu např. součet a předhodí této metodě první dvě číslice. Po provedení algoritmu dojde k validaci výstupu vůči třetí z číslic. Toto je celý princip jednotkových testů.



Obrázek 4 – implementace jednoduchého jednotkového testu

Hlavní výhodou jednotkových testů je to, že dávají vývojářům okamžitou zpětnou vazbu na jimi provedené úpravy kódu. Pokud tedy jako vývojář udělám chybnou úpravu v rámci funkčnosti sčítání, tak se ihned po spuštění testů dozvídám, že mé úpravy zanesly do softwaru chybu. Toto se nám může náramně hodit v komplexnějších aplikacích, kde může být i pro vývojáře velice obtížné dohlédnout všechny možné důsledky provedených úprav. Dalším benefitem je z pohledu nákladů i to, že pokud se o chybě dozvím ihned, tak se jako vývojář většinou orientuji v řešené problematice, a je tak pro mě jednodušší a hlavně rychlejší oprava, než když by se někdo k chybě vracel po měsících vývoje. Většinou jednotkové testy bývají do celého konceptu zakomponované tak, že vývojáři není umožněno aplikovat jeho změny kódu do té doby, dokud neprochází všechny jednotkové testy. Velkou výhodou může být také možnost provádět testy, aniž by k aplikaci existovalo uživatelské rozhraní.

3.3.2 Testy uživatelského rozhraní

Testy uživatelského rozhraní jsou navržené tak, aby simulovaly určitý průchod uživatele aplikací přímo prostřednictvím uživatelského rozhraní. Jako příklad můžeme využít horní menu aplikace Microsoft Word, kde jsou k dispozici různá tlačítka reprezentující určité funkčnosti. Dostupnost těchto tlačítek je řízena dynamicky, tudíž v kódu aplikace musí být definováno, za jakých podmínek je dané tlačítko aktivní, a kdy aktivní není. Toto už je funkčnost, kterou můžeme testovat na úrovni testování uživatelského rozhraní. Test může být koncipován tak, že se spustí daná aplikace a ve spolupráci s určitým rozhraním, které nám bude předávat informace o stavu jednotlivých tlačítek, můžeme ověřovat očekávaný stav vůči předpokladům. Opět máme tímto testem kompletně pokrytou funkčnost dynamického zobrazování stavu tlačítek a nemůže tak dojít k tomu, že se chyba neodhalí. Obecným shrnutím je, že tento typ testů umožňuje automaticky ověřovat různé prvky uživatelského rozhraní, jejich vlastnosti, rozložení a třeba i funkce.

Existují dva přístupy ke tvorbě těchto testů. Jednodušším z přístupů je způsob nahrávání testů. Uvedený způsob vytváření automatických testů je založený na nástroji, který nám umožní zaznamenat průchod určitým uživatelským scénářem, tuto nahrávku uložit, a poté jí opakovaně spouštět. Zde může být velkou výhodou v rámci přístupu, že testy zvládne obsluhovat v podstatě kdokoliv, kdo má zkušenosti s daným nástrojem pro tvorbu těchto nahrávek.

Druhým přístupem ke tvorbě automatických testů je deskriptivní programování. Tvorba tohoto typu testů je většinou náročnější, protože už pro vývoj potřebujeme specializovaného vývojáře. Vývojáři vytváří skripty v programovacích jazycích, které se následně spouštějí. Velkou výhodou jsou zde rozsáhlejší možnosti testování. Při zvolení správného frameworku lze ověřovat chování i složitějších uživatelských prvků a můžeme zde jít do detailů daného prvku. Příkladem může být testování funkcí komplexnější tabulky, kde lze například testovat funkčnosti jako různé seskupování, filtrování atd.

Vzhledem k tomu, že testy uživatelského rozhraní přímo simulují průchod uživatele nějakým scénářem, můžeme je využít i například k testování nefunkčních požadavků, což může být velice užitečné. Jaké nefunkční požadavky tedy můžeme reálně testovat? Můžeme se zaměřit na testování rychlosti aplikace nebo na paměťovou náročnost. Realizace těchto testů samozřejmě není tak jednoduchá jako například u jednotkových testů, ale může zde být velice rychlá návratnost. Pokud budeme vyvíjet aplikace, kde jsou například vysoké nároky na rychlost daného softwaru a měli bychom tyto nároky pravidelně manuálně testovat, bude to

velice nákladná záležitost, která se navíc může stát pro testery v rámci několika opakování noční můrou. Testování nefunkčních požadavků, konkrétně třeba rychlosti lze vysvětlit na následujícím příkladě. Představme si aplikaci, která obsahuje desítky tlačítek otevírající dialogová okna, ve kterých se načítají velké objemy dat. Otevírání dialogových oken tak může být poměrně náročnou operací z hlediska času. Můžeme tak vytvořit testy uživatelského rozhraní, které budou tato jednotlivá okna postupně otevírat a měřit, za jak dlouho jsou všechna data v okně načtená. Informaci opět obdržíme prostřednictvím frameworku, který s daným uživatelským rozhraním umí pracovat. Tyto informace můžeme v závěru běhu testu zpracovat a vytvořit z dat například grafický výstup, který může tester jednou za čas analyzovat. Tento grafický výstup může být potom v periodách spouštění automatických testů aktualizován a získáváme tedy křivku signalizující čas otevírání dialogových oken v čase s průběhem vývoje dané aplikace. Podobným způsobem mohou být realizovány testy paměťové náročnosti s tím rozdílem, že měříme přímo využití systémových prostředků na konkrétním zařízení.

Testy uživatelského rozhraní mohou být však oproti jednotkovým testům náročnější z pohledu doby běhu. Mohou běžet i třeba několik desítek hodin. Opět je zde základem volit vhodnou strategii pro spouštění testů. V případě, že jsou testy časově náročnější, může být například zvolena strategie, že testy běží přes noc na nejnovější verzi softwaru, kdy neblokují vývojáře ani testery a každý další den má vývojář zpětnou vazbu na nově vznikající funkčnosti, a je tedy stále možné reagovat rychle na objevené chyby.

3.3.3 Integrační testy

Při vývoji komplexnějších aplikací se mnohdy setkáváme s moduly, ze kterých je celý softwarový produkt složený. Tyto moduly mají výhodu v tom, že mohou být vyvíjeny odděleně a je možné je potom využít i v jiných aplikacích jako komponenty zajišťující nějaké služby. Tento způsob vývoje aplikací je velice známý pod pojmem SOA (Service Oriented Architecture). Zde se setkáváme s novou problematikou, kterou je třeba testováním pokrývat, a která lze automatizovat. Můžeme totiž provozovat nezávisle na sobě dva moduly, které budou fungovat bez problémů, ale při jejich vzájemném propojení už to tak být nemusí. Integrační testy se tedy zabývají testováním vzájemně propojených nezávislých celků. Co se týče automatizace, jedná se o asi nejnáročnější disciplínu. Důvodem může být využití různých programovacích jazyků nebo technologií v rámci jednotlivých modulů. Integrační testy mohou být hojně využité například v oblasti softwaru pracujícím s embedded zařízeními. Software běžící na počítači může konfigurovat hardwarové zařízení, kde se potom může

