ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta elektrotechnická

Katedra ekonomiky, manažerství a humanitních věd

Analýza využití test managementu v bankovnictví

Analysis of using test management in banking

Diplomová práce

Studijní program: Elektrotechnika, energetika a management
Studijní obor: Ekonomika a řízení elektrotechniky
Vedoucí práce: Ing. David Němeček

Radek Soukup

Praha 2014

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra ekonomiky, manažerství a humanitních věd

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Soukup Radek

Studijní program: elektrotechnika, energetika a management
Obor: ekonomika a řízení elektrotechniky

Název tématu: Analýza využití test managementu v bankovnictví

Pokyny pro vypracování:

- test management a jeho využití v bankovním sektoru
- základní charakteristika analyzované banky
- vymezení oblasti analýzy – řízení zaměstnanců
- popis zavedeného systému
- analýza využití systému a návrhy na jeho zlepšení, včetně ekonomického hodnocení

Seznam odborné literatury:

Podle pokynů vedoucího DP.

Vedoucí diplomové práce: Ing. David Němeček – Raiffeisenbank a.s.

Platnost zadání: do konce letního semestru akademického roku 2013/2014

Doc.Ing. Jaroslav Knápek, CSc.
vedoucí katedry

Prof.Ing. Pavel Ripka, CSc.
děkan

V Praze dne 15.11.2012

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a v souladu s Metodickým pokynem o dodržování etických principů pro vypracování závěrečných prací, a že jsem uvedl všechny použité informační zdroje.
Nemám námitky proti použití tohoto školního díla ve smyslu § 60 zákona č. 121/2000 Sb., o autorských právech a právech souvisejících, ve smyslu pozdějších znění tohoto zákona.

V Praze dne 5. 5. 2014 ……………………………………

Poděkování

Děkuji Ing. Davidovi Němečkovi za vedení mé diplomové práce a věcné připomínky při její tvorbě. Děkuji Ing. Jiřímu Samkovi, CSc. a prof. Ing. Oldřichovi Starému CSc. za věcné připomínky a doporučení k mé práci.
Dále děkuji mým rodičům, přítelkyni i zaměstnavateli za podporu a vytvoření příznivých podmínek po celou dobu vypracování diplomové práce.

Abstrakt

Práce je zaměřena na oblast testování bankovních aplikací. Úkolem tohoto procesu je nalézt a odstranit případné chyby v bankovních aplikacích. V úvodní části práce jsou analyzovány procesy provádění a řízení testování bankovních aplikací. Návazně jsou řešeny otázky optimalizace organizačního modelu a strukturování testovacích týmů. Cílem práce je nalézt vhodné metriky a metodu oceňování i vyhodnocování nákladovosti procesů testování. Aplikační část práce tvoří případová studie hodnocení skupiny testů bankovních aplikací s vyhodnocením vynaložených nákladů a celkové efektivnosti.

Klíčová slova

Bankovnictví, aplikace, chyby aplikací, model, testování, proces testování, organizace a řízení, analýza a třídy chyb, metrika, rizika, náklady, optimalizace nákladů, test management, CMMI

Abstract

The work is focused on the testing of banking applications. The purpose of this process is to find and fix any errors in banking applications. In the first part of the study analyzed the implementation process and control testing of banking applications. Following are the questions addressed by optimizing the organizational model and structure testing teams. The goal is to find an appropriate method of valuation metrics and evaluation of costs and testing processes. Application of case study of evaluation of group tests of banking applications, assessing the cost and overall efficiency.

Key words

Banking, applications, application errors, model testing, the testing process, organization and management, analysis and error classes, metrics, risks, costs, cost optimization, test management, CMMI

Obsah

1 Úvod 11

2 Testování software 14

2.1 Co je úkolem testování 14

2.2 Proces testování 16

2.2.1 Plánování testů 16

2.2.2 Test plán 17

2.2.3 Test management 18

2.2.4 Test analýza 19

2.2.5 Realizace testů a reportování chyb 19

2.2.6 Reportování testů 21

2.3 Úrovně a typy testů 22

2.3.1 Způsob testů 22

2.3.2 Forma testů 23

2.3.3 Metoda testů 23

2.3.4 Fáze testů 24

2.3.5 Úrovně testů 25

2.3.6 Typy testů 26

2.4 Chyba 28

2.4.1 Úplné otestování je nemožné 29

2.4.2 Cena chyby 30

2.4.3 Pravděpodobnost opravy chyby na době nalezení 30

2.4.4 Rostoucí cena chyby na fázi cyklu vývoje SW, ve které byla chyba odhalena ... 31

3 Základní charakteristika analyzované banky 34

3.1 Obecný popis procesu vývoje SW v bance 34

3.2 Okruhy oblastí aplikací 34

3.2.1 Hlavní transakční systém 35

3.2.2 Databáze údajů o klientech CRM ... 35
3.2.3 Schvalovací aplikace ... 35
3.2.4 Řízení rizik – RISK ... 36
3.2.5 Datový sklad - DWH ... 36
3.2.6 Obchod s cennými papíry - TREASURY ... 36
3.3 Proces vývoje aplikací ... 37
3.4 Vývojové týmy ... 38
3.5 Vymezení oblasti analýzy – řízení zaměstnanců ... 39
4 Organizace a řízení procesu testování v bance ... 39
4.1 Historický stav testování ... 40
4.2 Původní stav ... 41
4.3 Nový stav ... 46
5 Finanční zhodnocení variant organizace testování ... 50
5.1 Stanovení vnitřní ceny testování ... 51
5.1.1 Předpoklady pro použitelnost modelu pro výpočet vnitřní ceny testování ... 53
5.1.2 Podmínky pro nalezení minimální ceny systému ... 56
5.2 Příklad aplikace výpočtu vnitřní ceny testování ... 57
5.2.1 Údaje k příkladu ... 58
5.2.2 Společné podmínky pro obě varianty ... 60
5.2.3 Výpočet pro variantu a) ... 61
5.2.4 Výpočet pro variantu b) ... 61
5.2.5 Citlivostní analýza výsledků ... 62
5.3 Ekonomické zhodnocení původního a nového stavu ... 63
5.3.1 Vymezení ekonomického hodnocení původního a nového stavu ... 63
5.4 Způsob porovnání výnosů a nákladů testování ... 65
5.5 Stanovení nákladů ... 67
5.6 Stanovení výnosů ... 69
5.6.1 Stanovení vhodných metrik v procesu testování ... 70
5.6.2 Koeficienty pro stanovení výnosů z testování ... 72

5.6.3 Koeficient kvality testování T_k 74
5.6.4 Duplicita chyb 76
5.6.5 Pravidla pro přiřazení nalezené chyby do koeficientů K_x 77
5.6.6 Pokrytí odlišností dvou různých testování koeficienty a pravidly 78
5.6.7 Povaha a vypovídací hodnota hlavních koeficientů 79
5.6.8 Data pro stanovení koeficientů 79
5.6.9 Stanovení kvantitativních koeficientů 81
5.6.10 Stanovení kvalitativních koeficientů množiny posuzovaných aplikací 84
5.6.11 Výsledek kvantitativních koeficientů a vyhodnocení dat 85
5.6.12 Výsledek kvalitativních koeficientů a vyhodnocení dat 87

5.7 Vyhodnocení úspory nákladů 90

6 Návrhy na vylepšení systému test management 91

6.1 Model zralosti softwarových procesů CMMI 92
6.1.1 Stupně zralosti modelu CMMI 92
6.1.2 Metriky obecné 93
6.1.3 Popis metrik používaných v bance pro hodnocení výkonnosti testování 95
6.1.4 Vylepšení metrik používaných v bance pro hodnocení výkonnosti testování 96

6.2 SWOT analýza navrhovaných vylepšení 97

7 Závěr 99

8 Seznam použité literatury 100

1 Úvod

Současné bankovnictví i celý finanční sektor je významným odvětvím každé vyspělé ekonomiky. Podobně jako jiné ekonomicky významné aktivity stále více využívá moderní nástroje ICT. Dnešní bankovnictví je založeno na poskytování širokého spektra bankovních služeb a produktů, které jsou často provázány.

Prakticky celý soubor bankovních produktů a služeb od úvěrových přes kapitálový trh až po platební styk využívá podpory ICT. Přístup uživatelů k těmto službám musí být zajištěn z různých platforem počínaje bankomaty, přes PC, chytré telefony i tablety, QR kódy až po standardní mobilní telefon.

Narozdíl od řady jiných užití ICT má bankovnictví určitá specifika. Je požadován zejména nepřetržitý provoz (24/7), trvalý a současný přístup velkého množství uživatelů, a to při vysokém stupeň ochrany proti neoprávněným přístupům. Z toho vyplývá, že bankovní systémy musí mít velký provozní "výkon" a současně vysoký stupeň ochrany a provozní spolehlivosti. Z pohledu přístupů uživatelů jsou to otázky identifikace, autentizace a zabezpečených přenosů dat při komunikaci. Komplexnost portfolia služeb a uživatelská přívětivost přístupu k těmto službám je v současnosti jedním z významných konkurenčních faktorů každé z bank.

Bankovní aplikace mají svá specifika a jedno z nejvýznamnějších je požadavek na výkonnost a kvalitu. Je kladen velký důraz na to, aby byly tyto bankovní aplikace spolehlivé a důkladně prověřené již před nasazením do provozu. Chyby, které se projeví až u produkčního systému, mohou mít fatální následky. Nefunkční systém pro převod transakcí znamená obrovské finanční ztráty banky nebo klientů. S tím pak souvisí ztráta důvěry a odliv klientů i celkové poškození image banky. Jako příklad mohu uvést, že u jedné z velkých bank v ČR je jedna hodina výpadku transakčního bankovního systému je oceněna na 10 - 15 mil. Kč ztrát jenom z úrokových výnosů [11]. Současně existují i právní rizika, kdy např. při chybném odeslání

emailů s klientskými informacemi se banka může dopouštět porušení zákona o ochraně osobních údajů. Důkladné otestování všech bankovních aplikací je proto nezbytné.

Pro zajištění spolehlivého provozu bankovních systémů využívajících ICT je významnou a velmi aktuální činností testování (testing). Oblasti řízení procesů a vlastnímu testování SW (test management) se proto věnuje v bance velká pozornost, neboť se jedná o velmi specifický proces s mnoha odlišnostmi od aplikačního nasazování jiných typů vyvíjeného a testovaného SW, hlavním specifikem je současné testování produktu i jeho podpory ICT. Testování bankovních produktů a současně jejich podpory ICT není zatím plně dobře systémově zvládnuté a procesně popsané i metodicky zpracované. Významnou součástí řešení dané úlohy je stanovení optimálního rozsahu testování a s tím související finanční náklady.

Cílem mé práce je popsat proces testování, náklady a přidanou hodnotu testování aplikací v bance. Účelem je definovat, čím se test management zabývá a jak jej udělat co nejefektivnějším z pohledu organizace a řízení, přípravy i provádění testů a také řízení a monitorování výkonnosti zaměstnanců - testerů SW a aplikací. Velký důraz jsem kladl na způsob stanovení vhodných ukazatelů výkonnosti a důkladné promyšlení všech faktorů, které proces testování ovlivňují tak, aby ukazatelé výkonnosti byly v co nejvyšší míře objektivní.

Řízení zaměstnanců je zde zaměřeno zejména na ekonomické hledisko. V této práci se zabývám vyhodnocováním výstupů práce testerů a stanovením způsobu měření, který určí hodnotu těchto výstupů z kvalitativního i kvantitativního hlediska. Hlavním ekonomickým přínosem procesu testování musí být vyšší přidaná hodnota výstupu testování než náklady, které vlastní testování obnáší.

V závěrečné kapitole shrnuji hlavní návrhy na zlepšení procesu testování, které mají za úkol zvýšit úroveň stupňovitého modelu zralosti CMMI. Vyjmenovávám hlavní nedostatky aktuálně zavedených metrik a provádím SWOT analýzu navrhovaných vylepšení.

Výsledky mohou být použité při stanovení vnitřní ceny testování, zejména při otázce, kdy se testování vyplatí. Využit může být i zvolený postup při stanovení koeficientů pro ukazatele

kvalitativních i kvantitativních výstupů procesu testování, především okolní vlivy ovlivňující podmínky při testování, které jsem pro stanovení vyjmenoval a posuzoval.

Zdroje, ze kterých jsem vycházel, jsou informace a expertní sdělení z prostředí velké banky působící na českém trhu. Jako zaměstnanec jsem vázán dohodou o mlčenlivosti a proto záměrně nikde v této práci neuvádím, o jakou banku se jedná. Názvy aplikací a všechna citlivá data byla změněna tak, aby nebylo možné rozpoznat žádný údaj, který není veřejně přístupný. Všechny číselné hodnoty byly mírně pozměněny tak, aby byla zachována správnost údajů, které reprezentují.

Součástí aktivit při zpracování diplomové práce byla rozsáhlá rešerše informačních zdrojů - knihy, odborné články, informace z webu. Přehled nejdůležitějších titulů je uveden v příloze s komentářem popisujícím hlavní získané poznatky.

2 Testování software

Tato kapitola nabízí stručný pohled na oblast testování software. **Řízením testování se zabývá disciplína test managementu**. Pro celkové pochopení této disciplíny v prostředí bankovnictví je třeba si nejprve uvědomit, co je účelem testování, jakým způsobem testování probíhá, co zahrnují jednotlivé typy testů, role v týmu a dalších. Testování software je obecná kapitola, která platí nejen pro testování bankovních aplikací, ale i ostatních aplikací a systémů.

2.1 Co je úkolem testování

Podle [5] „**hlavním úkolem softwarového testera najít chyby, najít je co nejdříve a zajistit jejich nápravu**“. Bez zajišťování opravy chyb by jejich nalezení nemělo smysl, přesto však ne vždy dojde k opravě všech chyb. **Důvody kvůli kterým nemusí dojít k opravě chyb** mohou být různé:

- **Na opravu chyby není vzhledem k její závažnosti čas** - jeden z důvodů proč je nutné chyby odhalit co nejdříve, čím dříve tester chyby reportuje, tím více času bude mít programátor k jejich opravě. Pokud je nutné dodat produkt (aplikaci) do určitého termínu (např. nutná změna smluv podle Nového Občanského zákoníku) a nalezená chyba, jejíž oprava by tento termín ohrozila není příliš závažná, je možné opravu odložit a dodat z počátku aplikaci s touto chybou.

- **Oprava chyby se nevyplatí** - různé chyby jsou různě náročné na opravu. Pokud by byla oprava velmi nákladná (znamenala by velkou změnu v aplikaci a tudíž mnoho práce programátora) a zároveň se chyba vyskytuje jen velmi zřídka, je možné že se z ekonomických důvodů nevyplatí ji opravovat.

- **Nejedná se o chybu, ale o vlastnost** - v některých případech se může ukázat, že došlo k nesprávnému pochopení zadání a na první pohled jednoznačná chyba je ve skutečnosti požadovanou vlastností systému.

- **Oprava je příliš riskantní** - toto může platit v případě napjatého časového plánu projektu. U drobných chyb, jejichž oprava by znamenala vysoké riziko zanesení dalších, závažnějších chyb, je možné, že bude rozhodnuto o jejich ponechání místo opravy. V takovém případě je vhodné na ponechanou chybu upozornit jako na známou chybu v produktu a vyhnout se riziku zavlečení nových, neznámých chyb.

- **Programátor neví, jak chybu opravit** - v těchto případech může být oprava chyby jednoduchá, bezriziková, levná a rychlá ale kvůli omezeným schopnostem programátora nedojde k její včasné opravě.

Pro pojem „test“ a „testování“ existuje více definic. Podle IEEE **je testování porovnáním očekávané a existující situace**. Existující situací je zde myšlena testovaná aplikace a očekávanou pak požadavek zadavatele (obchodní útvary banky) formou zadání (produktové specifikace).