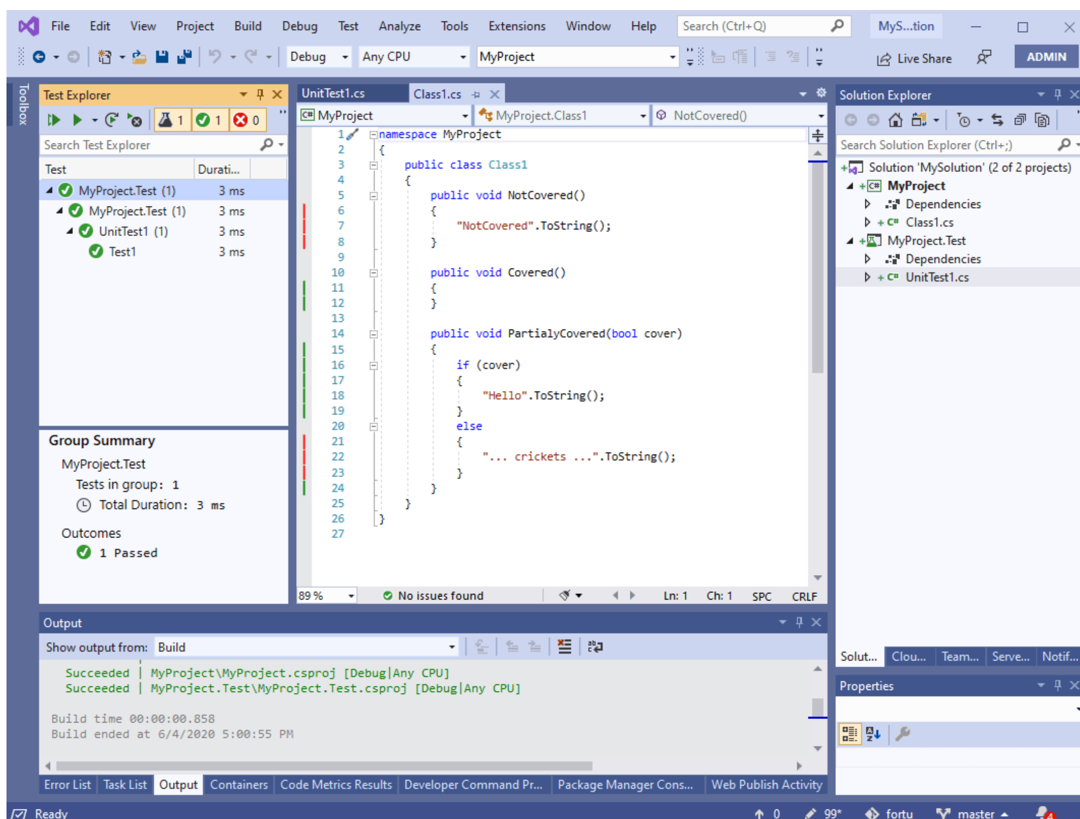
testovat až výstup z daného hardwarového zařízení (toto zařízení samozřejmě také obsahuje určitý software). V oblasti rozhraní mezi moduly bývá odhalováno paradoxně velké procento defektů.

3.4 Pokrytí kódu jednotkovými testy

Pokrytí kódu jednotkovými testy známé mnohdy pod pojmem code-coverage vyjadřuje poměrově pokrytí funkčního kódu jednotkovými testy. Čím větší je pokrytí kódu jednotkovými testy, tím menší je samozřejmě pravděpodobnost zavlečení defektu do softwaru. Automaticky lze očekávat, že čím větší bude pokrytí kódu jednotkovými testy, tím větší bude náročnost na údržbu. Code-coverage je zcela jistě dalším faktorem, který do jisté míry ovlivňuje kvalitu softwaru. Cílem analýzy pokrytí kódu je odhalit oblasti, které nejsou pokryté žádnými automatickými testy a zároveň detekovat oblasti, kde mohou naopak existovat duplicity. [3]

U této problematiky je třeba zvážit, do jaké hloubky je žádoucí mít kód pokrytý. Základní možností je pokrytí jednotlivých řádků a příkazů v kódu. Toto základní pokrytí už však nepočítá například s podmínkami nebo cykly, kde zůstávají nepokryté některé části kódu, který je vykonán jen v určitých případech. Pro tuto analýzu existují dnes již moderní nástroje, které jako výstup umí přesně zpracovat, které funkční části kódu jsou již aktuálně pokryté a které nikoliv. Nástroj je schopný následně určit procentuální míru pokrytí kódu a třeba i umožní detailní průchod skrze kód aplikace s tím, že od sebe graficky oddělí pokryté části od nepokrytých. [3]

Pro ilustraci je zde vybrán jeden doplněk, který je dostupný přímo pro vývojové prostředí Microsoft Visual Studio. Na obrázku níže je vidět, že tento doplněk vedle čísel jednotlivých řádků graficky zobrazuje, které části kódu pokryté jsou, a které ne.



Obrázek 5 – vizualizace pokrytí kódu [11]

3.5 Efektivní využití automatických testů

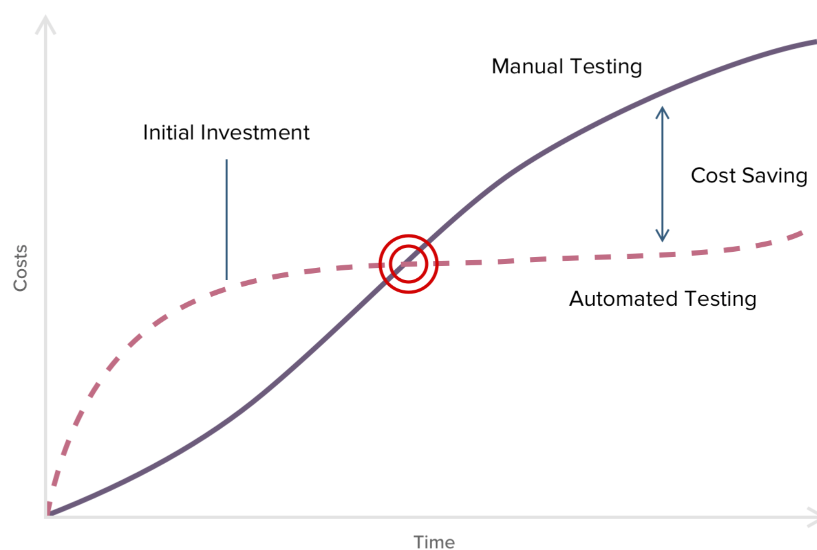
Efektivita procesu využití automatických testů se odvíjí primárně od vhodně navržené testovací strategie, architektury testů až po vhodné návrhy jednotlivých testů. Úspěšnost závisí na kombinaci základních faktorů a nelze tedy tvrdit, že například výborně propracovaným návrhem strategie dosáhneme efektivního procesu celého automatického testování.

Existuje mnoho způsobů a úhlů pohledu na tuto problematiku. Představme si zde některé základní. Určitě na efektivitu tohoto procesu lze nahlížet z pohledu financí. Můžeme říct, že aby se automatizace vyplatila, musí se investice do vývoje a údržby automatických testů vždy vrátit. To lze poměrně jednoduše vyčíslit. Můžeme vzít v potaz odhadované náklady na vývoj a údržbu a porovnat je s náklady spojenými s manuálním testováním stejné oblasti. Pokud zjistíme, že jsou náklady na manuální testování vyšší už zde získáváme čas, který tester může využít na nějakou jinou činnost, která může být pro danou společnost přínosem. Poté samozřejmě čím vyšší a rychlejší bude návratnost investice, tím lépe lze proces obhájit a lze ho i z pohledu financí považovat za efektivní.