Hlavní cíle testování jsou :

- **Validace** - ověření jsou-li všechny požadavky správně zahrnuty v rámci požadavku (SW vyhovuje požadavkům uživatele)

- **Verifikace** - ověření je-li specifikace správně implementována (SW vyhovuje specifikaci)

- **Identifikace rozporů** - nalezení a zaznamenání připomínek ke kvalitě produktu co nejrychleji a vyvinout maximální úsilí k jejich řešení

- **Ostatní** - Evidence výsledků testů, Reportování testů, Vyhodnocení testů, Návrhy na zajištění zlepšení kvality SW

Tester chyby vyhledává, reportuje, snaží se zajistit jejich nápravu, ale nerozhoduje o tom, zda se chyba opraví nebo bude-li produkt dodán, pokud se chyba neopraví. Testováním jsou pouze získány informace o chybách a jejich opravě a stavu kvality SW, nikoliv rozhodnutí o tom zda bude SW akceptován.

Testování nerovná se zajišťování kvality (QA – quality assurance), osoba zodpovědná za zajišťování kvality softwaru vytváří a prosazuje standardy a metody zdokonalující proces vývoje a minimalizující vznik chyb. Zde platí známý fakt - kvalitu nelze získat pouze kontrolou, ale jej ji nutno založit již při tvorbě aplikace a jejího SW řešení.

2.2 Proces testování

Testování probíhá jako ucelený proces, ve kterém mají hlavní aktivity (procesy) danou systémovou posloupnost. Mezi tyto aktivity obecně řadíme **Plánování testů, Test Management, Test analýzu, Test realizaci a Reportování testů**. Tato kapitola stručně popisuje jak jednotlivé procesy testování probíhají.

2.2.1 Plánování testů

Po zahájení projektu je prvním procesem plánování testů. Tato aktivita je klíčová, protože se během ní rozhodne, jak a ve které fázi budou konkrétní činnosti týkající se testingu přesně probíhat. Součástí plánování testů je i hrubý odhad věcné i časové náročnosti, který je

v pozdějších fázích doplněn a podle něj je možno naplánovat a rozvrhnout zdroje. Stěžejním dokumentem plánování testů je Test plán, tento dokument popisuje aktivity vycházející z plánování testů.

2.2.2 Test plán

Dokument Test plán je základem pro proces řízení testů, monitorování průběhu testů a hodnocení efektivity a výsledků testů. S dobře vypracovaným test plánem je možné včas podchytit potenciální rizika a řešit problémy, které se během provádění testů objeví (např. nedostatečné kapacity).

Test plán by měl obsahovat informace o následujících činnostech:

- definici testovacích rolí a jejich přirazení konkrétním členům týmu
- rámcový harmonogram (časový plán) projektu, na který pak navazuje harmonogram testování (milníky projektu)
- rámcovou specifikaci testovacích dat (detailní je součástí test analýzy)
- definování předmětu a rozsahu testování
- specifikaci co je a co není předmětem testování
- specifikace typů testů, které budou použité
- seznam aplikací a databází zahrnutých do testů
- specifikaci Testovacího prostředí
- vstupní a výstupní kritéria pro Integrační a Systémové testy
- vstupní a výstupní kritéria UAT testů
- požadované součinnosti pracovníků nebo útvarů, jenž se na testování přímo nepodílí
- detailní plán/harmonogram testů (pokud to vyžaduje velikost projektu)
- věcné ocenění a způsob práce s riziky

Test plán tedy mimo jiné definuje i **Test strategii** – popis jakou technikou bude produkt testován, a to jak z hlediska celkového produktu, tak každé jeho jednotlivé fáze (pro kterou

část aplikace bude použito manuální testování a kde bude lepší použít automatické, ve které fázi se bude testovat technikou Black box a ve které White box atd.)

2.2.3 Test management

Test management je řízení procesu testování SW. Tuto činnost vykonává zejména test manager, někdy se pro roli test manager používá označení test koordinátor. Kromě role test manager se však na aktivitě test management podílí více rolí, a to jak z oblasti testování (analytik, tester) tak z oblasti vývoje, analýzy i řízení projektů.

Podle metodiky vývoje software RUP má test manager na starosti následující úkoly :

- Projednává cíle testování a výstupní kritéria
- Zabezpečuje vhodné plánování a řízení zdrojů pro testování
- Vyhodnocuje výstupy a účinnost testování (test report)
- Zajišťuje potřebnou úroveň kvality rozhodování o důležitých defektech
- Stanovuje vhodnou úroveň zaměření na testování v procesu tvorby SW

Test manager má celkovou zodpovědnost za úspěch procesu testování. Je zodpovědný za prosazování kvality a testování, řízení a plánování zdrojů, rozhodování u sporných otázek ovlivňujících proces testování. Řídí celý proces testování, schvaluje požadavky na testování, schvaluje plán testování, přiděluje zdroje (HW, SW, lidské zdroje). Je zodpovědný za celkovou kvalitu testování a naplnění cílů testování definovaných v dokumentu Test plán.

2.2.4 Test analýza

Před realizací samotných testů je potřeba provést test analýzu. Pokud se jedná o krátké ad-hoc testování, je možné test analýzu vynechat, nicméně důležitost test analýzy roste s tím jak je projekt rozsáhlý a jak často se bude testování opakovat. Ze specifikace požadavků, tedy analýzy aplikace, kterou píše analytik, lze odvodit některé funkce programu ještě předtím, než jsou naprogramovány. Z toho důvodu může začít test analýza i ve chvíli kdy není ještě žádná část aplikace vytvořená. **Úkolem test analýzy je navrhnout co a jakým způsobem testovat, aby byla aplikace co nejlépe otestována s minimálním úsilím**.

Výstupem test analýzy je dokument obsahující testovací scénáře. Tyto scénáře jsou procesní postupy, které popisují dostatečně podrobně jakým způsobem a co přesně testovat, jaká jsou potřebná test data a počáteční podmínky u konkrétního testu. Dále jaký je očekávaný výstup testu (co se má zobrazit, které hodnoty), informace o výsledku testu (zda bylo testováno, pokud ano tak s jakým výsledkem, pokud byla chyba tak její identifikátor atd), oblast testování u konkrétního testu, požadavek, kterého se test týká a další. Díky test analýze je pak velmi dobře zdokumentováno, co se testovalo a s jakým výsledkem. Je možné při její tvorbě najít chyby ve specifikaci ještě dříve, než se jí začne zabývat vývojář.

Test analýza nabízí i prostor pro zamyšlení se nad tím, které scénáře se budou testovat. Scénáře mají různou prioritu a ne vždy je časový prostor na to otestovat vše. Tester si poté často klade otázku na základě čeho vybírat test data a množinu testovacích scénářů. **Úkolem je vybrat minimální množinu dat a scénářů, která nalezne maximální množství chyb.**

2.2.5 Realizace testů a reportování chyb

Protože úkolem testera je mimo jiné najít chyby co nejdříve, vlastní realizace testů probíhá obvykle tak brzy, jak jen je to možné. Jako první nastává obvykle statické testování tím, že si tester pročte analýzu a hledá v ní rozpory, možné problémy, nepravdy, nepřesné formulace apod. Po nalezení těchto rozporů je dobré vždy nejprve probrat s analytikem, zda se jedná o chybu, tento postup je vhodné provádět i před reportováním chyb s vývojáři – předejde se tak

zbytečné práci s reportováním chyb které chybami nejsou a navíc tyto špatně zadané chyby znehodnocují výstup práce testera.

Pokud dojde ke změně v analýze, pak se jedná o chybu - kromě drobností typu překlepy apod. Ty chyby, které chybami jsou, pak za pomocí vhodného reportovacího SW tester zapíše a zadá k opravě analytikovi. Zde se tester setká s prvním problémem svého poslání: úkolem testera je svým způsobem **nacházet chyby v práci analytiků a vývojářů a poté je hlásit (reportovat)**, z tohoto důvodu **je nutné, aby byl tester i dobrým diplomatem.**

Některé **zásady pro správné reportování chyb**, které musí/by měl tester dodržet (platí pro testování analýzy i aplikace) :

- **Chyby je třeba oznamovat co nejdříve**.

- **Chyby je nutné účinně a přesně popisovat** – tak aby z popisu mohl programátor chybu jednoznačně nasimulovat a následně odstranit.

- **Při oznamování chyb musí být tester nestranný** – neutrální, neosobní a věcný, popis chyby musí přesně popisovat daný problém a např. nijak nekomentovat práci vývojáře/analytika

Také účinný **popis chyby (bugu)** by měl dodržet dále uvedené zásady:

- **Minimální** - stručný a věcný, nesmí obsahovat nadbytečné informace

- **Jednotlivý** - každá chyba by měla popisovat právě jednu věc, nemělo by se stát že jeden reportovaný bug bude popisovat několik chyb (jedním z důvodů je i to že se pak na ostatní chyby může zapomenout, při opravě případně při opětovném spuštění systému). Výjimkou bývá např. seznam více podobných chyb, pokud to situace umožní – např. nefunkční stejné tlačítko na mnoha obrazovkách, špatné pojmenování problému v analýze které se opakuje a podobně. V takových případech může být vhodné v jednom reportovaném bugu popsat více chyb.

- **Reprodukovatelný** - chybu, kterou nelze reprodukovat (popsaným způsobem znovu navodit), zpravidla není možné opravit. Někdy může být chyba způsobena jinou akcí, než kterou tester předpokládá, proto je třeba otestovat, zda chyba nastane i za jiných podmínek a z toho odvodit zda právě jimi nebyla chyba způsobena.

- **Jasný a obecný** - maximální množství informací které vývojář/analytik vyžaduje a žádné zavádějící, nejasné, nadbytečné nebo mnohoznačné informace. Zároveň co nejobecnější popis - pokud např. chybný výpočet určitého algoritmu nastává ve všech obrazovkách, je zbytečné psát jen jednu konkrétní obrazovku, ve které tester tuto chybu poprvé navodil.

Toto jsou obecné zásady pro definici a zadávání chyb, není nutné se všemi vždy řídit, záleží na konkrétním projektu, způsobu testování i domluvě členů týmu. Obecně praxe ukazuje, že reportování chyb je při dodržení těchto zásad nejefektivnější.

Po statické kontrole analýzy nastává obvykle testování aplikace, nejprve neformálními smoke testy, poté oficiálními testy. Každá chyba reportovaná pomocí určitého nástroje (SW) má svůj **životní cyklus**. Ten se liší podle použitého nástroje a procesu testování (domluvy), v zásadě jsou v tomto cyklu velmi častými stavy, kdy chyba je:

- **zadaná** (tester ji právě vytvořil)

- **stornovaná** (pokud řešitel usoudí, že je špatně zadaná a nejedná se o chybu)

- **opravená** (pokud ji řešitel opraví – tím ji obvykle vrátí zpět na testera k přetestování)

- **uzavřená** (tester ověří opětovným testem, že chyba byla správně opravena případně že opravou nedošlo k zanesení dalších chyb a uzavře ji)

2.2.6 Reportování testů

Po dokončení testování přichází na řadu vytvoření dokumentu zvaného **Test report**. Tento dokument vytváří obvykle tester, který dané testy prováděl. Jedná se o report obsahující všechny informace o proběhlém a dokončeném testování. Je v něm zaznamenáno datum zahájení a ukončení testů, název i verze testované aplikace, osoba zodpovědná za dodávku produktu (aplikace), za testy,za vývoj, otestované oblasti/scénáře, prostředí na kterém testy proběhly, testovací databáze – název i verze, seznam nalezených chyb (opravených i existujících), výsledek testu (příp. výsledky testů jednotlivých požadavků), rizika vyplývající z podmínek testování (např. nemožnost otestovat některé situace) apod.

Test report je v podstatě výstup z testů a tento dokument zachovává všechny potřebné informace o tom, jak, kde, kdy, kým, na jakém prostředí a s jakým výsledkem proběhly testy. Díky němu je vše řádně zdokumentováno a zákazník (zadavatel požadavku) si po přečtení test reportu může sám vyhodnotit, v jakém je dodávaná aplikace stavu.

2.3 Úrovně a typy testů

Testování lze rozdělit podle různých kritérií a hledisek. V této kapitole bude uvedeno základní rozdělení testů podle kategorií, jako je účel testování, způsob, fáze, forma a další.

2.3.1 Způsob testů

Testovat lze dvěma základními způsoby: Staticky a dynamicky.

- **Statické testování** je testování „něčeho, co neběží“. Jedná se např. o testování uživatelské specifikace, které lze provádět ještě před vývojem SW, pokud je k němu hotová analýza (specifikace). Dále lze staticky testovat zdrojový kód jeho kontrolou (často probíhá jako kontrola 4 očí mezi dvěma programátory), datový model, nápovědu v aplikaci atd.

- **Dynamické testování** vyžaduje vlastní spuštění aplikace. Toto testování se využívá v pozdějších fázích vývoje SW a vyžaduje alespoň prototyp aplikace, příp. hotové databázové procedury a prostředí, na kterém je lze spouštět a testovat. Dynamické testování tedy zahrnuje všechny typy testů, které uživatel provádí nad běžící aplikací.

2.3.2 Forma testů

Způsob testování a stupeň formality rozděluje testy na smoke, formální a neformální.

- **Smoke testy** jsou rychlé testy základních funkčností a slouží k ověření, že daná aplikace je vhodná k testování. Nejsou zde do detailu kontrolovány formáty vstupů a výstupů, testuje se jen ta základní funkcionalita, u níž nepředpokládáme, že se bude měnit. Např. při testování zadávání zahraničních plateb zkontrolujeme smoke testem to, že lze platbu zadat a tento základní scénář je funčkní, neřešíme již nevalidní formáty platby, měny a mnoho dalšího.

- **Formální testy** jsou řízené procesy. Je třeba, aby jim předcházela test analýza, chyby při nich objevené byly reportovány a nedošlo k vychýlení z vybraných testovacích případů.

- **Neformální testy** jsou ty testy, ve kterých tester sám určí, co a jak bude testovat. Komponenty a jednotlivé změny jsou označeny a zdokumentovány (analýza), ale nejsou zde určeny jednotlivé testovací případy k provedení. Nejsou tak dobře kontrolovatelné a dokumentované jako formální testy, ale v rukou zkušeného testera se mohou stát vhodným nástrojem k zlepšení způsobu případně množiny formálních testů do budoucna.

2.3.3 Metoda testů

Podle toho, jakou metodou probíhá testování, rozdělujeme jej na testování typu black box, white box a grey box.

- **Black box** znamená „testování černé skříňky“. Při tomto způsobu testování ví tester, jaký výstup dostane při určitém zadaném vstupu, ale neví, jakým způsobem funguje aplikace uvnitř. Proces, kterým dojde k transformaci vstupu na výstup, tedy není při black box testování zkoumán, je pouze porovnán očekávaný a známý výsledek. Tyto testy jsou realizovány bez znalosti vnitřní datové a programové struktury a využívají se tam, kde jsou přesně definované vstupy a rozsahy možných hodnot.

- **White Box**, tedy „testování bílé skříňky“, je testování kdy má tester přístup ke zdrojovému kódu programu a vidí tedy, jak celý proces probíhá uvnitř sw aplikace. Tester při tomto testování musí rozumět zdrojovému kódu, testování white box vyžaduje znalost vnitřních datových a programových struktur a také toho, jak je sw systém naimplementován.

- **Grey Box** je kombinace black box a white box. Tester nezná do detailu zdrojový kód jako u white box testování, ale zároveň má určitou představu o vnitřních strukturách programu a není to tedy čistě black box testing. Jestliže tester ví, jak funguje produkt uvnitř, umí ho lépe otestovat zvenku. Grey box probíhá jako black box testování, tester je však při něm obohacen o znalost vnitřní struktury programu.

2.3.4 Fáze testů

Podle toho, v jakém cyklu vývoje SW dochází k testování, rozdělujeme testy na fáze vývoje:

- **jednotkové (Unit) testy**

- **integrační a systémové testy**

- **uživatelské akceptační testy (UAT)** – akceptace dodávky uživatelem.

Fáze testů úzce souvisí s úrovní testů, v každé fázi jsou obvykle prováděny testy určité úrovně (např. není obvyklé aby se ve fázi vývoje prováděly uživatelské akceptační testy a ve fázi akceptačních testů testy jednotkové).

2.3.5 Úrovně testů

- **Unit testy** píše sám vývojář před předáním aplikace k otestování testerovi. Jedná se o testy sloužící k otestování dílčích částí zdrojového kódu. Za tyto části považujeme samostatně testovatelné části aplikačního programu.

  Definice jednotkových testů podle ISTQB :

  „Testování komponent hledá defekty uvnitř softwarových komponent a verifikuje fungování softwarových komponent (např. modulů, programů, objektů, tříd, atd.), které jsou testovatelné samostatně. Může být vykonáno v izolaci od zbytku systému v závislosti na kontextu životního cyklu vývoje a systému. Při testování komponent mohou být použity nástavce, ovládače a simulátory.“

- **Integrační testování** může být buď v rámci **vnitřní integrace**, tedy komunikace mezi jednotlivými komponentami uvnitř aplikace, a **vnější integrace** – komunikace mezi aplikací a integrovanými aplikacemi. Podle ISTQB : „Integrační testování testuje rozhraní mezi komponentami, interakce s různými částmi systému jako jsou operační systém, souborový systém, hardware anebo rozhraní mezi systémy.

  Může existovat víc než jedna úroveň integračního testování a integrační testování může být vykonáno na objektech testování různé velikosti.“

- **Systémové testování** se zaměřuje na aplikaci jako celek, tak, jak ji vnímá zákazník. Spojení integračního a systémového testování se nazývá System Integration Test (SIT). Jedná se o stěžejní úroveň testů.

Definice systémového testování podle ISTQB :

„Systémové testování se zabývá chováním celého systému/produktu, jak byl definován rozsahem vyvíjeného projektu nebo programu.

V systémovém testování by mělo testovací prostředí korespondovat s cílovým nebo provozním prostředím v co největší možné míře. Cílem je minimalizace rizika, že selhání, která jsou specifická pro dané sw prostředí, nebudou při testovaní nalezena.

Systémové testování může zahrnovat testy založené na rizicích a/nebo na specifikacích požadavků, obchodních procesech, případech užití nebo jiných popisech chování systému na vyšší úrovni. Dále mohou být testy založeny na interakcích s operačním systémem a na systémových zdrojích“.

- **Akceptační testování** – jedná se o testování zákazníkem (uživatelem), který na základě výsledků akceptuje dodávku sw aplikace pro nasazení do provozu. **User Acceptance Test (UAT) jsou tedy akceptační testy na straně zákazníka,** prováděné na jeho vlastním prostředí. Uživatel obvykle testuje ve spolupráci s vývojovým týmem podle svých vlastních scénářů. Podle ISTQB : „V akceptačním testovaní je cílem nastolení důvěry v systém, jeho části nebo specifické nefunkcionálním charakteristiky systému. V akceptačním testování není hlavním účelem nalezení defektů. Akceptační testování může ohodnotit připravenost systému pro nasazení a používání, i když není nevyhnutelně poslední úrovní testování. Například integrační testování rozsáhlých systémů může být vykonáno po akceptačním testování systému.“

2.3.6 Typy testů

Podle toho, zda-li je testování zaměřeno na funkce, které má aplikace vykonávat (podle specifikace, dokumentace, ústního zadání nebo toho, jak je chápána testery) nebo na

testování těch vlastností aplikace, které přímo nesouvisí s jejími funkcemi, ale zároveň jsou podstatné pro její správnou činnost, rozlišujeme dva základní typy testů : **Funkční** a **nefunkční**. Dále rozlišujeme typy testů v případech, kdy testujeme opravu chyby nebo pravidelně testovanou oblast na testy **konfirmační** a **regresní**.

- **Funkční testy** zahrnují obecně testování všech funkcí a vlastností, které má aplikace vykonávat a tyto testy ověřují, že pracují správně a odpovídají požadavkům zákazníka. Tyto funkce a vlastnosti mohou být popsané ve specifikaci produktu. Funkční testy mohou být vykonávány na všech úrovních testování (např. testy komponent mohou být založeny na specifikaci komponent).

- **Nefunkční testy** zahrnují testování vlastností aplikace, které nesouvisí přímo s jejími funkcemi, ale jsou důležité pro její správnou činnost. Patří sem testování výkonu, zátěžové testování, stres testování, testování spolehlivosti, udržovatelnosti, použitelnosti, přenositelnosti ale neomezuje se jen na ně. Termín nefunkcionální testování popisuje testy vyžadované pro měření charakteristik systému a softwaru, které mohou být kvantifikovány vůči různým stupnicím měření, jako například doby odezvy pro testování výkonu. I nefunkční testy mohou být vykonávány na všech úrovních testování.

- **Konfirmační testy -** pokud tester objeví chybu, reportuje ji a vývojář ji opraví, po opravě je potřeba aby tester zkontroloval (znovu otestoval) že byla chyba opravena správně a že její opravou nedošlo k zavlečení dalších chyb. Konfirmační testy mají tedy za úkol ověřit opravení/odstranění chyby a potvrdit tak správné opravení té části aplikace, u které dříve tester reportoval chybu.

- **Regresní testy** zahrnují opakované testování již testovaného programu po modifikaci s cílem najít všechny defekty, které byly zaneseny nebo objeveny jako důsledek změny (změn) provedených v souvislosti s odstraňováním chyby. Tyto defekty se mohou nacházet v testovaném softwaru nebo v jiné související nebo nesouvisející softwarové komponentě. Regresní testování se provádí, když je změněn software nebo jeho prostředí. K těmto testům bývá často vhodné vytvořit automatické testy, protože se jedná o často se opakující, stejně prováděné testy.

2.4 Chyba

Pojem chyba má mnoho významů a synonym - anomálie, závada, odchylka, selhání, defekt, nekonzistence, omyl a další. Podle [5] jsou chybou následující situace:

1. SW nedělá něco, co by podle specifikace produktu dělat měl
2. SW dělá něco, co by podle údajů specifikace produktu neměl dělat
3. SW dělá něco, o čem se produktová specifikace nezmiňuje
4. SW nedělá něco, o čem se produktová specifikace nezmiňuje, ale měla by se zmiňovat
5. SW je obtížně srozumitelný, těžko se s ním pracuje, je pomalý, nebo - podle názoru testera SW – jej koncový uživatel nebude považovat za správný.

Standard IEEE 610.12-1990 (R2002) pak rozlišuje následující pojmy:

1. **Omyl** (**Mistake**) : Lidská akce produkující nesprávný výstup. Příklad: nesprávná akce programátora.

2. **Závada** (**Fault**) : Nesprávný krok, proces nebo definice dat v počítačovém programu. Příklad: nesprávná instrukce nebo tvrzení.

3. **Chyba** (**Error**) : Rozdíl mezi vypočítanou, měřenou nebo pozorovanou hodnotou nebo podmínkou a pravdivou, specifikovanou nebo teoreticky správnou hodnotou nebo podmínkou.

4. **Selhání** (**Failure**) : Neschopnost systému nebo komponenty splnit své funkce definované požadavky na výkon

Název zde používaný pro tento pojem není podstatný, proto bude nadále vymezeno slovo „chyba“ odpovídající anglickému „bug“ pro situace podle definice [5], ve které jsou pojmy IEEE pro chybu zahrnuty.

2.4.1 Úplné otestování je nemožné

Testováním lze zjistit přítomnost defektů, nikoliv však jejich absenci. **Úplné otestování** bankovních **aplikací je** většinou prakticky **nemožné**, lze toho dosáhnout pouze u triviálních případů. Například pokud bychom měli otestovat jednoduchý součet dvou čísel A + B, kde A a B jsou 32-bitová přirozená čísla (2^32 = 4,294,967,295) a 1000 testů bude trvat pouze 1 vteřinu, celý test by pak zabral jednoduchým výpočtem 593,066,617 let :

Příklad : test všech možných kombinací součtů těchto dvou čísel :

(4,294,967,295 * 4,294,967,295) / 1000 / 60 / 60 / 24 / 360 = **593 066 617**

Z toho důvodu se při testování omezujeme na nalezení tříd ekvivalence, tedy minimalizaci množiny testovacích dat.

Třída ekvivalence, nebo-li množina ekvivalentních případů je taková množina testových případů, která testuje stejnou věc nebo odhaluje stejnou chybu. Např. při sčítání dvou čísel je vhodná třída ekvivalence sečtení maximálních možných hodnot, minimálních možných hodnot a dalších hraničních podmínek a jen několika málo středních hodnot (13+7 bude s vysokou pravděpodobností testovat to samé co 13+2).

Cílem rozdělení ekvivalentních případů je zredukovat množinu testovaných případů (dat) na menší podmnožiny, u kterých předpokládáme, že jsou pro otestování sw dostatečné. Tento předpoklad znamená riziko a pro testování je tedy důležité řízení rizik. Je vhodné se vždy snažit zahrnout do tříd ekvivalence hraniční podmínky, speciální znaky, prázdnou množinu apod. – všechny situace, které by mohly reálně nastat a ohrozit funkčnost systému.

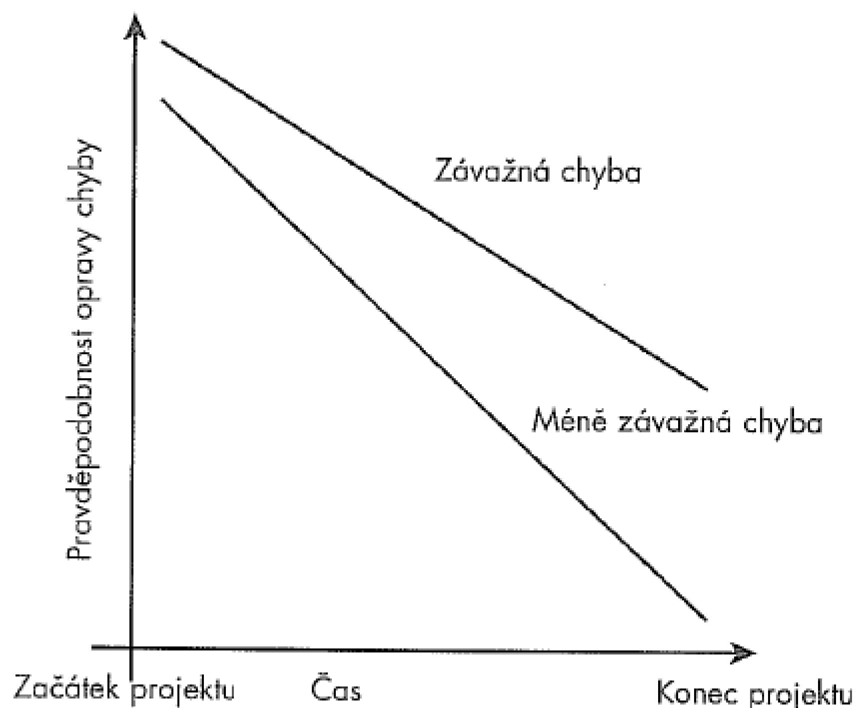
2.4.2 Cena chyby

Oprava každé chyby stojí určitou finanční částku. Pod pojmem „cena chyby“ rozumíme částku za nalezení a opravu chyby, ve které je započítán čas všech osob podílejících se na opravě samotné, ale i veškeré další náklady. Těmito náklady bývá např. reputační riziko[1], které chyba zapříčiní a s ním spojený odliv klientů, finanční ztráty způsobené neplatnými smlouvami které se kvůli chybě vygenerovaly a další. Pokud se chyba odhalí v průběhu testování, pak budou tyto další náklady ušetřeny. Pokud se odhalí ještě v době psaní analýzy, ušetří se i práce vývojáře a testera, protože dojde k opravě dříve, než to ovlivní jejich práci, při které by tester chybu reportoval a programátor opravoval. Cena chyby je silně závislá na čase, ve kterém se chyba odhalí, proto má tester za úkol odhalovat chyby co nejdříve. První příležitost má tester během statických testů kontrolou analýzy.

2.4.3 Pravděpodobnost opravy chyby na době nalezení

Čím dříve se chyba nalezne, tím levnější a u méně závažných chyb i pravděpodobnější bude její oprava. Na pravděpodobnost opravy chyby má také velký vliv její význam. Tuto skutečnost popisuje orientačně graf závislosti pravděpodobnosti opravy chyby na její závažnosti a čase, kdy byla chyba odhalena vzhledem k fázi konkrétního projektu:

[1] Ztráta image a dobrého jména banky.

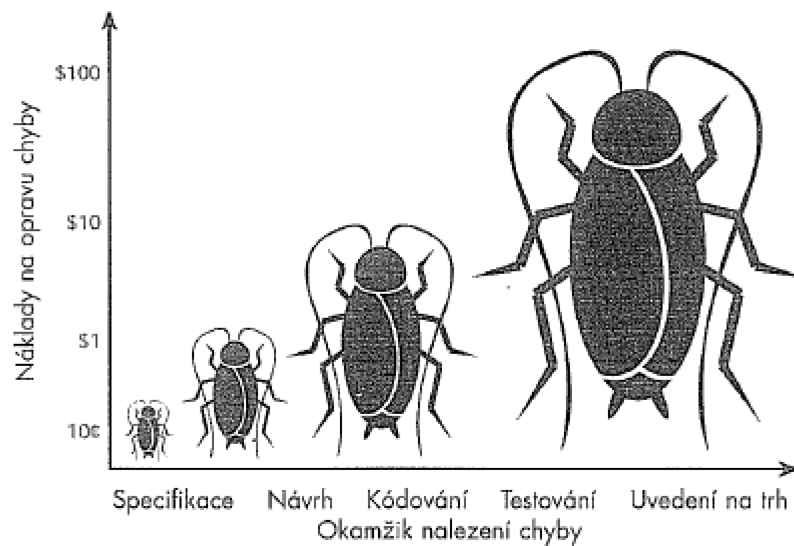


Obr.1 Pravděpodobnost opravy chyby v závislosti na době jejího nalezení [5]

2.4.4 Rostoucí cena chyby na fázi cyklu vývoje SW, ve které byla chyba odhalena

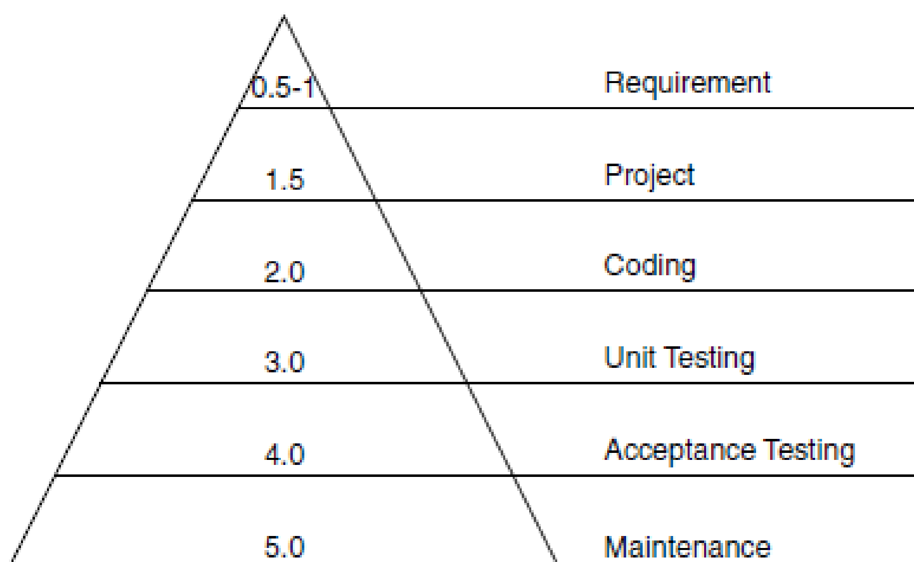
Protože se na vývoji SW podílí více osob, bude oprava chyby znamenat různé množství práce také podle toho kolik osob se muset nalezením a odstraněním chyby zabývat. Pokud dojde k jejímu odhalení již ve fázi zadávání požadavku (zadavatel si to uvědomí sám příp. při konzultaci s analytikem), změní se zadání a programátor a tester s tím nebudou mít žádnou práci. Naopak pokud se na chybu přijde až po nasazení SW do produkce, bude s tím mít práci více účastníků (zadavatel, analytik – předělání analýzy, vývojář – vlastní oprava, tester – přetestování, změna scénářů) a navíc tato změna může znamenat reputační i finanční riziko. Rovněž budou muset i všichni členové vývojového týmu strávit čas připomenutím problematiky SW, jelikož od jeho nasazení mohla uplynout značná doba.