3.6 Náklady spojené s automatizací

3.6.1 Vývoj testů

Vývoj automatických testů bohužel není v projektech zanedbatelnou položkou, co se týče nákladů. Náklady se budou lišit v závislosti na typu automatických testů a jejich rozsahu. Pokud se vydáme cestou automatického testování softwaru je samozřejmě nutné počítat s vyššími počátečními náklady. Tyto náklady v počátcích převyšují náklady na samotné manuální testování, ale do budoucna se postupně snižují. Náklady na manuální testování mají přibližně lineární tvar (viz obrázek č. 6).



Obrázek 6 – křivka nákladů pro manuální i automatické testování [9]

Na obrázku je velice důležitý průnik křivek automatizovaného testování a manuálního testování. Z pohledu nákladů je vždy naším cílem tohoto průniku dosáhnout co možná nejrychlejším možným způsobem. Z křivky vyznačené pro automatizované testování také vidíme, že nejrazantněji stoupá v počátku, kdy probíhá vývoj a poté už je spíše stabilizovaná.

3.6.2 Údržba testů

Údržba automatických testů může být kvůli jejím nákladům jedním z důvodů, proč se například od automatizace testování může ustoupit, protože dlouhodobě představuje většinu nákladů, které jsou s automatizací spojeny. Údržba je velice důležitý kontinuální proces, se kterým je třeba již v začátcích veškerých implementací počítat a cílem dobře promyšleného efektivního procesu automatizace testování je údržbu co možná nejvíce minimalizovat.

3.7 Nejčastější problémy spojené s automatizací

Nejčastější problémy vznikají většinou neočekávanými událostmi z oblasti prostředí a vývoje softwaru, pro který jsou automatické testy implementovány. V rámci prostředí se může jednat třeba o aktualizace softwaru, které nemusíme mít vždy pod manuální kontrolou (automatické aktualizace). Dalším velice častým problémem může být opomenutí automatické testy upravit při změně funkcionality. Tomu lze však v některých případech poměrně dobře předcházet, a to třeba při pokrytí unit testy. Problematičtější to však může být v rámci testů uživatelského rozhraní nebo integračních testů, kde běh testů bývá částečně oddělený od přímého vývoje.

3.8 Kvalita softwaru

3.8.1 Předpoklady pro kvalitní software

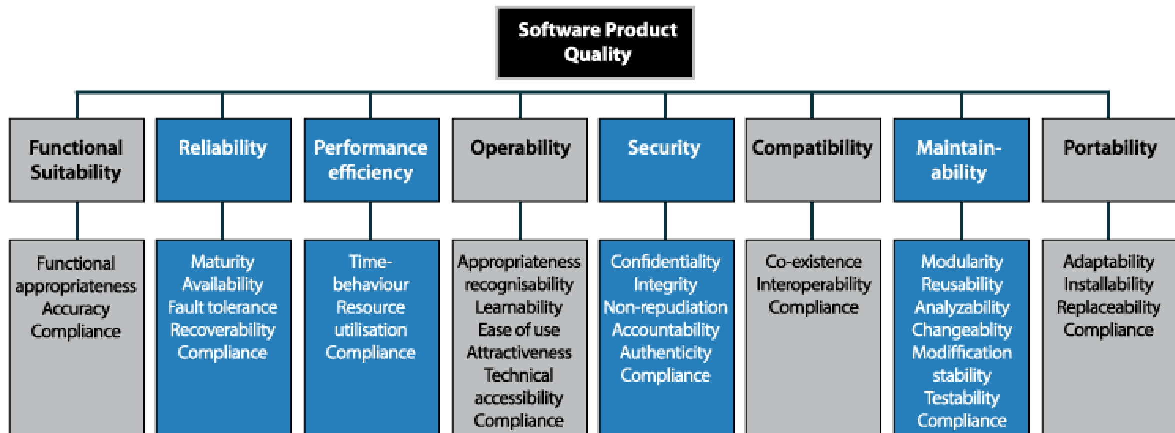
Základním předpokladem pro kvalitní software je v případě vývoje softwaru na zakázku dobře odhadnutá smlouva o dodání. Nejčastějším problémem bývá při návrhu smlouvy podcenění nákladů na vývoj. S tímto problémem se však můžeme setkat i při vyvíjení softwaru pro interní potřeby, kde sice společnost nemusí být až tak striktně limitována termínem dodání, ale z hlediska plánování dalších činností také není vhodné nabírat zpoždění. K problému dochází ve chvíli, kdy se proces vývoje začne zpožďovat vůči původním odhadům, a to hlavně z důvodu, že je snaha proces vývoje urychlit. Toto urychlování sebou ale samozřejmě nese velký vliv na kvalitu. Konkrétním důsledkem může být právě třeba to, že se rozhodne ušetřit na implementaci automatických testů, které by v budoucnosti mohly odhalit určité množství zanesených chyb.

Je tedy důležité v rámci odhadů pro implementaci zohlednit veškerá rizika, která sebou vývoj nových funkčností může nést a tyto rizika do odhadů promítnout. Odhady tedy mohou v podstatě umožnit vytvoření prostoru na zohlednění veškerých aspektů kvality softwaru. Pokud tento prostor nevznikne už v rámci odhadů, může následně dojít k zanedbání některých z těchto aspektů na úkor právě původních odhadů, pokud bude nutné dodržet termín již akceptované smlouvy pro dodání softwaru.

3.8.2 Modely kvality

Model kvality softwarového produktu lze definovat jako souhrn charakteristik a jejich vzájemných vztahů, které poskytují rámec pro specifikaci požadavků na kvalitu a její hodnocení. [3]

Každá norma vztahující se ke kvalitě softwaru definuje trochu jiný soubor aspektů, dle kterých může být kvalita posuzována. Na obrázku níže je vidět v první úrovni základní souhrn jednotlivých aspektů a ve druhé úrovni vlastnosti, které dané aspekty reprezentují.



Obrázek 7 – definice aspektů kvality softwaru [10]

Z historie jsou známé tyto modely kvality softwaru: [3]

1. McCallův model
2. Boehmův model
3. Dromeyho model
4. Model FURPS

McCallův model kvality softwaru je převážně zaměřený na proces vývoje. Snaží se najít co možná nejvýhodnější průnik mezi požadavky uživatelů a potřeby vývojářů. Tento model nahlíží na kvalitu ze tří hlavních pohledů: [3]

1. Revize produktu – proveditelnost změn
    a. Flexibilita
    b. Udržovatelnost
    c. Testovatelnost
2. Činnost produktu – provozní charakteristiky
    a. Správnost
    b. Spolehlivost
    c. Účinnost
    d. Integrita
    e. Použitelnost
3. Přechod produktu – schopnost přizpůsobení se novému prostředí
    a. Přenositelnost
    b. Znovupoužitelnost

c. Interoperabilita

Dalším historicky známým modelem je Boehmův model, který na rozdíl od McCallova modelu staví požadavky koncových uživatelů jako nejdůležitější a nebere v potaz v takové míře technické detaily. Boehm jakožto autor tohoto modelu se však nezabýval tím, jak by stanovené charakteristiky kvality mohly být měřeny. Stejně tak jako předchozí model je tento také rozdělen do tří úrovní: [3]

1. Udržovatelnost
    a. Srozumitelnost
    b. Modifikovatelnost
    c. Testovatelnost
2. Vlastnosti při užití
    a. Spolehlivost
    b. Účinnost
    c. Použitelnost
3. Přenositelnost