Obecně zobrazuje tuto skutečnost známý graf:



Obr. 2 Rostoucí náklady na opravu chyby podle okamžiku nalezení [5]

Orientační zobrazení poměru nákladů na fázi cyklu vývoje SW, ve které byla chyba nalezena, zobrazuje podle zdroje [9] graf od Boehm a Basili z roku 2001 :



Obr. 3 Poměry nákladů na opravu chyby podle fáze [9]

Kde nalezení chyby na produkci znamená cca 5X vyšší náklady na opravu než při odhalení chyby již při specifikaci požadavku. Není zde zohledněno reputační a finanční riziko způsobené nasazením, pouze vlastní náklady na opravu. Tyto hodnoty se samozřejmě výrazně liší v závislosti na rozsahu projektu, typu chyby a dalších vlivech, vždy platí, že čím později je chyba odhalena, tím je její oprava nákladnější.

3 Základní charakteristika analyzované banky

Banka, jejíž test management v této práci uvažuji, je na českém trhu jedna z větších. Tato banka má počet klientů v řádu statisíců, není nízkonákladová - jde o banku snažící se konkurovat především kvalitou a současně rozsahem poskytovaných služeb. Tato banka poskytuje široké spektrum produktů a prezentuje se spíše jako prémiová. I proto je zde kladen velký důraz na důkladné otestování vysokého množství rozličných aplikací na podporu různých produktů pro retailové i korporátní klienty. Současně je snaha minimalizovat náklady.

3.1 Obecný popis procesu vývoje SW v bance

K pochopení podrobnějších detailů procesu testování je nutné si uvědomit, jak celý proces vývoje SW ve standardní bance funguje. Tento proces je popsán v této kapitole. Nejprve pro přehled uvedeme, jakých oblastí se týkají aplikace, které se ve standardní bance zpravidla vyvíjejí.

3.2 Okruhy oblastí aplikací

Pro pochopení logiky rozdělení aplikací v bance, nad kterými je realizováno testování, je vhodné rozdělit tyto aplikace do určitých logických celků. Těmto celkům říkáme okruhy oblastí jednotlivých aplikací a obvykle se určitý tým vývojářů věnuje vývoji právě v rámci jednoho okruhu aplikací v určité oblasti. Oblast představuje obecnější pojem než okruh. Někdy spadá pod jednu oblast právě jeden okruh aplikací, například pod oblast správy rizik patří jen okruh aplikací pro oddělení řízení rizik, v jiném případě může být pod jednou oblastí více okruhů aplikací. Např. oblast pro podnikovou (korporátní) klientelu. Podstatné je že každý okruh má několik specialistů, kteří aplikace pro ten konkrétní okruh vyvíjejí, ale nebývá výjimkou, pokud testeři testují aplikace pro více okruhů i oblastí.

3.2.1 Hlavní transakční systém

Základní stavební kámen dnešní banky je vždy tzv. **„Core“** systém **hlavní transakční systém** - tento systém umožňuje provádění transakcí (ať se jedná o platbu kartou, zahraniční platbu, převod peněz do jiné měny (pokud toto daná banka umožňuje). Pod pojmem transakce není myšlen jen platební styk, ale jedná se o všechny transakce spojené s produkty banky. Uživatelským rozhraním klientů banky, které je nad tímto systémem, je v současné době nejznámější internetové bankovnictví.

3.2.2 Databáze údajů o klientech CRM

Další velmi důležitou oblastí je **CRM** (**Customer relationship management**) - systém pro řízení vztahů se zákazníky, což v případě banky znamená rozsáhlou aplikaci s údaji všech klientů, jejich rodných čísel, ič, adres, účtů, apod. Tato aplikace slouží pro sledování aktivit zákazníků a na základě vyhodnocení těchto aktivit jim poté banka nabízí další produkty, které vyhodnotí jako vhodné a zajímavé pro banku i klienta. Cílem těchto nabídek je kromě zvýšení zisků také budování vztahu se zákazníkem. Pro CRM je charakteristická návaznost na ostatní systémy – data z CRM se přenášejí do řady dalších systémů a z těchto systémů se zase některá data přenášejí do CRM, je tedy potřeba vysoký stupeň integrace CRM s ostatními aplikacemi a současně efektivní způsob sdílení dat.

3.2.3 Schvalovací aplikace

Banka také poskytuje hypotéky, úvěry a další půjčky. Pro usnadnění rozhodování o tom, zda klient dostane půjčku, jak velkou a s jakým úrokem (ohodnocení rizikovosti klienta) používají zaměstnanci na pobočkách speciální **schvalovací aplikace**, které sbírají data z ostatních systémů a na základě algoritmů z nich vypočtou pomocné informace určující rizikovost klienta, jeho platební morálku. Dívají se i do registru dlužníků (Solus) a insolvenčního rejstříku, což jsou externí databáze, ze kterých si banka automaticky stahuje data.

3.2.4 Řízení rizik – RISK

K minimalizaci rizik nejen při poskytování úvěrů, ale i rozhodování o investicích a dalších aktivitách slouží speciální oddělení **RISK**. Jeho činnost je podporována aplikacemi, které automaticky neustále vyhodnocují rizikovost klientů (fyzických i právnických subjektů) na základě sběru velkého množství dat, příp. i díky spolupráci s ostatními bankami. Tyto aplikace automaticky vyhodnocují, které bankovní produkty mohou být klientům poskytnuty, ale i rizikové události typu firma spolupracující s klientem se dostala do platební neschopnosti a hrozí, že náš klient nedostane proplacenou fakturu apod. Pro uvedenou činnost se často používá pojem **Řízení rizik** - (**Risk management**).

3.2.5 Datový sklad - DWH

Datový sklad (**data warehouse**) je dalším typem skupiny aplikací. Všechna historizovaná data o všech transakcích, zrušených účtech, podvodech, poskytnutých produktech i projektech jsou uložena v datovém skladu. Zpravidla se datový sklad testuje dotazy nad databázemi a kontrolou dat, je však natolik rozsáhlý, že jej lze považovat za další oblast aplikace a jeho správu zajišťuje samostatné oddělení. Bankovní data se důsledně zálohují, což je také významná aplikace.

3.2.6 Obchod s cennými papíry - TREASURY

Další oblastí je **Treasury**, správa investic. Týká se obchodování na finančních trzích, sledování a řízení rizika likvidity, měnového rizika apod. Treasury má své vlastní aplikace s možností získávat data např. z burzovních databází.

Jak je patrné, jednotlivých okruhů je více než jen obecně známé klientské aplikace, a pod jedním okruhem se skrývá řada aplikací[2]. Tyto aplikace vyvíjejí jednotlivé specializované vývojové týmy, z nichž každý má znalost jiné oblasti – není v lidských silách, aby jeden tým zastřešoval všechny potřebné oblasti.

3.3 Proces vývoje aplikací

Každá aplikace v bance má svého vlastníka, což je osoba zodpovědná za správný běh a funkčnost aplikace. Obvykle je vlastník zároveň i zadavatelem úkolů cílených na vývoj aplikace. Vývojem aplikace je myšleno vylepšování, modernizování, příp. upravování aplikace tak, aby odpovídala novým požadavkům oddělení banky, které s aplikací pracuje, novému občanskému zákoníku apod. Zadavatelem bývá většinou osoba mimo útvar IT, není to však nutným pravidlem.

Vývoj probíhá standardně tím způsobem, že zadavatel zadá požadavek, ve kterém specifikuje, co by chtěl v aplikaci vyvinout. Tento požadavek konzultuje s analytikem, který má na starost analýzu dané aplikace a má znalosti o její činnosti z uživatelského i technického hlediska. Analytik projedná s vývojářem, který danou aplikaci vyvíjí, jak náročná by požadovaná změna byla a tuto informaci sdělí zadavateli. Ten po obdržení odhadu finančních nákladů na úpravu aplikace rozhodne, zda se bude změna provádět. Toto rozhodnutí může záviset na mnoha faktorech, někdy je změna nutná, jindy se může zadavatel rozhodnout na základě rozpočtu, který má na vývoj aplikací příslušné oddělení v bance přiděleno.

Pokud se zadavatel rozhodne, že chce požadovanou změnu vyvinout, dostane analytik za úkol napsat analýzu. Během psaní této analýzy, které říkáme technická dokumentace nebo jen dokumentace aplikace, analytik konzultuje řešení se zadavatelem a vývojářem. Po dokončení analýzy (někdy již v průběhu jejího vytváření) vývojář naprogramuje požadovanou funkčnost v aplikaci a přijde na řadu tester, který aplikaci otestuje. Testy ověří, že aplikace odpovídá popisu v analýze a odhaluje v ní chyby, které úpravou vznikly. Tyto chyby tester zadává vývojáři k opravě a případně zajišťuje jejich opravení. Zanalyzovaná,

[2] Např. i pro Core systém mohou existovat speciální aplikace pro verifikaci plateb určité výše.

naprogramovaná a otestovaná aplikace je poté nasazena zadavateli a ten testy ověří, zda aplikace odpovídá jeho požadavkům – zadavatel má roli UAT testera a provádí uživatelské akceptační testování. Po akceptaci aplikace zadavatelem dojde k nasazení této aplikace do produkce, kdy s ní pracují ti zaměstnanci banky, kterým je určená, případně klienti banky – záleží na povaze aplikace.

3.4 Vývojové týmy

Jednotlivé týmy schopné vyvíjet a opravovat aplikace se skládají z vedoucího, který za příslušný tým zodpovídá, a dále z programátorů (vývojářů), kteří tvoří jádro vývojového týmu a bez nichž by nebylo možné aplikace vyvíjet. Vývojové týmy mohou být interní (zaměstnanci banky oddělení IT vývoje) i externí firmy.

Vývoj pouze s programátorem bez realizačního týmu a řádné dokumentace, by však znamenal, že např. s jeho odchodem by odešla i znalost aplikace a to by neslo vysoké riziko, proto další zúčastněné, kteří mají za úkol chování aplikací popsat, a to nejen z uživatelského, ale především i z hlubšího technického pohledu[3]. Podle analýzy, kterou analytik napíše, je možné aplikaci naprogramovat a otestovat. Tester ověřuje kvalitu SW. Podle [5] „úkolem testera je najít chyby, najít je co nejdříve a zajistit jejich nápravu".

Ve skutečnosti zahrnuje role každého z členů týmu více úkolů v rámci procesu vývoje SW než jen jeho základní práce. Programátor kromě vývoje aplikace a podpory analýzy a testingu ještě např. připravuje instalaci a nutné poznámky pro správné nainstalování aplikace administrátory a nastavení parametrů - **Release notes**. Analytik kromě popsání analýzy řeší se zadavatelem možné komplikace při implementaci změn a hledá s ním jiné možné řešení, které je z hlediska IT jednodušší pro implementaci a zároveň bude splňovat požadavky zadavatele. Tester napíše testovací scénáře v rámci **Test analýzy**, realizuje testy a hlásí chyby a po ukončení testování vytváří **Test report**, což je přehled dokončeného testování[4].

[3] Např. datový model, komunikaci s dalšími aplikacemi.

[4] Otestovaných změn, chyb nalezených, opravených i těch co se opravit nestihly, verze a prostředí, na kterém byly provedeny testy apod.

3.5 Vymezení oblasti analýzy – řízení zaměstnanců

Disciplína test managementu zahrnuje širokou škálu činností týkajících se realizace a řízení procesu testování. V této práci se zabývám možnostmi aplikace systému test managementu v bance, a to především v oblasti řízení zaměstnanců. Toto řízení je zaměřeno zejména na sledování klíčových ukazatelů výkonnosti testerů a nastavení vhodných metrik k tomuto testování, finančním hodnocením jednotlivých organizačních uspořádání procesu testování, cenou testování a výpočtem stanovujícím do jaké doby se vyplatí testovat. Tato **analýza není orientována na** psychologické modely řízení zaměstnanců, finanční správu hardware potřebného k procesu testování, nástrojů automatického testování ani formálních postupů pro psaní testových skriptů.

Je to proto, že v praxi pro úspěšnost procesu testování je naprosto klíčová účelná a efektivní komunikace mezi členy týmu realizujícího procesy testování a odstraňování chyb. Tato oblast je také dominantní pro efektivitu a minimalizaci nákladů celého procesu testování, a proto se na ni plně soustřeďuji

4 Organizace a řízení procesu testování v bance

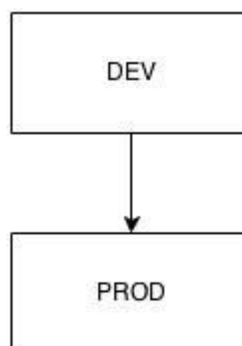
Test management znamená v podstatě veškeré řízení procesu testování v bance. V této práci se zabývám především řízením zaměstnanců, a to jak z pohledu jejich vhodného organizačního uspořádání, tak i z pohledu měření a optimalizace jejich výkonnosti. Tato kapitola bude popisovat stav procesu testování v bance tak i organizační uspořádání v etapách historického, původního a nového stavu.

4.1 Historický stav testování

K přiblížení stávajícího stavu testingu v naší bance je třeba pohled do historie. Když vývoj v bance začínal, žádný testing oficiálně neexistoval, testování prováděli zaměstnanci IT nebo jiných oddělení jako jeden z mnoha úkolů své práce. U některých aplikací neexistovalo ani testovací prostředí a tyto vyvinuté aplikace se buď netestovaly vůbec, nebo se testovaly rovnou na produkci. Jádro systému, které provádělo transakce a další aplikace obdobné důležitosti, své testovací prostředí měly.

Testy se tedy většinou prováděly buď jednou, nebo vůbec. Jen ve výjimečných případech provádělo IT SIT testy a poté ještě vlastník aplikace UAT testy.

Většinou probíhalo nasazení aplikace tím způsobem, že z vývojového prostředí šla posléze rovnou do produkce:



Obr. 4 původní model nasazování aplikace [autor]

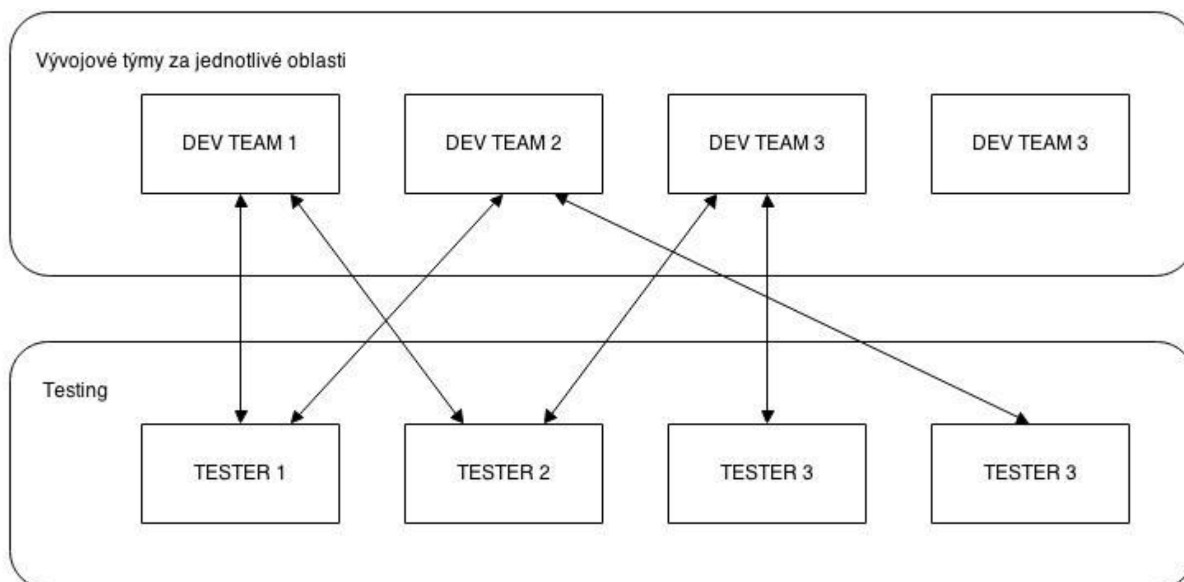
Příklad situace, kdy se aplikace testovala až v produkci, dokumentuje konkrétní schvalovací aplikace[5] . Tato schvalovací aplikace byla často předělávána, probíhaly na ní drobné úpravy i přidávání dalších funkcionalit. Běžný klient se s ní nesetkal, sloužila jako pomocník zaměstnancům banky, tedy z pohledu IT byla tato aplikace určena pouze pro interní zákazníky a ti si ji také testovali.

Pokud v této aplikaci nebylo možné danou funkcionalitu otestovat na prostředí DEV, testoval ji programátor ve spolupráci s interním zákazníkem až po nasazení na **produkci**, zde označenou jako prostředí **PROD**. Protože bylo potřeba, aby následující den aplikace fungovala, mnohokrát probíhalo toto testování dlouho do noci, bylo totiž nutné odladit všechny kritické chyby, které by mohly způsobit nefunkčnost aplikace následující den.

4.2 Původní stav

Další fází bylo zavedení oficiálního testovacího oddělení. Toto oddělení mělo svého vedoucího, 3 test koordinátory a 5 testerů. Jednalo se o sjednocené testovací oddělení pro celou banku, testeři v něm testovali napříč různými projekty, koordinátoři organizovali UAT testy které prováděl vlastník aplikace, zatímco testeři byli organizováni zejména manažery dodávky jednotlivých produktových týmů, kteří řídili proces dodávek aplikací.

[5] Pro schvalování hypoték, úvěrů a ohodnocení platební morálky klienta



Obr. 5 testing po zavedení testovacího oddělení [autor]

Pod DEV TEAM 1 si můžeme představit například vývojový tým pro aplikace z oblasti risku, DEV TEAM 2 např. pro oblast transakčního systému. Tyto týmy obsahovaly 3-6 programátorů, z nichž jeden byl hlavní – vedoucí vývojového týmu. Vývojovým týmům pak byli přidělováni testeři a analytici podle potřeby, stejně jako manažeři dodávky, kteří měli na starost řízení dodávek a alokaci kapacit těchto týmů.

Jak ukazuje Obr.5, nastávala situace, při které testoval jeden tester pro několik produktových týmů současně. To mělo za následek řadu komplikací, každý tým měl svá vlastní interní pravidla. Ačkoliv platila nějaká obecná pravidla pro celé IT, vždy se v drobných detailech tato pravidla přizpůsobila konkrétnímu týmu[6].

[6] Jeden tým zapisoval bugy v nástroji A, druhý zase v nástroji B, jeden tým požadoval kompletní popis chyby i s výpisem logu, druhý zase co nejkratší a nejstručnější, používaly se jiné nástroje pro sdílení souborů s aktuální verzí aplikace atd.

Pro příslušného testera bylo nutné se často přeorientovávat mezi jednotlivými projekty, které se mezi sebou výrazně lišily. Každý produktový tým měl zpravidla i jiného analytika, který dodržoval vlastní systém psaní analýzy[7], neboť analytici se křížili mezi projekty podobně jako testeři.

Z hlediska lokace seděl testing pohromadě, což přinášelo jisté výhody i nevýhody. V případě řešení důležitých záležitostí bylo pro testera nutné se přesunout k danému vývojovému týmu kvůli důležitosti osobního kontaktu[8], čímž ztrácel čas. Pokud se tester rozhodl vývojáře navštívit na jejich místě, ne vždy tam byl vývojový tým přítomen (porady týmu). Obdobná situace nastala i v případě potřeby programátora něco neodkladného řešit s testerem. Nepřítomnost testera v týmu měla také za důsledek informační šum, spousta věcí, které si tým domluvil ad-hoc se zapomněla testerovi komunikovat a kvůli tomu často vznikala zbytečná práce[9]. Sjednocené testování měl však i své výhody, dodržování testovací metodiky bylo jednotnější, testeři měli navzájem rychlou zpětnou vazbu, výměnu testovacích analýz, technik a v případě že se křížily i aplikace[10] docházelo ke sdílení znalosti testovaných aplikací.

Původní stav vs. Nový stav

Výhody sjednoceného testování oproti testování v produktových týmech:

- Sjednocená testovací metodika a její jednotná implementace
- Tester zná širší okruh aplikací napříč oblastmi
- Sdílení znalostí v rámci testingu (metody, techniky, aplikace)

---

[7] Někde byl datový model součástí analýzy, jinde nikoliv apod.
[8] Komunikace emailem apod. trvala řádově hodiny až dny

[9] Např. tester 20 minut řešil, proč mu něco nefunguje, přičemž si vývojový tým domluvil, že dočasně vypne systém, prostředí atd. a zapomněl dát testerovi vědět.

[10] Např. podle Obr. TESTER1 i TESTER2 testovali aplikace pro vývojový tým DEV1

- Vyšší vzájemná zastupitelnost testerů[11]

- Vyšší flexibilita při plánování kapacit testingu na projekty a vývojové týmy

- Vedoucí testingu má všechny testery pohromadě a díky tomu i přímý dohled nad jejich činností

Nevýhody sjednoceného testování oproti testování v produktových týmech:

- Nižší úroveň znalosti testovaných aplikací

- Informační šum až úplná ztráta informací od vývojového týmu

- Nutnost neustálého fyzického přesouvání mezi testingem a vývojovým týmem

- Nejednotná metodika analýz u jednotlivých týmů

- Neefektivita při práci na více projektech v rámci několika vývojových týmů

- Manažeři dodávky a členové vývojových týmů nemají přehled o tom, co právě příslušný tester dělá

- Různé systémy pro správu verzí aplikací u různých týmů

- Režijní činnosti z důvodu odlišných systémů a interních pravidel testerům zabírají mnoho času (demotivace i vytížení kapacit)

[11] Pokud mají oba testeři znalost dané aplikace a zároveň jsou nealokováni

- Tester oddělený od vývojového týmu je často vnímán jako cizí osoba a komplikace než jako člen týmu a podpora vývoje (záleží na osobnosti vývojářů)

- Tester je více oddělený od zdrojů změn v aplikaci – nižší proniknutí do oblastí vývoje i analýzy, menší příležitost ovlivnit analýzu v době jejího vzniku

- Nutnost přizpůsobovat způsob testování i reportování chyb jednotlivým týmům a neustále se v odlišných potřebách a pravidlech orientovat

- Projektové schůze u několika vývojových týmů se mohou krýt a tester tak může přijít o některé klíčové porady

- Přetíženost testerů při nízkém objemu otestovaných funkčností pokud je tester nucen se často přepínat mezi jednotlivými projekty[12]

Jak ukazuje popis výhod a nevýhod testerů v jednotném testovacím týmu, produktové týmy s vlastním testerem se zdají být vhodnější variantou než oddělený testing. Jde zde ovšem i o směr metodiky vývoje software – v současné době jsou hlavním světovým trendem agilní metodiky [3]. U naší banky rovněž dochází postupně k přechodu od vodopádu k agilnímu vývoji. Znaky agilního vývoje jsou mimo jiné menší týmy, v nichž všichni členové ovlivňují změny v aplikacích (vyjadřují se k nim – v případě testerů se jedná o včasné podchytávání rizik vhodnou analýzou rozporů v zadání s technickými možnostmi a stávající logikou aplikace).

Tento stávající model, kdy jsou samotné vývojové týmy tvořeny pouze vývojáři, však spadá k vodopádu i v oblasti analýzy – příklad: analytik přidělený týmu A dodělá analýzu A a poté přejde k týmu B. Mezitím tým A aplikaci podle analýzy A a poté se k týmu A přidělí tester, který aplikaci A otestuje. Tento tester nalezne zásadní rozpory v analýze, rozdíly v analýze a aplikaci a další nesrovnalosti a požaduje spolupráci analytika k vyjádření případně opravu analýzy. Tento analytik však již dávno analyzuje pro tým B a analýzu A neviděl několik měsíců, trvá mu dlouho, než si rozpomene na všechny detaily a souvislosti, některé věci

[12] Přepínání testovaných aplikací, reportovacích nástrojů, databází, analýz.

opomene apod. – jedná se o typický vodopádový model, kdy se po dokončení analýzy již v pozdějších fázích složitě něco mění. Oproti tomu agilní vývoj podporuje flexibilitu vývoje a časté změny v aplikaci, tedy i pro něj jsou vhodnější vývojové (produktové)[13] týmy, kde je kromě vývojářů permanentně přítomen jak tester, tak i analytik[14]. Tato skutečnost spolu s výhodami testerů jako součástí produktových týmů způsobila transformaci testingu na nový stav, kdy jsou FAT testeři inherentní součástí produktových týmů, zatímco samostatný testing zajišťuje SIT testy případně koordinaci UAT testů.

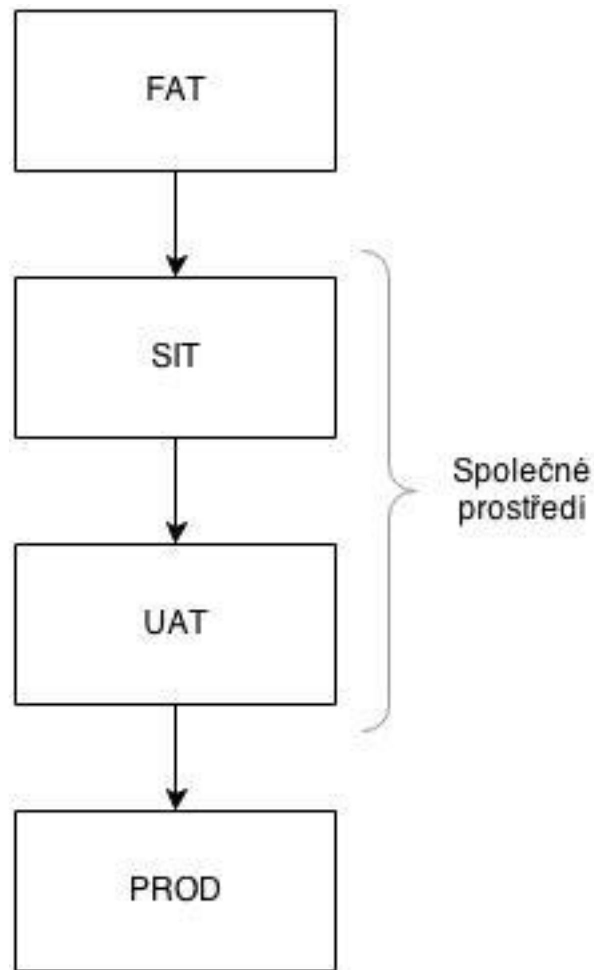
4.3 Nový stav

Nová organizace testingu má za úkol zlepšit kvalitu bankovního software napříč všemi oblastmi. Došlo ke zvýšení počtu testerů, zkušení testeři z původního sjednoceného testování se stali součástí jednotlivých produktových týmů. V těchto týmech přibyli i analytici, kteří do nich byli zakomponováni stejně jako testeři. Vznikli tedy FAT testeři mající za úkol odhalit co nejvíc chyb v co nejmladší fázi vývoje. Tito testeři netestují vnější integraci aplikací do dalších systémů.

Samotné oddělení testingu má nyní na starost především SIT testy, kdy dochází zejména k testování integrace napříč systémy. SIT testeři také testují to, co FAT testeři nestihli nebo nemohli otestovat (např. z důvodu nefunkčního prostředí). Ideálně by probíhaly standardní FAT testy a poté SIT a UAT testy, jako to ukazuje Obr.6, kde je vidět že SIT testeři a UAT (zpravidla business vlastníci aplikací) mají dohromady jedno společné testovací prostředí, v praxi však reálně existují funkcionality, které stačí otestovat FATově protože do žádného jiného systému nezasahují, stejně tak jako SITové testy které není třeba (příp. ani možné) testovat FATově. UAT testy by pak měly probíhat vždy a při nich zadavatel rozhodne o akceptaci.

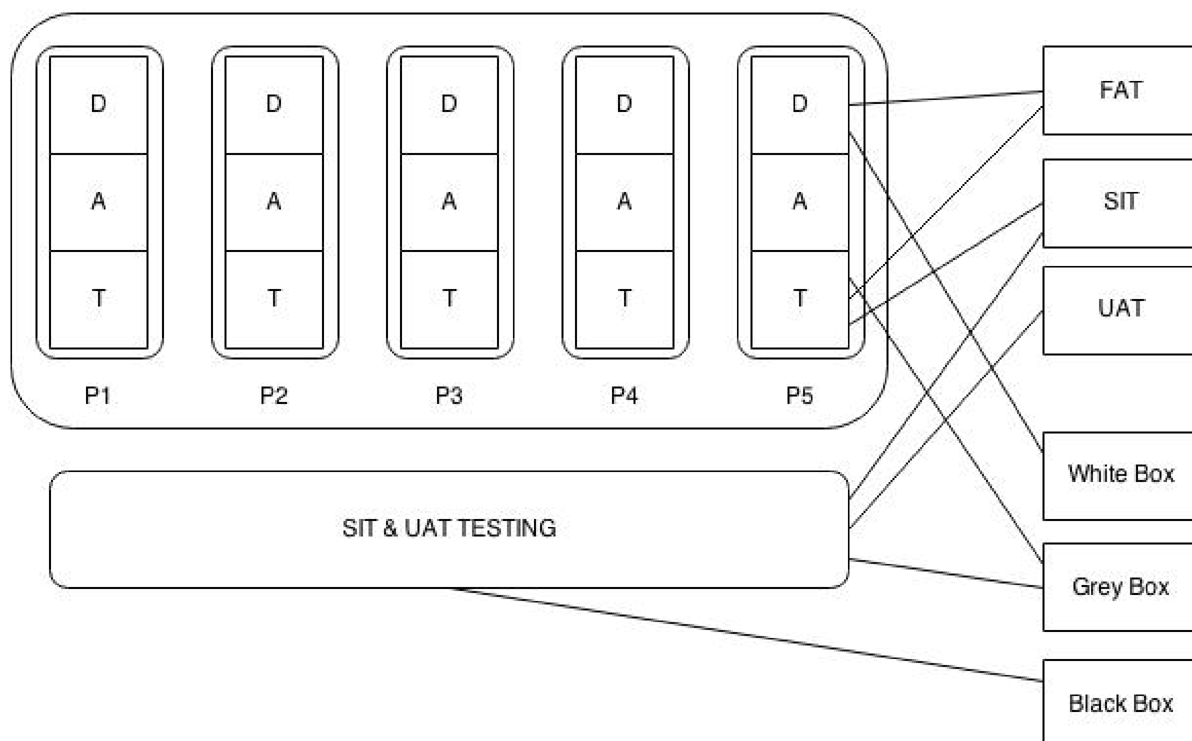
[13] Vývojovými i produktovými týmy se myslí jedno a totéž, jde o týmy vyvíjející aplikace pro konkrétní oblast. Název vývojový tým vzbuzuje pocit, že jsou součástí pouze vývojáři, produktový tým pak navazuje pocit, že výstupem je určitý produkt – to je však poněkud zavádějící, v každém případě je výstupem produkt, který je kromě vývoje většinou i analyzován a testován.