Dromeyho model nahlíží na kvalitu hlavně z pohledu produktu. Dromey hodnotil kvalitu softwaru na základě jeho komponent, kde komponentami mohly být proměnné, příkazy, funkce, požadavky, moduly a podobně. Model následně vycházel z toho, že tyto vnitřní charakteristiky kvality přímo určují vnější atributy kvality daného produktu. Dromeyho rámec kvality se skládal ze tří modelů: [3]

1. Model kvality implementace
2. Model kvality požadavků
3. Model kvality návrhu

Model FURPS je zajímavý hlavně z toho důvodu, že to byl první model založený na zájmu odvětví vývoje softwaru. Konkrétně to byl model představený ve společnosti Hewlet-Packard. Jeho použití bylo širší a zahrnovalo v sobě i třeba různé kategorizace testovacích případů nebo uživatelských scénářů. Model FURPS popisuje tyto základní faktory kvality: [3]

1. Funkčnost
2. Použitelnost
3. Spolehlivost
4. Výkonnost
5. Podporovatelnost

3.8.3 Normy pro oblast kvality softwaru

Pro oblast kvality softwaru je definována celá řada norem. Problematický byl vývoj těchto norem, který byl hodně roztříštěný a měl celou řadu nedostatků. Některé normy byly vůči sobě nekonzistentní, a bylo nutné tento problém řešit. Začalo se tak dít v rámci projektu SQuaRE, který měl odstranit nekonzistence mezi některými normami a zároveň odstranit jejich nedostatky. [3]

Seznam norem týkajících se kvality softwaru:

1. ISO/IEC 9126
2. ISO/IEC 14598
3. ISO/IEC 12119
4. ISO/IEC 12207
5. ISO/IEC 15504
6. ISO/IEC 15939
7. ISO/IEC 25000-25099 (SQuaRE)

Většina těchto norem byla později přebírána i v České republice.

3.8.4 Kvalita kódu dle SOLID

SOLID je jeden z principů používaných v oblasti návrhových vzorů pro vývoj softwaru v objektově orientovaném programování. Obecně můžeme návrhový vzor chápat jako šablonu pro řešení častých obecně známých problémů. Jejich cílem je předcházet vzniku nevhodných návrhů struktury kódu. Nevhodný návrh potom může mít vlastnosti jako křehkost, rigidita, imobilita, viskozita nebo zbytečná složitost.

Křehkost softwaru se vyznačuje tím, že změny kódu mohou často způsobovat těžko dohlédnutelné nežádoucí efekty (zanesení defektů). Pokud software projevuje známky křehkosti je s ním samozřejmě spojena náročná údržba. [3]

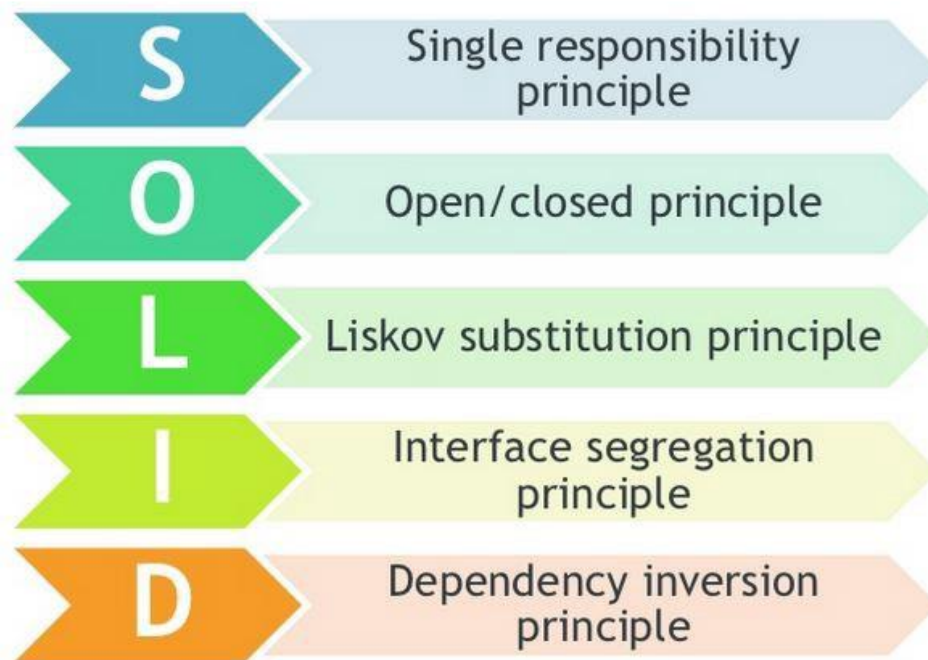
Další důležitou vlastností je rigidita. Rigidita se vyznačuje problematicky modifikovatelným kódem, i přestože se nemusí jednat o žádné velké úpravy. Následky rigidity mohou potom být například časté podceňovaní náročnosti úprav, což se může projevovat zejména u nových vývojářů. [3]

Imobilita má za následek duplikování kódu, a to z důvodu, že kód není možné jednoduše sdílet přes různé oblasti, kde by to dávalo smysl a je jednodušší napsat stejný kód znovu. [3]

Viskozita vyjadřuje, do jaké míry je rozdíl v náročnosti implementace mezi správným architektonickým řešením anebo pouhým obcházením správného návrhu. Viskozita je vysoká,

je-li obcházení správného návrhu výrazně snazší než implementace správného architektonického návrhu. [3]

Akronym SOLID má v každém svém písmenu skryté jedno doporučení pro určitou oblast (viz obrázek níže).



Obrázek 8 – SOLID akronym [12]

Princip jedné odpovědnosti vyznačující první písmeno „S“ doporučuje, aby každá třída měla pouze jednu odpovědnost, což by mělo zajistit nižší rigiditu a imobilitu kódu. [3]

Druhé písmeno „O“ reprezentuje princip otevřenosti a uzavřenosti. Tento princip poukazuje na to, že veškeré třídy, metody dané aplikace by měly být otevřeny pro rozšíření, ale zároveň uzavřeny pro změnu. Existuje zde jedna výjimka, a to opravy defektů, která bude znamenat právě změnu. Cílem je vyvarovat se vlastnostem rigidita, křehkost a viskozita. [3]

Další písmeno „L” takzvané Liskovové princip substituce je hodně blízký s předchozím principem otevřenosti a uzavřenosti. Tento princip souvisí s dědičností a konkrétně definuje, že potomci určitých objektů by měli být schopni kdykoliv zastoupit své předky.

Písmeno „I“ reprezentuje princip oddělení rozhraní. Tento princip je přímo definován: *„Žádný klient by neměl být nucen do závislosti na rozhraní, která nepoužívá“.* [3] Při návrhu je vhodné se zamyslet nad strukturou jednotlivých rozhraní a spíše se je snažit rozdělit do menších specifických celků než vytvářet rozsáhlá obecná rozhraní. [3]

Poslední písmeno „D“ řeší princip inverze závislostí. U tohoto principu je snahou odstraňovat závislosti na konkrétních třídách v kódu a nahrazovat je závislostmi na abstrakcích. [3]

3.8.5 Měřitelnost kvality softwaru

Co se týče kvality softwaru, bývá o ní často mluveno na velmi abstraktní úrovni a je někdy obtížné ji jednoduše definovat. Hlavním problémem zde může být i fakt, že některé z aspektů kvality softwaru jsou takřka neměřitelné.

Je vhodné se tak zaměřit hlavně na vlastnosti, které jsou jednoduše měřitelné. Může to být třeba spolehlivost, kterou můžeme měřit podle počtu defektů v daném softwaru.