[14] Analytiků a testerů může být v rámci jednoho vývojového (produktového) týmu libovolný počet, zpravidla však bývá obecně nejvíce vývojářů.



Obr. 6 postupné nasazování aplikace v novém testingu zaměřeném na agilní vývoj [autor]

Podrobněji popsaný stávající stav ukazuje Obr.7. Zde je zobrazeno, které typy testů provádějí jednotlivé skupiny v rámci produktového týmu (vývoj, analýza nebo test) a které typy testů provádí samotné oddělení testingu zejména pro vnější integrační testy a uživatelské akceptační testy (SIT a UAT).



P1,….P5 – jednotlivé produktové týmy

D – Development (vývoj)

A – Analysis (analýza)

T – Testing

Obr. 7 Typy testů v novém testingu zaměřeném na agilní vývoj [autor]

Vývojář provádí zejména FAT testy typu White Box, v ideálním případě kompletní unit testy. Analytik zpravidla neprovádí žádné testování. Tester v produktovém týmu má na starost kompletní FAT testy a vnitřní SIT testy – testuje vnitřní integraci v rámci aplikace, nikoliv komunikaci s ostatními aplikacemi[15]. FAT testy provádí stylem Greybox, netestuje přímo zdrojový kód, ale modifikuje data v databázi pomocí SQL a PLSQL.

[15] FATový tester nasimuluje data která aplikace importuje nebo exportuje do jiných aplikací a tím částečně otestuje i vnější integraci, ale pouze na úrovni očekávaného vstupu/výstupu dat – kompletní testování vnější integrace provádí až SITový tester který má k dispozici funkční rozhraní, pomocí kterého aplikace komunikují a ta data mezi nimi při testu skutečně pošle a uvidí prostup z jedné aplikace do druhé.

Samotný testing provádí velmi často Black box, případně i grey box – modifikace z dat je zde však v menší míře než u FATového testera v produktovém týmu, systémové integrační testy jsou více zaměřeny na funkcionalitu a komunikaci aplikací mezi sebou pomocí rozhraní. Dochází zde k SITovým testům vnitřní i vnější integrace a také uživatelským akceptačním testům – v ideálním případě by byla k dispozici pro oba typy testů různá testovací prostředí, reálně však SITy i UATy probíhají na stejném prostředí v posloupnosti nejdříve SIT, poté UAT. Pokud zadavatel usoudí že SITové testování je postradatelné/zbytečné, je možné ho plně vynechat a provést pouze UAT testy – ty by však měly být provedeny vždy, jimi zadavatel na základě jejich výsledku akceptuje nebo odmítne implementaci požadavku. Je na individuální dohodě zda UAT testy provede testing a dodá zadavateli výstupy z těchto testů, nebo zda je celé provádí sám zadavatel. UAT testy jsou výhradně typu Black Box.

Výhody integrace testerů a analytiků do produktových týmů se v plné míře projeví až po delší době, integrace FATových testerů a jejich plné pokrytí aplikací daného produktového týmu trvá i několik let[16].

Jedním ze smyslů zlepšení pokrytí testovaných aplikací je i ulehčení testování businessu. Pokud neprobíhaly FAT ani SIT testy, musel business zajišťovat kompletní testování sám v rámci UAT testů, což podle nejnovějších zjištění není správný postup, UAT testy by měly sloužit pouze k akceptačnímu testování konečným uživatelem (zadavatelem, v našem případě vnitřním zákazníkem) a nikoliv ke kompletnímu otestování všech kombinací, které business zdržují od práce.

[16] Pokud vůbec k plnému pokrytí testování dojde – příkladem může být tým kde na 4 vývojáře a 2 analytiky připadá 1 tester, nabízí se otázka zda-li je v takovém případě vůbec reálné aby jediný tester pokryl takový objem aplikací

5 Finanční zhodnocení variant organizace testování

Testování přináší informaci o tom, v jakém je daný produkt (aplikace) stavu, kolik chyb přibližně obsahuje a jak je použitelná pro běžný provoz[17]. Hodnotit testing z hlediska finančního přínosu je velmi zavádějící, protože **disciplína softwarového testování sama o sobě nepřináší přímý finanční užitek**, jak potvrzuje mnoho autorů (např. [4],[5]).

Cena této informace je v podstatě cenou za testing, s tím, že kvalitní testování přispívá ke kvalitě tím, že tester se i snaží o zajištění odstranění chyb. **Včasné testy a odhalování chyb snižují cenu opravy chyby, čímž testing rovněž přispívá ke snížení finančních nákladů.**

Stěžejním problémem je určení ceny chyby – v praxi se většinou manažeři nezabývají tím, kolik banku stojí to, že je v produkci chyba a vnímají pouze tento stav jako fakt stejně jako nutné náklady k jejímu odstranění. Prakticky vždy je odstranění chyby tím dražší, čím později se chyba nalezne, v produkci je nejdražší, navíc je třeba započítat náklady, které tato chyba způsobí. Tyto náklady se velice liší aplikaci od aplikace a případ od případu, typickým příkladem může být překlep. Na první pohled je překlep chyba o nízké prioritě, která nebude mít významný dopad. Toto však platí pouze tehdy, pokud se jedná o překlep v aplikaci pro zaměstnance banky, se kterou klient nepřichází do styku, který navíc nezpůsobí záměnu příp. ztrátu dat. Překlep v nápovědě internetového bankovnictví je vizitkou kvality příslušné banky, způsobuje reputační ztráty. Překlepy v samotné aplikaci určené pro klienty pak působí velmi neprofesionálně a určitou část klientů od banky odrazují. Překlepy ve znění smluv, způsobující zmatení pojmů a tím pádem nepřesnost smluv, které předpřipravuje bankovní aplikace, mohou mít závažné finanční dopady.

Ani u kritických chyb není snadné určit jednoznačně všechny náklady. Kritickou chybou se myslí např. závažná chyba v aplikaci, která způsobí její dočasnou nedostupnost. Tato

[17] Tato informace je omezená testovacím prostředím a dalšími vlivy a faktory, jedná se o přibližný přehled – v závislosti na tom, jak moc testovací prostředí dokáže simulovat reálné běžné prostředí provozu, reálnou zátěž, data atd.

nedostupnost má obvykle přímé i nepřímé náklady. Nedostupnost v aplikaci pro faktoring[18] způsobí jednoznačné a jasné finanční ztráty, které jsou vidět v tom, kolik smluv bylo z důvodu nedostupnosti porušeno, o kolik možných kontraktů banka přišla apod. Vedle toho existují i nepřímé náklady ve zhoršení pověsti banky, pohledu klientů i dodavatelů na vzniklou situaci apod. Nedostupnost v systému včasného varování může a nemusí způsobit finanční ztráty, a pokud způsobí, jejich velikost je extrémně proměnná v závislosti na tom, jak moc rizikový subjekt byl touto nedostupností neodhalen a v jakém finančním objemu jsou škody, které tato skutečnost nakonec způsobila.

Testování je proces, jehož důležitou součástí je řízení a vyhodnocování rizik [15] a chybou vzniká riziko. Nikdy není zaručené, že by se testováním na chybu přišlo, jde pouze o vyhodnocení pravděpodobnosti. Kvalitně řízeným a organizovaným testingem lze maximalizovat pokrytí testovaných aplikací a minimalizovat počet chyb v těchto aplikacích. Stejně tak není jisté zda-li chyba nějaké škody v produkci způsobí, a pokud ano, jak velké tyto škody budou – je možné že se chyba i v produkci odhalí dříve, než způsobí ztráty. Některé chyby zůstanou v aplikaci po celou dobu jejího používání a nikdy se na ně nepřijde.

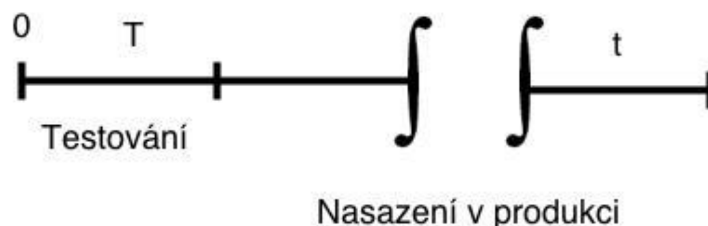
Finančně vyhodnotit přínos testingu je velmi obtížný úkol. Náklady se dají určit celkem jednoznačně, s výnosy je to složitější – přesto lze s určitou pravděpodobností vypočítat, zda se testování ještě vyplatí nebo je finančně výhodnější testování ukončit. Vzhledem k povaze a charakteru vzniku a důsledků chyb, jak bylo popsáno výše, nelze stanovit výnosy z testingu tak jednoznačně, jako náklady. Pokud však odhadneme kolik chyb objevíme testováním za nějaký časový interval a zároveň i cenu těchto chyb, lze s určitou spolehlivostí vypočítat, jestli se nám testování vyplatí.

5.1 Stanovení vnitřní ceny testování

Vycházíme-li z předpokladu, že výchozí čas od zahájení testování bude 0 a od této doby po dobu T bude software testován, poté se testování dokončí a po čas t bude software (bankovní aplikace) nasazen v produkci a používán, ilustruje tuto situaci Obr.8. Typicky je čas t, po

[18] Faktoring je služba, kterou banka poskytuje právnickým osobám, kdy spravuje financování krátkodobých úvěrů poskytnutých při dodávkách zboží a služeb a přenáší na sebe část rizika nezaplacení od dodavatele za určitý poplatek od klienta, kterému služby faktoringu banka poskytuje.

který je aplikace používána, mnohonásobně delší než čas T, po který je vyvíjena a testována[19]. Po doběhnutí času t je aplikace zcela vyřazena a nahrazena jinou, příp. jiným systémem nebo procesem.



Obr. 8 Časová škála běhu aplikace [6]

V prostředí bankovních aplikací existuje několik základních předpokladů, které musí platit pro použitelnost modelu pro výpočet ceny testování SW. Než si tyto předpoklady uvedeme, je potřeba jasně vydefinovat co bude v této kapitole myšleno cenou opravy chyby.

Cena odstranění chyby nalezené během testování (c_1) zahrnuje sumu všech zdrojů použitých k odstranění chyby, ladění aplikace a také náklady obětované příležitosti nevyužití těchto zdrojů jinde, např. kapacit testera, analytika a vývojáře.

Cena odstranění chyby nalezené na produkci po nasazení aplikace (c_2) zahrnuje tytéž náklady jako c_1, navíc je však třeba počítat se ztrátou nebo omezením způsobeném nedostupností aplikace, poškozením a tím i nutnou opravou dat, ztrátu reputace a další vlivy, které tato chyba způsobila, než byla odhalena, včetně přímých finančních nákladů. Ve skutečnosti jsou velké rozdíly jak v cenách opravy chyb, tak i v poměru velikostí c_1 a c_2 – není výjimkou, když je tento poměr i desetinásobek a víc.

Tento model počítá navíc s tím, že chyba je zde uvažována jako jedna entita, pro zjednodušení se nebere v potaz závažnost a typ chyby.

[19] V tomto modelu se omezujeme na disciplínu testování, avšak statické testování analýzy a jednotkové testy mohou probíhat v době, kdy ještě není aplikace vyvinuta ani z části.

5.1.1 Předpoklady pro použitelnost modelu pro výpočet vnitřní ceny testování

- **Cena odstranění chyby zjištěné na produkci je vždy vyšší než cena odstranění této chyby, pokud by byla nalezena a odstraněna již ve fázi testování**. Platí

$$c_2 > c_1$$

- **Chyby do aplikace zavlečené mají povahu stochastického procesu**, kde m(t) je očekávaný počet chyb v čase t. Potom **platí, že četnost chyb v čase t, kterou označíme z(t), je derivací m(t).**

- **z(t) je monotónní klesající spojitá funkce** - tento předpoklad je možné pro použitý model opustit, ve skutečnosti však v drtivé většině případů platí – typicky čím více se aplikace testuje, tím méně je v ní nalezeno chyb, a to i v případě zavlečení nových chyb opravou chyb starých. Naopak hned po prvním nasazení je počet nalezených chyb nejvyšší, a protože z(t) hledáme na časovém intervalu T, nikoliv v bodě, je funkce z(t) skutečně monotónní, klesající a spojitá.

Pokud jsou výše popsané podmínky splněné, je možné popsat model sloužící k výpočtu ceny testování, který počítá i s časovou hodnotou peněz, což je vhodné pro dlouhodobé, roky trvající projekty.

Takový model výpočtu ceny testování zformuloval Robert L. Vienneau [6] :

$$C_t(T) = \int_0^T [c_1 z(x) + c_3] e^{-(1+r)x} \, dx$$

Kde:

T je datum nasazení software do produkce (ukončení testování a předávka aplikace)

C(T) je celková cena testování, pokud je software nasazen v čase X

C_1 je cena za odstranění chyby nalezené během procesu testování

C_3 je cena testování za jednotkový čas

Z(x) je četnost chyb v čase x nebo-li četnost výskytu chyb v čase x. Očekávaný počet chyb v malém časovém intervalu (x, x + dx) je z(x)dx.

$\mathbf{e}^{\mathbf{-(1+r)x}}$ počítá s časovou cenou peněz kde r představuje tzv. cenu ušlé příležitosti (ušlý výnos se zohledněním rizika), tedy současná hodnota y korun utracených v čase x je rovna $y * e^{-(1+r)x}$

Jediná proměnná, o které nebylo více zmíněno, je c_3. Cena testování za jednotkový čas zahrnuje cenu zdrojů použitých během testování (mzdové náklady testerů + vytížení ostatních členů týmu), cenu ušlé příležitosti nevyužitím těchto zdrojů jinde a také náklady na nenasazení systému. Těmito náklady je myšlen ušlý zisk, který by banka získala dřívějším nasazením aplikace, příp. snížené výdaje dřívějším nasazením apod.

C_t(T) je tedy současná hodnota ceny testování, pokud je SW nasazen v čase T.

Pro kratší projekty, kterých je většina, je možno ze vzorce odstranit část počítající časovou hodnotu peněz $e^{-(1+r)x}$. Pro dlouhodobé projekty, které probíhají několik let, je však použití této části vzorce významným zpřesněním výpočtu.

C_t(T) roste přímo úměrně s dobou, po jakou je aplikace testována. Označme nyní **C_o(T)** jako současnou hodnotu ceny operací, které jsou s aplikací prováděny na produkci po jejím

nasazení do provozu. Pod pojmem operace se v tomto případě neskrývají běžné operace prováděné s aplikací uživatelem, ale operace týkající se odstraňování chyb z produkce.

Podle [6] platí pro $C_o(t)$ vzorec:

$$C_o(T) = \int_T^t c_2\, z(x)\, e^{-(1+r)x}\, dx$$

kde počítáme s časovým intervalem od nasazení do produkce a ukončení testování T po samotné ukončení užívání aplikace v čase t.

Jak již bylo zmíněno výše, c_2 značí cenu odstranění chyby během operací včetně všech dalších nákladů které tato chyba způsobila. I zde je třeba odhadnout množství chyb v určitém čase.

Platí, že celková cena systému je součtem těchto dvou cen :

$$C(T) = C_t(T) + C_o(T)$$

Na jedné straně máme nízké náklady na opravy chyb ve fázi testování, jednotkovou cenu za vykonávání testů a cenu nenasazení aplikace v produkci, na druhé straně máme ve fázi produkce velmi vysoké náklady za každou chybu, která se objeví. Hlavním úkolem je zjistit, jaké je optimální časové rozmezí po které aplikaci testovat, tedy doba T, tak, abychom minimalizovali náklady, tedy aby C(T) byla minimální. Jinými slovy, jak dlouho se vyplatí aplikaci testovat a kdy už je výhodnější ji nasadit do produkce.

Označme optimální dobu testování jako T_o. Podle [6] T_o algebraickými úpravami nalezneme jako

$$(c_2 - c_1) * z\,(T_o) = c_3$$

Protože $c_2 - c_1$ reprezentuje v podstatě peníze ušetřené odstraněním chyby během testu místo produkce, levá strana představuje sumu financí, které získáme testováním za mezní dobu.