Dalším měřitelným aspektem může být výkonnost. Výkonnost lze testovat různými zátěžovými testy, kde můžeme sledovat časy odezvy během definovaných operací. Tyto testy zároveň připadá v úvahu automatizovat.

Udržovatelnost může být také jedním z měřitelných aspektů kvality softwaru. Můžeme ji měřit na základě řádek kódu aplikace nebo třeba dle počtu implementovaných tříd. Při běžícím vývoji může být zajímavé sledovat vývoj počtu přibývajících řádků v aplikaci a například korelaci s počtem zanášených defektů.

Testovatelnost lze také zcela jistě měřit podle toho, kolik technologií je potřeba k testování využít, jak kvalitní dokumentace dané funkčnosti je k dispozici nebo například jak dlouho může trvat objasnění testovaného problému někomu, kdo je v dané problematice nový a nemá s ní tolik zkušeností.

4 Vlastní práce

4.1 Výběr strategie

4.1.1 Koncepce testů

Software, pro který jsou automatické testy koncipovány, je určen pro konfiguraci embedded zařízení, a proto jsem se rozhodl hlavně z důvodu možností pokrytí více oblastí rozšířit sadu integračních testů. Tyto integrační testy jsou schopné simulovat proces konfigurace daného zařízení skrze konfigurační software a blíží se tak nejvíce k reálnému scénáři využití v praxi. Testy tedy pracují skrze konfigurační software s připojenými fyzickými zařízeními a jsou složené ze dvou částí. První částí je testování uživatelského rozhraní pouze v rámci konfiguračního softwaru, což lze i následně oddělit a využívat samostatně. Druhou částí je možnost pracovat s reálným fyzickým zařízením, kde se z testů stávají testy integrační. Nejenom, že jsou testy schopné ověřit správné chování prvků uživatelského rozhraní, ale umožňují i například ověření správného zápisu konfigurace na základě vložených dat prostřednictvím uživatelského rozhraní.

4.1.2 Požadavky

Strategie testovacího procesu byla stanovena na základě rychlosti návratnosti investice, kde byla požadována ze strany firmy návratnost do jednoho roku. Na základě tohoto požadavku jsem zpracoval plán pro implementaci sady testů. Cílem bylo pokrýt co největší množství funkcionalit, což vedlo primárně k úvaze, kolik automatických testů půjde pro nové zařízení jednoduše s drobnými úpravami využít. Provedl jsem analýzu, jejíž výsledkem bylo možné využití již implementovaných 35 testů uživatelského rozhraní. Prvotním cílem bylo implementovat 10 nových testů uživatelského rozhraní na takové obecné úrovni, aby šly testy následně využít i pro další zařízení, která jsou daným softwarem podporovaná, ale bohužel by potom nevycházela návratnost investice do jednoho roku. Zvolil jsem tedy počet nových testů 5. Celkem tak bude nové zařízení pokrývat 40 automatických testů uživatelského rozhraní.

4.2 Business-case pro automatické testy

Předpokladem je nasazení 40 automatických testů uživatelského rozhraní, z nichž 35 testů bude pouze vhodně rozšířeno pro podporu nového zařízení a 5 testů bude nově vytvořených tak, aby bylo možné je následně využít i třeba pro další nově vyvíjená zařízení.

Pro návratnost investice jsem počítal s tím, že primárně největší návratnost se pohybuje v oblasti regresních testů, které manuálně probíhají na daném projektu jednou za měsíc v rozsahu 15MD. Business-case je postavený spíše ze strategického pohledu ušetření nákladů v budoucím vývoji softwaru a není zde tedy počítáno s ovlivněním prodejů v oblasti trhu a s určitou cílovou skupinou uživatelů, pro které by tento projekt byl atraktivní natolik, že by ho po uvedení na trh pravděpodobně zakoupili.

Tabulka 1 – odhadované náklady na vývoj automatických testů

Úkol	Vývoj
Převzetí 35 testů včetně drobných úprav	2MD
Implementace 5 nových testů	10MD
Nastavení automatického spouštění testů	1MD
Celkem	**13MD**

Tabulka 2 – odhadované náklady na údržbu automatických testů

Úkol	Údržba/měsíc	Údržba/rok
Údržba 40 automatických testů	1.5MD	18MD
Údržba prostředí	0.25MD	3MD
Celkem	**1.75MD**	**21MD**

V tabulkách výše jsou vyčíslené všechny náklady, které jsou spojené s vývojem a údržbou automatických testů. Cílem by mělo být obhájit návratnost investice již po prvním roce, takže nyní vypočteme náklady na manuální testování (ve stejném rozsahu pokrytí) v případě absence těchto automatických testů.

Tabulka 3 – odhadované náklady na manuální testování

Úkol	Náklady/měsíc	Náklady/rok
Otestování oblasti v rozsahu, co pokrývají automatické testy	3MD	36MD
Správa testovacích scénářů	0.5MD	6MD
Celkem	**3.5MD**	**42MD**

Na základě splněných požadavků na návratnost investice bylo možné implementaci automatických testů schválit.

4.3 Stanovení hypotéz

1. Implementace automatických testů by měla být realizovatelná v mezích původních odhadů
2. Údržba testů a prostředí by měla být uskutečnitelná v mezích původních odhadů

4.4 Implementace testů

Implementace testů probíhala v rámci programovacího jazyka C# a frameworku TestStack.White, který je určený právě k testování uživatelského rozhraní. Testovací framework TestStack.White umožňuje jednoduše přes přidělené identifikátory přístup k jednotlivým prvkům uživatelského rozhraní. Je tedy nutné všem prvkům, se kterými je žádoucí v rámci automatických testů pracovat přidělit jejich identifikátor, který potom slouží k získání daného prvku (např. jednoduchý text-box) a umožňuje s ním pracovat. Lze tedy ověřit jeho vlastnosti, což může být třeba, jestli je prvek možné editovat nebo je z nějakého důvodu aktuálně editace omezená a zároveň lze do prvku vložit nějaký vstup. Jako vývojové prostředí jsem používal Microsoft Visual Studio.

Tabulka 4 – reálné náklady na implementaci

Úkol	Odpracovaný čas
Převzetí 35 testů včetně drobných úprav	1.25 MD
Implementace 5 nových testů	13.5 MD
Nastavení automatického spouštění testů	0.5 MD
Celkem	**15.25 MD**

4.5 Prostředí spouštění automatických testů

Prostředí pro běh automatických testů bylo přebráno z již hotového řešení a v rámci mé práce jsem prováděl analýzu tohoto prostředí a jeho problémových oblastí, na které jsem během vývoje a následné údržby automatických testů narážel.

K běhu testů je potřeba počítač, skrze který jsou automatické testy spouštěné. Na tomto počítači je instalován software, který na základě definovaného plánu spouští automaticky testy v určeném čase. Testy jsou spouštěné vždy na nejnovější vydané verzi softwaru v čase mimo běžnou pracovní dobu společnosti, typicky v nočních hodinách.

Hardwarová zařízení, jejichž konfiguraci zajišťuje testovaný software, jsou k danému počítači připojená prostřednictvím ethernetového rozhraní, případně pomocí USB.

4.5.1 Zásahy nepovolaných osob

První problém, se kterým jsem se setkal, byl přístup nepovolaných osob přímo na zařízení, jejichž konfiguraci software testuje, což mělo za následky časté padání testů na neočekávaných událostech. Typicky se jednalo o restart zařízení nebo o vyvolání takové události, která způsobila reakci v uživatelském rozhraní na testovacím počítači a už s ní nebylo v rámci automatického testu počítáno. Jako řešení tohoto problému jsem navrhl změnu parametrů zabezpečení spojení, protože byly použité výchozí, což byla jednoznačně chyba. Toto řešení bylo i později naimplementováno a vnější zásahy do těchto zařízení byly dostatečně odfiltrovány.

Druhým problémem byl fyzický zásah přímo do zařízení, na kterém byly testy spouštěné. Zde se typicky projevovala zvědavost lidí při běhu testů a nutkání se zařízením něco zkusit, což mělo za následky opět nahodilé padání některých testů. Byla tedy ze zařízení odpojená myš s klávesnicí pro omezení práce s testovací stanicí, což ale zatím nepokrývá všechny možné zásahy do zařízení.

4.5.2 Změny v operačním systému

Nejčastěji jsem se v rámci své práce setkával s problémy, které byly spojené přímo s operačním systémem na počítači, kde byly automatické testy spouštěné. Příčinou byla ve většině případů automatická aktualizace operačního systému. Každá změna operačního systému byla považována za rizikovou a snahou bylo dostat aktualizace Windows pod kontrolu a dělat je manuálně. Tento úkol vypadal na první pohled celkem jednoduše, ale v prostředí, které podléhá firemním IT politikám to už tak jednoduché nebylo. Aktualizace byla vynucená, i přestože se na daném zařízení v rámci nastavení vypnula. Po tomto zjištění již nebylo možné jednoduše změnit chování a zatím jsem se rozhodl dělat revizi po aktualizacích manuálně.

Další příčiny spojené s aktualizací operačního systému:

1. Aktualizace .NET frameworku
2. Změna nastavení asociace aplikací s určitými soubory

4.5.3 Vyskakování oken během průběhu testů

Tuto problematiku rozdělím hned v počátku do dvou částí. V první části jsem začal pozorovat problémy s vyskakovacími okny souvisejícími se spouštěním skriptů dané společnosti. Otevřené okno se po spuštění skriptu stalo oknem aktivním a znemožnilo tak například operaci kliknutí na příslušný prvek v uživatelském rozhraní. Tento problém jsem

ohlásil vývojářům a byl vyřešen spouštěním skriptů na pozadí, což opět do jisté míry stabilizovalo prostředí automatických testů.

Druhou část problémů způsobovaly notifikace v prostředí Windows. Tyto notifikace však bylo možné v nastavení operačního systému vypnout a problém byl tímto odstraněn.

4.6 Běh testů

Běžně byly testy spouštěné každý den v nočních hodinách na nejnovější verzi, pokud se nová verze softwaru v daný den vydávala. Pokud nastal nějaký problém v rámci testovacího prostředí a testy nebyly spuštěné, spustila se sada manuálně.

Běh jednoho opakování této sady 40 testů trval v průměru z 5 sledovaných běhů 2 hodiny a 25 minut (zaokrouhleně). Mnou realizovaný manuální průchod obdobnými scénáři byl časově mnohem náročnější. První průchod mi časově zabral 9 hodin při zohlednění přípravy testovacího prostředí a administrativnímu plnění všech náležitostí. Zde je možné si všimnout poměrně velké odchylky vůči prvotním odhadům na manuální testování, kde byl původní odhad na jeden průchod 24 hodin, což ale může být reálné, pokud by tyto scénáře měl procházet například nový zaměstnanec bez zkušeností v daných oblastech softwaru. Je zde tedy jednoznačně vidět časová úspora při prvním průchodu a to nezanedbatelných 6 hodin a 35 minut.

4.7 Údržba automatických testů

Údržbu 40 implementovaných automatických testů jsem sledoval po dobu 3 měsíců a zaznamenával jsem, kdy bylo nutné z důvodu úprav chování aplikace upravit i chování některých automatických testů. Testy jsem pro tuto práci zobecnil na formát Test X, kde X vyjadřuje číselnou identifikaci daného testu. V rámci tabulky v této kapitole je uvedený souhrn testů, pro které bylo nutné uskutečnit zásahy přímo do jejich kódu a je zde uvedený i orientační rozsah náročnosti úprav. Vyhodnocení vůči původním odhadům je popsané v kapitole s výsledky.

Tabulka 5 – náklady na modifikace testů v období 3 měsíců

Test	Náročnost úprav (MD)
Test 3	0.25 MD
Test 8	0.75 MD
Test 22	2MD (opravy se promítly do více testů)
Test 23	

Test 24	
Test 31	0.25 MD
Test 36 (nově implementovaný test)	0.5 MD
Celkem za 3 měsíce	**3.75 MD**

4.8 Sledování počtu defektů

Na základě možností, které jsem měl, jsem se rozhodl začít sledováním počtu defektů v softwaru a analýzou jejich vlastností. Z těchto měření by měla být následně stanovená metrika pro možné sledování vývoje kvality softwaru z pohledu počtu defektů. Defekty jsem v první fázi rozdělil do dvou kategorií. První kategorie obsahovala všechny aktuálně existující defekty a druhá kategorie obsahovala pouze již uvolněné defekty. Počty defektů jsem sledoval skrze softwarový nástroj pro evidenci dat souvisejících s defekty, kde jsem si vytvořil filtry, které mi umožnily k číselným údajům opakovaně přistupovat. Data z filtrů jsem stahoval vždy jednou týdně po dobu 6 měsíců na přelomu roku 2020/2021 a každému defektu byla přiřazována určitá bodová váha podle závažnosti defektu.

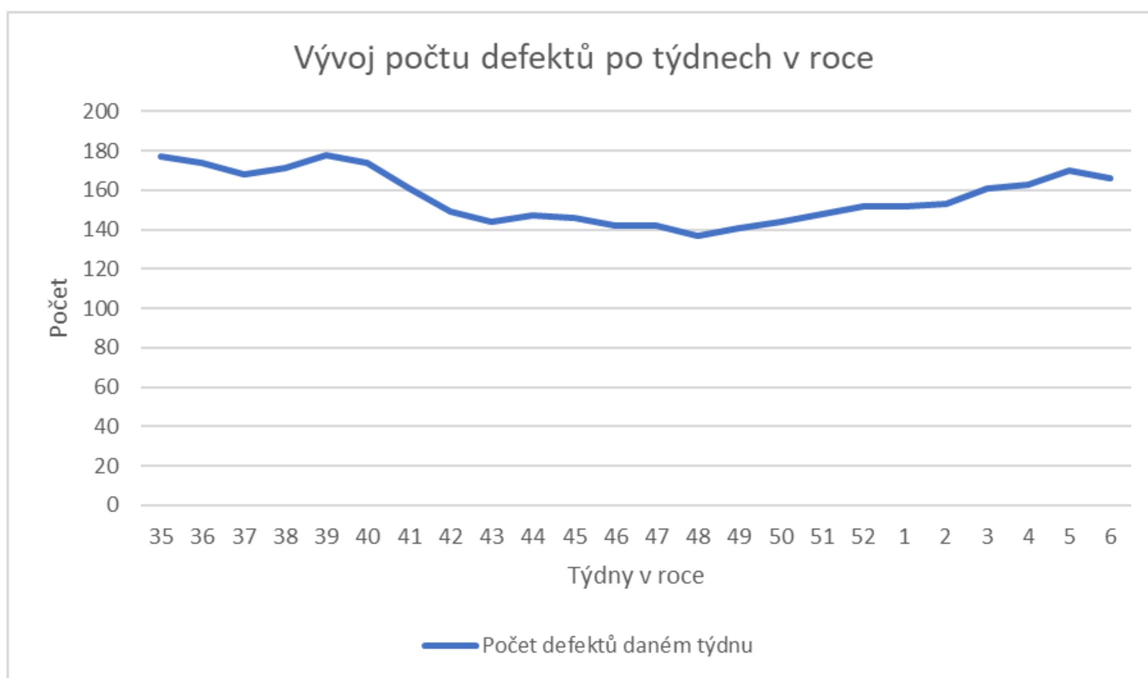
4.8.1 Analýza již evidovaných defektů v systému

Ještě před počátkem sledování defektů v čase proběhla analýza evidovaných defektů z pohledu jejich aktuálnosti a reprodukovatelnosti. Tato analýza byla dělaná za účelem zpřesnění dat, která byla později sledována, protože systém obsahoval i defekty, které byly již některými zásahy do kódu upraveny nebo se dále v softwaru nevyskytovaly.