Oproti tomu c_3 znamená cenu testování za jednotku času, pravá strana tudíž představuje mezní náklady způsobené testováním za dané časové období. Protože jak mezní náklady, tak i mezní výnosy nastanou ve stejný časový okamžik T_o, efekt časové hodnoty peněz je vyrušen – ve vzorci pro výpočet T_o tuto hodnotu nenajdeme.

Z toho vyplývá, že pravidlo pro minimalizaci nákladů je testovat dokud je mezní výnos z odstranění chyby během testování namísto produkce vyšší než mezní náklady na testování samotné.

5.1.2 Podmínky pro nalezení minimální ceny systému

Pro nalezení minimální ceny je třeba více podmínek. V praxi nastanou zpravidla automaticky, je však třeba je matematicky popsat. Aby existovalo řešení, čas strávený testováním nesmí být záporný. Stejně tak tento čas nesmí přesáhnout dobu do konce používání aplikace v produkci. T_o splňuje tato kritéria proveditelnosti právě tehdy, když platí:

$$z\,(t) \leq c_3 \,/\, (c_2 - c_1) \leq z\,(0)$$

Pokud je tato podmínka porušena, nastane jeden ze dvou případů:

1. Aplikaci je výhodnější nasadit do produkce bez jakéhokoliv testování.

2. Aplikaci je výhodnější raději nikdy nenasazovat do produkce.

Pokud nastane situace, že při zahájení testování přesáhnou náklady na testování možné výnosy z odstranění chyb během testování oproti produkci, je lepší testování v tomto případě vůbec neprovádět a aplikaci nasadit rovnou. Takové případy v praxi skutečně nastávají. Pokud se změnou kódu na 99 % nezavlekla chyba a zároveň platí, že pokud by se takováto chyba zavlekla, tester by ji neměl reálně šanci odhalit, pak je vhodné nasadit aplikaci do produkce rovnou[20]. Předpokládá to však zběžnou kontrolu běhu aplikace vývojářem.

Nastane-li situace, že snížená ztráta pomocí odstranění chyb během testování bude významnější než náklady na testování a to až do doby kdy se aplikace zcela přestane používat, neměla by být nikdy nasazena. Reálně může být aplikace v tomto případě nasazena i v případě vědomí všech zainteresovaných stran (vývoj, testing, zadavatel atd.) o tom, jaké to bude mít důsledky. Důvodem nasazení může být nutnost kvůli legislativní změně nebo návaznost jiného systému banky na funkčnost této aplikace, bez které by nefungoval. V takovém případě by se však měla i tato ztráta zahrnout do nákladů c_3, protože vzniká nenasazením aplikace do produkce a je tedy současně i cenou testování za jednotkový čas.

5.2 Příklad aplikace výpočtu vnitřní ceny testování

Pokud máme k dispozici dostatečné množství informací o testovaném produktu**, je možné v rámci optimalizace procesu testování určit, jak dlouho se vyplatí aplikaci testovat.** V praxi není možné v prostředí banky nasazovat aplikace do produkce kdykoliv. Nasazování probíhá v případě naší banky čtvrtletně - jindy než na konci každého čtvrtletí není možné aplikaci nasadit. Pokud má právě jeden tester na starost testování aplikace A, pak – s dostatečným množstvím informací o této aplikaci – je možné vypočítat, na jak dlouho se

[20] Případně do UAT, ale v tomto modelu se pohybujeme pouze v dimenzích testování a běhu aplikace v produkci.

vyplatí přidělit tomuto testerovi úkol, aby aplikaci A testoval. Tento výpočet si ukážeme na následujícím příkladě.

5.2.1 Údaje k příkladu

Zadavatel odhadl, že nenasazení nové funkčnosti v aplikaci A stojí banku měsíčně 100 000 Kč. Bez nasazení této funkčnosti nemůže banka poskytovat smlouvy, které by právě tuto částku každý měsíc vydělaly oproti stávajícímu stavu. Reputační riziko zde zadavatel vyhodnotil jako zcela zanedbatelné.

Banka tuto funkčnost v aplikaci A vyvinula a tuto aplikaci umí testovat pouze tester T, žádný jiný tester nemá potřebnou znalost dané aplikace a nejsou k dispozici alokace kapacit testerů na to, aby bylo možné jiného testera pro tuto aplikaci zaškolit. Tester T stojí měsíčně banku 80 000 Kč. Tato cena zahrnuje přímé i nepřímé náklady včetně času ostatních účastníků procesu testování, který tester T potřebuje jako podporu testování. Kdykoliv něco tento tester testuje, je nutné oficiálně vykázat jeho alokovanou kapacitu na toto testování a činnost testování později řádně finančně zařadit. V současné době by tester T mohl připravovat testovací scénáře na jiné aplikace, které bude v budoucnu testovat, není to však příliš akutní – cenu ušlé příležitosti toho, že bychom testera v současné době alokovali na jinou činnost než psaní těchto scénářů, odhadneme pouze na 5 000 Kč.

Od zahájení testování uběhne právě 1 měsíc do doby, kdy bude možné aplikaci nasadit. Po této době bude možné aplikaci nasadit až za čtvrt roku a pak vždy po čtvrt roce. Rozhodujeme se tedy, kdy bude nejvýhodnější aplikaci nasadit. Pokud budeme uvažovat za časovou jednotku $t = 1$ měsíc a časové období $Q = 3t$ (kvartál), pak je možné aplikaci nasadit v čase $t + n*Q$, kde n je nezáporné celé číslo.

V procesu testování u aplikace A platí, že SIT testování není potřeba a testování UAT bude probíhat souběžně s FAT testováním poslední týden před nasazením, protože zadavatel, který je zároveň i UAT testerem, má v testerovi T velkou důvěru a UAT testování tak bude jen formalita. Díky těmto skutečnostem můžeme uvažovat pouze FAT testování, na které se omezíme a které provádí právě tester T.

Průměrná cena za odstranění chyby v průběhu testování je 400 Kč. Tato cena zahrnuje čas všech účastníků procesu vývoje SW, kteří se na odstraňování chyby podílí – tester (zápis a eskalace chyby) , analytik (konzultace s testerem zda se skutečně jedná o chybu) a vývojář, který touto opravou stráví v průměru několik minut.

Průměrná cena za odstranění chyby, která se vyskytne důsledkem aktuálních úprav v aplikaci A v produkci, je na základě minulých zkušeností rovna 40 000 Kč. Jedná se nejen o přímé náklady na samotné odstranění chyby, ale i o vyčíslení všech škod, které tato chyba způsobí[21].

Díky tomu, že aplikace A byla v minulosti již mnohokrát upravována, může tester T zkušeně odhadnout, kolik v ní dokáže odhalit chyb za určitý čas testování. Tester navíc odhadne, kolik by potřeboval času na důkladné protestování aplikace k tomu, aby nalezl maximum chyb. Ze zkušeností víme, že pokud tester aplikaci otestuje důkladně a nalezne maximální množství chyb M, na produkci se poté vyskytne zhruba 10% z M. Kdybychom mu dali jen 25% času, který by potřeboval na důkladné otestování, pak se po nasazení na produkci vyskytne přibližně 30% z M a při testování tester odhalí jen 60 % z M. Tester po konzultaci s vývojářem odhadl M = 30 a čas, za který toto množství chyb odhalí, stanovil na 4 kalendářní měsíce.

Protože aplikaci lze nasadit v čase t + n * Q, prozkoumáme nejprve varianty n = 0 a n = 1, nasazení po měsíci testování a nasazení po čtyřech měsících testování. Označme tyto varianty jako a) pro n = 0 a b) pro n = 1. V případě, že bude výhodnější nasadit aplikaci po měsíci testování, již nemá smysl se zabývat dalšími variantami. Pokud vyjde, že bude výhodnější nasadit aplikaci po 4 měsících testování, můžeme zkoumat případy, kdy se bude n nadále zvyšovat, ale pouze za předpokladu, že ani po 4 měsících testování nebude naplněno M. V případě naplnění M již nemá smysl aplikaci dále testovat, neboť tam tester žádné další chyby neodhalí. Protože zadavatel má za úkol v rámci optimalizace procesů minimalizovat náklady, akceptuje nasazení v tom čase, kdy to bude nejlevnější.

[21] Toto vyčíslení provedl zadavatel ve spolupráci s vedoucím oddělení vývoje SW.

5.2.2 Společné podmínky pro obě varianty

Pro varianty a) i b) platí, že hledáme co nejnižší cenu C(T), kde $C(T) = C_t(T) + C_o(T)$. Omezení dodávky aplikace pro nás znamená, že C(T) nehledáme na libovolném intervalu t, ale zvlášť pro variantu a) a zvlášť pro variantu b) kde každá z těchto variant má daný časový interval.

Protože testování nebude trvat příliš dlouho, neboť se jedná jen o drobný vývoj a nikoliv o dlouhodobý projekt, můžeme zanedbat časovou cenu peněz.

Pro stanovení $C_o(T)$ platí tedy:

$$C_o(T) = \int_T^t c_2 \, z(x) \, dx$$

kde $c_2 = 40\,000$

Pro $C_t(T)$ platí vztah:

$$C_t(T) = \int_0^T [c_1 \, z(x) + c_3] \, dx$$

Kde $c_1 = 400$

$c_3 = 100\,000 + 80\,000 + 5\,000 = 185\,000$ pro každý měsíc, tedy pro každé t = 1

5.2.3 Výpočet pro variantu a)

Pro první variantu bychom aplikaci nasadili v nejbližším možném termínu a nečekali bychom na další kvartální nasazování, tudíž n = 0. Testovali bychom po dobu 1 měsíce, platí, že t + n*Q = 1.

Označme jako množství chyb nalezených při testování Mt a množství chyb nalezených v produkci Mp. Pro obě varianty platí M = 30, ale tester stanovil čas potřebný k dosažení tohoto výsledku 4 měsíce a varianta a) znamená dobu testování pouze 1 měsíc, tudíž platí předpoklad, že tester má jen 25 % potřebného času. V takovém případě bude Mp = (M / 100) * 30 a Mt = (M / 100) * 60, tudíž Mp = 9 a Mt = 18.

Pro variantu a) tedy platí:

Náklady chyb objevujících se v produkci:

$C_o(T) = c_2 * Mp = 40\,000 * 9 = 360\,000$ Kč

Náklady na testování:

$C_t(T) = c_1 * Mt + c_3 * 1 = 400 * 18 + 185\,000 = 192\,200$ Kč

Celkové náklady:

$C(T) = C_t(T) + C_o(T) = 192\,200 + 360\,000 = 552\,200$ Kč

5.2.4 Výpočet pro variantu b)

Pro druhou variantu nasadíme aplikaci až po důkladném otestování za 4 měsíce, kdy n = 1 a proto platí t + n*Q = 4.

Dále platí, že M = Mt = 30 a Mp = 3 (Mp je v tomto případě 10% z M).

Pro variantu b) tedy platí:

Náklady chyb objevujících se v produkci:

$C_o(T) = c_2 * Mp = 40\,000 * 3 = 120\,000$ Kč

Náklady na testování:

$C_t(T) = c_1 * Mt + c_3 * 4 = (400 * 30) + (185\,000 * 4) = 752\,000$ Kč

Celkové náklady:

$C(T) = C_t(T) + C_o(T) = 752\,000 + 120\,000 = 864\,000$ Kč

Protože hledáme minimální C(T) a $552\,200 < 864\,000$, rozhodneme se v tomto případě pro variantu a), tedy nasadit aplikaci po jednom měsíci testování. Ukázalo se, že testovat aplikaci po další 3 měsíce by bylo dražší než ji nasadit po jednom měsíci a není proto třeba zabývat se dalšími variantami.

5.2.5 Citlivostní analýza výsledků

V našem případě je výhodnější nasadit aplikaci v nejbližším možném termínu. Je vidět, že cena za odstranění chyby během testování c_1 je velmi nepatrná a nemá na výsledek příliš významný vliv. Měsíční náklady na testera mají vliv značný, tato proměnná se však v praxi příliš nemění, není běžné, aby měl tester třetinové nebo trojnásobné náklady oproti průměru.

Větší význam už má cena za odstranění chyby v produkci c_2. Všimněme si, že v tomto příkladě byl počet očekávaných chyb v produkci v případě, kdy testování trvalo pouze jeden měsíc, třikrát vyšší oproti stavu, kdy by testování trvalo 4 měsíce. To není příliš vysoký rozdíl, pokud by časové omezení testování znamenalo desetkrát vyšší výskyt chyb v produkci,

mělo by to na výsledek významný vliv. Tento vliv by rostl přímo úměrně se zvyšující se cenou za opravu chyby v produkci.

Dále cena za každý měsíc nenasazení úprav v aplikaci, kterou stanovil zadavatel, činí 100 000 Kč. To je v tomto případě vysoká částka, která převyšuje měsíční náklady na testera a významně zvyšuje cenu testování.

Mezi nejvýznamnějšími proměnnými faktory, které ovlivňují dobu, po kterou se ještě vyplatí testovat, tedy patří:

- Měsíční cena ušlé příležitosti za nenasazení aplikace.
- Cena opravy a dopadů chyb v produkci.
- Množství chyb v produkci, které je tester schopen svým testováním redukovat.

5.3 Ekonomické zhodnocení původního a nového stavu

V této kapitole bude provedeno porovnání nákladů a výnosů testování v původním a novém stavu u těch položek, kde je toto srovnání smysluplné, relevantní a zároveň je výstup porovnání použitelný pro manažerské zhodnocení.

5.3.1 Vymezení ekonomického hodnocení původního a nového stavu

Původním stavem se v této kapitole myslí tester ve sjednoceném testingu bez zaintegrování do produktového týmu, novým stavem pak FAT tester jako člen produktového týmu. Vymezení rozsahu ekonomického hodnocení původního a nového stavu je výhradně

zaměřeno na porovnání testera v produktovém týmu a sjednoceném testingu na základě těchto předpokladů:

- Testeři, kteří přešli ze sjednoceného testování do produktového týmu, jsou **vhodným reprezentativním vzorkem**. Jejich výkonnost a pracovní zkušenost se během porovnávaného období příliš nezměnily[22] a jsou k dispozici informace o nákladech na tyto zaměstnance i měřitelná data výstupů jejich práce.

- Z důvodu nových bankovních systémů nutných pro správné fungování banky je nezbytně nutné provádění SIT integračních testů i UAT testů v každém případě. **Jediná položka, kterou je možno měnit a organizačně přesouvat, jsou právě FAT testeři** - u nich je otázka, zda je efektivnější pokud testují v produktových týmech nebo ve sjednoceném testingu.

Při volbě srovnávání testera byla brána v úvahu i tato hlediska:

- I v novém stavu existuje sjednocené testování (SIT a UAT testing), má však na starost testování integrace mezi různými systémy které nebylo prováděno v plném rozsahu v původním stavu, toto testování se nyní provádí u všech aplikací z důvodu plného pokrytí všech systémů v rámci zlepšení procesu testování.

- Test koordinátoři dělají stejnou práci ve sjednoceném testingu v novém i původním stavu (mají na starost koordinaci projektů a UAT testů) a změna pro ně je prakticky nulová.

Z důvodu bankovního tajemství, smlouvy o mlčenlivosti a nevynášení interních informací o zaměstnavateli bude v této kapitole provedena jistá anonymizace dat. Nebudou uváděny konkrétní názvy testovaných aplikací, ale jejich parametry jako rozsah, složitost apod., budou

[22] Několik měsíců praxe navíc má zanedbatelný význam v případě kdy se jedná o zaměstnance s praxí více než 3 roky.

uváděny v koeficientech odpovídajících skutečnosti. Základním předpokladem tohoto porovnání je, že náklady původního a nového stavu jsou stejné, tedy mzdy testerů se po jejich organizačním přesunu nijak nezměnily. Vytížení ostatních členů produktového týmu v novém stavu oproti stavu původním je zanedbatelný faktor který nebude brán v úvahu, reálně je toto vytížení přibližně stejné.