Tato analýza nyní probíhá pravidelně po 6 měsících, ale v rámci sledování počtu defektů proběhla pouze jednou.

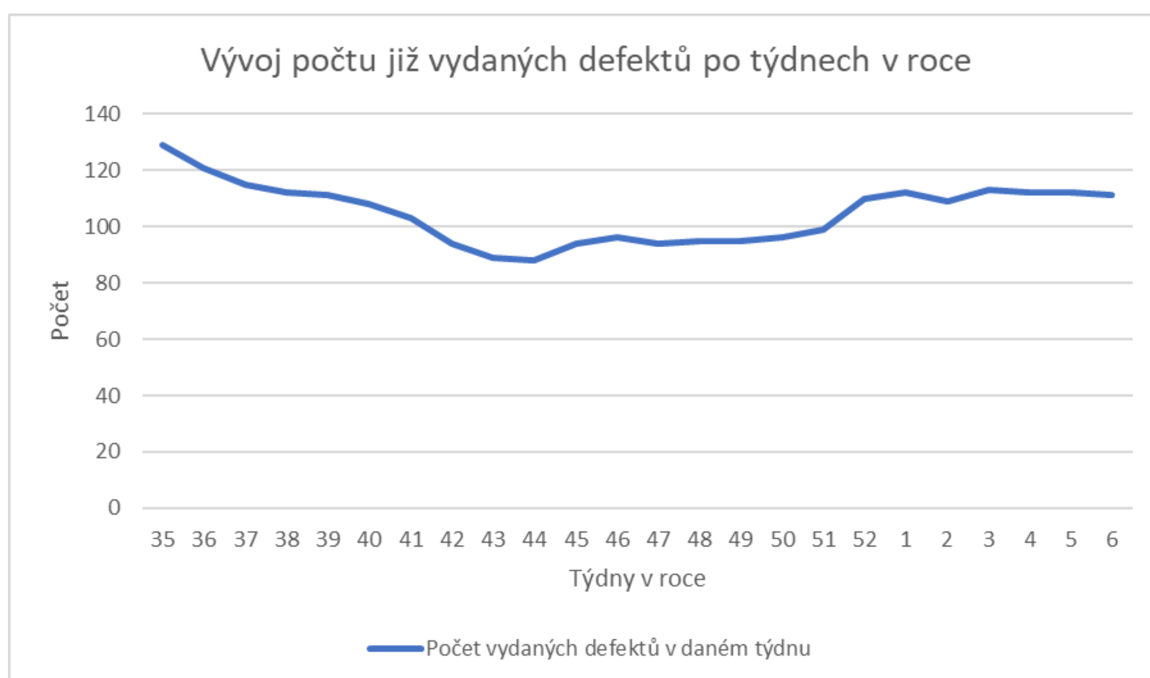
4.8.2 Grafy s vývojem počtu defektů

Na obrázku číslo 9 je vidět graf s vývojem počtu chyb průběžně v jednotlivých týdnech.



Obrázek 9 – graf s počtem všech evidovaných defektů

Obrázek číslo 10 je zaměřený pouze na defekty, které už se nacházejí na vydaných verzích softwaru. Osy zde mají stejný význam jako v předešlém grafu, a to tedy počet vydaných defektů průběžně v jednotlivých týdnech.



Obrázek 10 – graf s počtem vydaných defektů

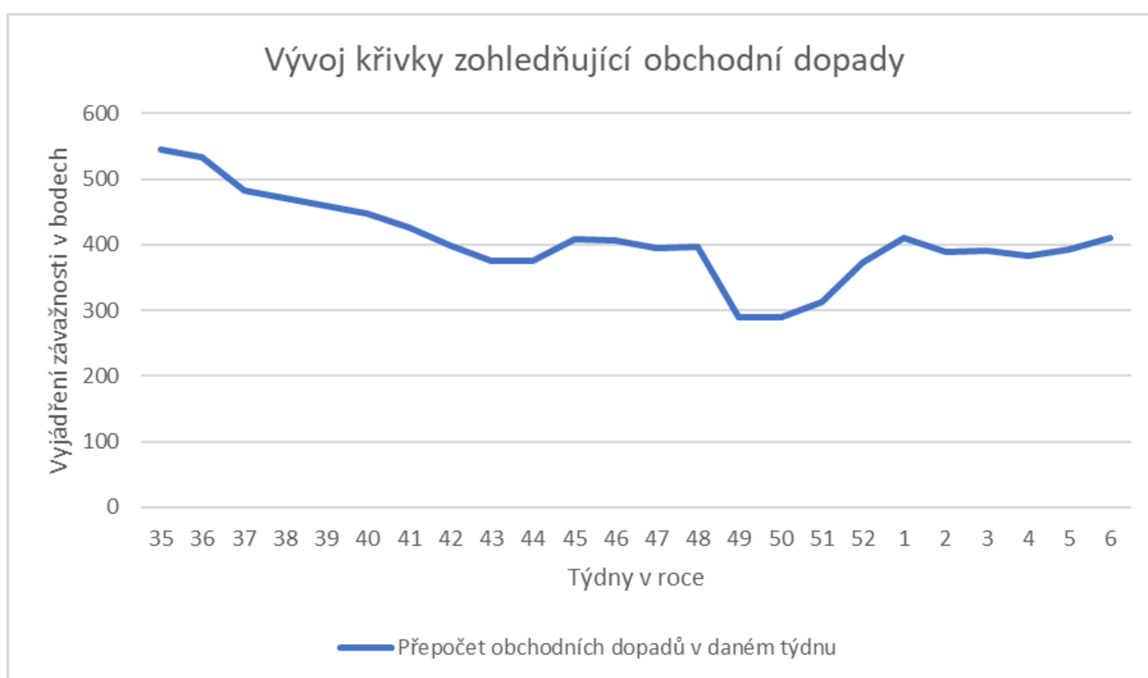
4.8.3 Graf se zohledněnou závažností

V této části bylo mým cílem sledovat lineární křivku zohledňující hlavně závažnost jednotlivých defektů, které už jsou dostupné ve vydaných verzích. Musel jsem tedy přistoupit k určitému přepočtu závažnosti na vyjádření poměru jednotlivými stupni možného hodnocení. Závažnost chyb bylo možné ohodnotit v rámci stupnice 1-3, kde 3 reprezentuje nejzávažnější. Rozhodl jsem se tedy pro násobení mocnin číslem 10. Princip jsem zvolil takový, že počet defektů se rozdělil do třech skupin a každá tato skupina byla násobena 1x, 10x nebo 100x podle toho o jakou závažnost se jednalo. Nejzávažnější defekty byly násobeny největším násobkem (100x). Největší skoky v rámci tohoto grafu byly způsobené uvolněním nové verze softwaru, a to hlavně z důvodu vydání většího počtu defektů, které se následně promítly do tohoto grafu. Další skokové nárůsty mohlo způsobit regresní testovaní, při kterém častokrát bývaly odhalené i defekty, které už byly v předchozích verzích softwaru uvolněné a ihned se do této křivky také projevily.

Obrázek 11 – graf znázorňující vývoj defektů z pohledu závažnosti

4.8.4 Graf se zohledněným obchodním dopadem

Pro některé defekty mohlo být zajímavé sledovat i jejich obchodní dopady, protože například i přes jejich závažnost nemusely být problematické z obchodního pohledu. Pro toto sledování byl využitý stejný mechanismus přepočtu jako u sledování závažnosti defektů. Defekty tedy měly 3 možnosti ohodnocení jejich dopadu z pohledu obchodu, kde 3 znamenala nejvyšší dopad. Tyto úrovně potom byly násobeny koeficienty jedna, deset nebo sto. Z grafu je následně možné vyčíst přepočtený bodový koeficient opět v průběhu jednotlivých týdnů.



Obrázek 12 – graf znázorňující vývoj defektů z pohledu obchodních dopadů

5 Výsledky

5.1 Ověření platnosti odhadů

5.1.1 Odhady pro implementaci

Původní odhad na celkovou implementaci činil celkem 13 MD. Do tabulky níže jsem rozepsal reálně odvedené časy ve stejné struktuře, v jaké byly uvedené původní odhady.

Tabulka 6 – porovnání původních odhadů vůči reálným nákladům

Úkol	Reálný čas	Odhadovaný čas
Převzetí 35 testů včetně drobných úprav	1.25 MD	2 MD
Implementace 5 nových testů	13.5 MD	10 MD
Nastavení automatického spouštění testů	0.5 MD	1 MD
Celkem	**15.25 MD**	**13 MD**

Vůči původním odhadům je zde vidět, že došlo k jejich mírnému překročení, a to konkrétně o 2.25 MD. Překročení odhadů bylo pouze v rámci implementace, ostatní dílčí činnosti byly realizovány s drobnými rezervami. Překročení odhadů v rámci implementace bylo způsobené nečekanými komplikacemi při práci s novými prvky uživatelského rozhraní, které ještě nebyly v aplikaci nikde použité, s čímž jsem při prvotních odhadech nepočítal.

5.1.2 Odhady pro údržbu

Měsíčně byla údržba celkem odhadnutá na 1.75 MD. Vzhledem k tomu, že v rámci této práce jsem sledoval údržbu po dobu 3 měsíců, budu se zabývat vyhodnocením pouze tohoto časového úseku. Na tyto 3 měsíce tak vychází dotace 5.25 MD. Za sledované tři měsíce byla vykázaná práce celkem 3.75 MD. V porovnaní pro první 3 měsíce vznikla rezerva vůči původním odhadům 1.5 MD, která může být využitelná například v případě komplikací v některých dalších měsících.

5.2 Počet nalezených chyb automatickými testy

Po implementaci celé sady testů jsem v časovém období třech měsíců sledoval, jestli tyto testy odhalí nějaké nově zanesené defekty, které nebyly odhaleny přímo při implementaci. Defekty odhalené automatickými testy byly nakonec ve sledovaném období čtyři. Tři z těchto defektů byly na stupnici závažnosti 1-3 označené prostředním číslem 2 a ten jeden zbývající číslem 1, které značilo nejmenší závažnost.

5.3 Ověření hypotéz

Stanovenou hypotézu číslo 1 pro implementaci automatických testů v původních odhadech nelze potvrdit, protože v rámci implementace došlo k překročení původních odhadů o 2.25 MD.

Hypotézu číslo 2 lze potvrdit, protože údržba automatických testů nepřesáhla původní odhad ve sledovaném období 3 měsíců. Naopak zde vznikla rezerva v podobě 1.5 MD vůči původním odhadům.

5.4 Stanovení metriky pro sledování kvality

Na základě sledování vývoje počtu defektů v softwaru byla stanovena metrika, dle které se začalo regulovat vydávání softwaru. Metrika je založená na hranici bodů pro graf se zohledněnou závažností defektů. Hranice byla stanovena po dohodě s produktovým managementem. Pokud byla tato hranice překročena, vyhradila se ve vývoji větší kapacita právě na opravy vzniklých defektů za účelem snížení této křivky a možnosti tak daný softwaru dodat zákazníkům v příslušné kvalitě. Do budoucna jsem navrhnul přenést tuto koncepci na kombinaci závažnosti defektů a jejich obchodních dopadů, což by mělo zpřesnit pohled na stav defektů.

6 Závěr

V rámci teoretické části práce proběhlo zpracování informací, které jsem následně využíval ve své praktické části. Konkrétně se jednalo o problematiku procesu automatizace testování softwaru a možnostech, které automatizace může přinášet. Druhou část teoretické práce tvořily témata související s kvalitou softwaru.

V praktické části jsem zpracoval business-case, dle kterého následně proběhla implementace automatických testů a procesu jejich automatického spouštění. Dále jsem sledoval vývoj defektů pro testovaný software.

Za výsledky mé práce považuji obhájený business-case, vyhodnocené stanovené hypotézy a jednoduchou metriku pro sledování kvality softwaru z pohledu počtu defektů.

7 Seznam použitých zdrojů

1. BUREŠ, M. -- RENDA, M. -- DOLEŽEL, M. -- Efektivní testování softwaru, 2016. ISBN 978-80-247-5594-6.
2. ELFRIEDE, D. -- Implementing Automated Software Testing, 2009. ISBN 0321580516.
3. ROUDENSKÝ, P. -- Kvalita softwaru TEORIE A PRAXE, 2018. ISBN 978-80-7402-322-4.
4. FEWSTER, M. -- Software Test Automation, 1999. ISBN 9780201331400
5. LEWIS, W E. -- VEERAPILLAI, G. *Software testing and continuous quality improvement.* Boca Raton: Auerbach Publications, 2005. ISBN 0849325242
6. Cyklus vývoje softwaru [online]. 2020 [cit. 2020-12-10]. Dostupné z: https://agiletech.medium.com/top-6-software-development-life-cycle-sdlc-models-methodologies-14e421bb4dd1
7. Strategie automatického testování [online]. 2019 [cit. 2020-12-10]. Dostupné z: https://www.simform.com/test-automation-strategy/
8. Automatizace testování [online]. [cit. 2020-12-14]. Dostupné z: https://www.symbio.com/solutions/quality-assurance/test-automation/
9. Automatizace testování [online]. 2019 [cit. 2020-12-14]. Dostupné z: https://www.cleveroad.com/blog/test-automation-roi
10. Charakteristiky kvality softwaru [online]. [cit. 2021-01-21]. Dostupné z: https://www.it-cisq.org/cisq-supplements-isoiec-25000-series-with-automated-quality-characteristic-measures/
11. Fine Code Coverage – doplněk pro sledování pokrytí kódu [online]. [cit. 2021-02-10]. Dostupné z: https://marketplace.visualstudio.com/items?itemName=FortuneNgwenya.FineCodeCoverage
12. SOLID [online]. 2017 [cit. 2021-02-10]. Dostupné z: https://medium.com/@alinepaulucci/solid-no-pick-up-stick-mess-to-your-software-37bce9b3356f