Přestože náklady jsou stejné, bude v pozdější části kapitoly uvedena částka reprezentující měsíční náklad banky na testera - pro představu jaká částka se skutečně ušetřila touto změnou organizace testingu. Protože jakékoliv zveřejňování skutečných mezd je z výše zmíněných důvodů přísně vyloučeno, bude pro tuto představu použit údaj o průměrné mzdě testera z oficiálních stránek ČSÚ.

5.4 Způsob porovnání výnosů a nákladů testování

Náklady na testování jsou jednoznačně měřitelná a rozpoznatelná položka, definovaná přímými i nepřímými náklady na zaměstnance (testera). Podle [7] znamená kalkulace nákladů správné stanovení nákladů na jednici produktu, reprezentovanou jako *kalkulační jednice* „KJ“. Touto kalkulační jednicí je v našem případě prodávaná činnost – testing.

Pokud by kalkulační jednicí byl produkt (výrobek), bylo by množství výnosů snadno měřitelné. Při objektivním měření testování nás však nezajímá KJ jen z hlediska kvantitativního, tj. jak dlouho byl produkt testován případně kolik bylo skutečně otestováno, ale pro posouzení skutečného přínosu je třeba zohlednit i hledisko kvalitativní. Se stoprocentní přesností takové posouzení není nikdy možné, protože testování je činnost závislá na mnoha faktorech s minimem monotónnosti, pokud jde o porovnávání testování různých aplikací.

Pro názornost tohoto příkladu lze porovnat dva typy činností, výstavbu zdi a otestování aplikace. Zatímco rychlost a přesnost s jakou je zeď postavena lze zjistit velmi snadno, stejně jako by tomu bylo u řady dalších činností příp. výrobků, u testování software je tomu jinak.

Nyní bude následovat zamyšlení nad některými otázkami, které komplikují měřitelnost výstupů testování a které si klademe a mají vliv na samotný proces testování. **Pokud dva různí testeři testují dvě různé aplikace, mají srovnatelné podmínky právě tehdy, když jsou kladně zodpovězeny následující otázky:**

- Jsou aplikace stejně složité?
- Jsou aplikace stejně rozsáhlé?
- Běží aplikace na stejně rychlém HW, tedy mají oba testeři stejně dlouhou odezvu?
- Existuje dokumentace pro obě aplikace? Je tato dokumentace stejně přehledná?
- Jsou pro obě aplikace k dispozici testovací data?
- Pokud je třeba vytvořit testovací data, je tato tvorba dat pro obě aplikace stejně náročná?
- Vytvořili oba testeři pro účely testování stejně kvalitní test analýzu?
- Byl výstupem obou testování stejně kvalitní test report?
- Měli oba testeři k dispozici funkční testovací prostředí po celou dobu testování aplikace?
- Reportovali oba testeři chyby stejnou metodou (1 nález = 1 chyba)?
- Našli oba testeři chyby o stejné důležitosti?

- Byly reálně zachytitelné chyby, které testeři nenašli, nebo se jednalo o nahodilé chyby vznikající ve zcela unikátních situacích při provozu, které lze při procesu testování jen velmi těžko odhalit?

- Měli testeři stejnou podporu od analytiků, vývojářů a všech ostatních zainteresovaných osob?

Je zjevné, že ve skutečnosti nejsou téměř nikdy naprosto stejné podmínky pro dvě různá testování. Téměř žádná firma nemá prostředky k tomu, aby více testerů provádělo stejné testy stejných aplikací. Z toho lze vyvodit, že **žádné měření testování nemůže změřit naprosto objektivně výkonnost testera**, protože ve skutečnosti nikdy nenastane přímé porovnání dvou měřitelných veličin za stejných podmínek. V procesu testování je zahrnuto příliš mnoho proměnných, např. i včetně toho pod jakým tlakem je tester veden k tomu, aby v daném čase dokončil testování některé části aplikace.

Z toho důvodu nebude ani porovnání prováděné v této kapitole zcela exaktní. Některé vlivy jsou však zanedbatelné a některé veličiny se měřit dají, je však potřeba přistoupit k měření citlivě a promyslet vždy konkrétní měřenou situaci se znalostí okolností a souvislostí v daném projektu. Neexistuje metrika schopna zcela plně zachytit komplexní proces testování, místo toho **je vhodné pro zpřesnění měření kombinovat více různých metrik** [8].

5.5 Stanovení nákladů

Označíme-li testera ze sjednoceného testování T_1 a testera z produktového týmu T_2 podle časové posloupnosti, při níž jsme od původního sjednoceného testování přešli organizačně k novému testingu integrovaného v produktových týmech, pak pro náklady s mírnou odchylkou platí[23]

[23] Z důvodu stejné mzdy T_1 a T_2 i přibližně stejných ostatních nákladech na zaměstnance v produktovém týmu i sjednoceném testingu.

$$T_1 = T_2.$$

Během porovnávaného období se nezměnila seniorita ani mzdy testerů. Ostatní nepřímé náklady na zaměstnance, jako např. sw. licence, pronájem pracovního místa, cena ušlé příležitosti ostatních IT pracovníků se kterými SW konzultuje svou práci a další, zůstaly prakticky nezměněny. Pokud v těchto nákladech některé změny nastaly, pak jsou natolik drobné a zanedbatelné, že se můžeme spokojit s tvrzením $T_1 = T_2$.

Konkrétní hodnotou představující náklad na testování bude součet přímých i nepřímých nákladů na zaměstnance – testera. Protože proces interního testování je v našem případě aktivita prováděná v centrále banky a tato centrála je v hlavním městě, je třeba při mzdových nákladech uvažovat dražší lokalitu. Podle [13] je průměrná hrubá měsíční mzda IT odborníků v kraji Praha v roce 2012 přibližně 50 000 Kč. Není však přímo definováno, že se jedná o pozici tester a navíc se jedná o aritmetický průměr.

Podle [12] je medián hrubé měsíční mzdy v korunách za 4. čtvrtletí 2012 pro hlavní město Praha na pozici „Specialisté v oblasti testování softwaru a příbuzní pracovníci“ 58 212 Kč, se započítáním odměn, přesčasů, náhrad je to 72 796 Kč. Použitelnost tohoto údaje je omezena tím, že oblastí „příbuzní pracovníci“ se myslí i test manažeři a další zaměstnanci na jiné pozici než „IT tester“, kteří tuto mzdu značně zkreslují. Podle [11] bylo zjištěno, že je významný rozdíl mezi externími a interními testery. Protože kmenoví zaměstnanci banky testující vyvíjený software jsou v případě produktových týmů i sjednoceného testování interními zaměstnanci banky, je třeba počítat s nákladem na interní zaměstnance. Podle [11] je průměrná měsíční hrubá mzda interního testera včetně odměn 43 000 Kč. Superhrubá mzda a nepřímé náklady tuto částku zdvojnásobí [11].

Zde je **přehled** některých významných **nepřímých nákladů na testera**:

- Nájem pracovního místa včetně služeb (kancelář – úklid, fyzická ostraha objektu apod.)
- Licence SW

- Nákup HW (nutná obnova po určitém čase)
- Čas ostatních zaměstnanců v IT vývoji kvůli podpoře testera
- IT podpora (instalace aplikací apod.)
- Vedoucí zaměstnanec
- Mzdové účetnictví
- Oddělení IT bezpečnosti – proti úniku dat, přenosu virů a dalšího škodlivého SW do interní sítě banky apod.
- Dovolená, pracovní neschopnost, školení zaměstnance

Podle [11], kde se vycházelo z diskuze s manažery oddělení testování, jsou průměrné měsíční náklady zaměstnavatele na jednoho interního testera, mezi které zahrnujeme všechny přímé (mzdové) i nepřímé náklady, dohromady 86 000 Kč. Z této částky budeme vycházet v pozdější části kapitoly.

5.6 Stanovení výnosů

Stanovení výnosů bude podstatně složitější než u nákladů. Pomocí definování koeficientů zohledňujících kvantitativní a kvalitativní výstupy a pokrývající většinu z otázek výše vypsaných bodů bude v následující kapitole odhadnut vhodný výpočet pro co nejpřesnější určení výnosů z testování.

Jak již bylo zmíněno, testování samo o sobě přímý finanční výnos nepřináší, hlavním přínosem testování je informace o stavu v jakém daná aplikace je. Kromě tohoto přínosu vznikají v rámci procesu testování jako test analýza scénáře, podle kterých je možné aplikaci znovu testovat a test report, souhrnně popisující informace o proběhlém testování. Výše

zmíněné jsou kvantitativně měřitelné ukazatele za předpokladu, že omezíme doplněním vhodných koeficientů některé další vlivy. V kapitole 5.4 byly pomocí otázek vymezeny hlavní vlivy na testování, některé jsou dostatečně zanedbatelné, jiné se dají vyjádřit pomocí koeficientů, jejichž určení je závislé na znalosti porovnávaných aplikací, prostředí a dalšího. Jako výnos testování lze tedy v této kapitole uvažovat kvantitativně vyjádřené množství otestovaných aplikací. Je však třeba zohlednit i kvalitativní složku, opět s citlivým přihlédnutím vždy ke konkrétní individuální situaci v procesu testování a okolnostem. Problematika vyjádření klíčových ukazatelů v testování (KPI) bude podrobněji probrána v dalších kapitolách, nyní bude uveden pro přehled základ této problematiky.

5.6.1 Stanovení vhodných metrik v procesu testování

Pomocí metrik lze objektivně měřit některé výstupy testování. Je však třeba přistupovat k metrikám obezřetně a nevnímat je jako absolutní ukazatel výkonnosti nebo kvality, ale vnímat jejich přidanou informační hodnotu. Neexistuje natolik univerzální metrika, aby bylo možné ji aplikovat v každém procesu testování stejným způsobem, vždy je vhodné promyslet modifikaci metriky tak, aby co nejpřesněji měřila požadované ukazatele a brát v potaz působení okolních vlivů.

Notoricky známým případem nevhodného použití metrik bylo ustanovení nejmenované IT firmy, kdy bylo rozhodnuto o tom, že vývojáři budou hodnoceni podle počtu řádků zdrojového kódu, který napíší. Krátce po aplikaci tohoto rozhodnutí do praxe se ukázala jeho nesmyslnost, vývojáři se snažili roztáhnout kód do co nejvyššího množství řádků a psali ho složitěji a nepřehledněji. Kvalita a přehlednost kódu šla rapidně dolů. To samé nastane v případě, že tester bude hodnocen pouze na základě metriky počtu nalezených chyb. Při tomto pravidle řada testerů začne hlásit i sebemenší překlepy jako samostatné chyby[24], reportovat řadu drobností v dokumentaci aplikace, které by přitom bylo efektivnější vyřešit domluvou s analytikem apod.

[24] V některých případech je vhodnější reportovat více drobných a nepodstatných překlepů v aplikaci najednou do jednoho bugu, vývojář je stejně opraví naráz a sníží se tím režijní náklady testera i vývojáře, stejně jako se zvýší přehlednost v nástroji na reportování bugů kde nevzniknou stovky drobných chyb o velmi nízké závažnosti.

Velmi rozšířená metrika je porovnat počet chyb nalezených v testování testerem a součtem chyb nalezených při akceptačním testování a v produkci po nasazení aplikace. I v tomto případě je však vhodné postupovat k ukazateli obezřetně, někdy je akceptační testování z nedostatku času zadavatelů provedeno velmi rychle a testovány jsou jen nejnutnější scénáře, jindy si zadavatel podrobně protestuje aplikaci a objeví např. i chyby, které tam už existují roky a s testovanou funkčností vůbec nesouvisí. Stejně tak při chybách, které se objeví později v produkci, je dobré vždy přihlédnout k tomu, zda bylo reálně možné, aby tyto chyby tester během testování odhalil.

Čtenáři, neznalému podrobně bankovní prostředí ani oblast testování, lze na obecném příkladě názorně přiblížit, co se v jednom z případů může skrývat pod tím, že „*musí být reálně možné, aby chyby tester během testování odhalil*“ :

> **Příklad chyby, která není** standardním manuálním testováním bez podpůrných nástrojů **reálně odhalitelná:**
>
> Máme za úkol otestovat jednoduchý kalkulátor provádějící pouze součet dvou čísel. Testujeme výpočetní část, kde musí platit a + b = c. Zadáváme postupně různé vstupy a,b a patnáctkrát nám vyjde správné c.
>
> Po pozdějším nasazení aplikace do produkce se však po půl roce ukáže, že kombinace a + b = c zafungují vždy správně kromě situace, kdy a = 751, b = 623 a jejich součet c pak aplikace zcela výjimečně v tomto případě vyhodnotí jako 1375 místo 1374, je to tedy závažná chyba způsobující chybná data. Je v takovém případě možné chtít po testerovi aby tuto chybu odhalil během testování?[25]

[25]Toto byl pouze ilustrativní případ, ve skutečnosti se v bance nevyskytne takováto situace při součtu dvou čísel, ale může se stát že pro zcela unikátní IČO nebude z nějakého důvodu fungovat správně import, a vzhledem k omezeným časovým i technickým zdrojům při testování zkrátka nemusí být reálně možné otestovat dostatečně velkou množinu dat (IČ) na to, aby se na tuto chybu přišlo.

5.6.2 Koeficienty pro stanovení výnosů z testování

První dvě otázky, zda jsou testované aplikace stejně **složité i rozsáhlé**, mají v disciplíně testování velmi zásadní význam. Rozdíl ve složitosti i velikosti jednotlivých aplikací používaných v bance může být i desetinásobný a vnitřní integrita aplikace může být narušena zavlečením nových chyb i do těch částí aplikace, které se neměnily a tester s nimi pro testování nepočítá. Je to však i riziko testera, ten by měl provést regresní testy minimalizující riziko zavlečení nových chyb i do těch částí aplikace, na kterých se nemělo nic měnit. Tyto regresní testy znamenají pro testera další práci, která většinou není příliš viditelná a je proto třeba velikost i složitost aplikace zohlednit. Mějme proto **koeficienty A_v a A_s stanovující velikost a složitost aplikace** v podmnožině porovnávaných testovaných aplikací[26]. Tyto koeficienty musí stanovit osoba s důkladnou znalostí všech porovnávaných aplikací.

Dále je třeba brát v potaz **HW a odezvu chování aplikace.** HW by měl být dostatečný ke spuštění a testování příslušné aplikace, tato podmínka bývá v popisované bance nesplněna jen ve velmi výjimečných situacích a lze ji proto zanedbat. Odezva je však závislá na aplikaci samotné – jsou operace, kdy tester pro jejich dokončení musí čekat řádově minuty a to už významný vliv má. V případě našeho porovnávání je pak vhodné toto zahrnout do **koeficientu složitosti aplikace A_s** a brát v potaz, že složitostí se myslí i nepříjemnost způsobená pomalejší odezvou, příp. dokončením operace, na které tester musí čekat řádově minuty.

Dokumentace, tedy analýza popisující žádoucí chování aplikace, je pro testera velmi zásadní faktor. Zadavatel většinou neocení vliv testera na dokumentaci, je pro něj podstatná kvalita aplikace. Pro testera pak znamená více práce a dalších režijních nákladů opravovat a hlídat dokumentaci, která nepřesně nebo chybně popisuje aplikaci. **Koeficient kvality dokumentace D_k** bude mít pro své určení stejná pravidla, jako měly koeficienty A_v a A_s s tím

[26] Vždy je třeba zohlednit v jaké množině aplikací testovanou aplikaci uvažujeme. Velikost i složitost by neměly být univerzální koeficienty aplikovatelné na všechny aplikace v bance. Je tomu tak z důvodu přesnosti odhadu velikosti a složitosti při zachování stejných jednotek těchto koeficientů - čím větší množinu aplikací budeme porovnávat, tím zkreslenější odhad složitosti i velikosti dostaneme.

rozdílem, že osoba stanovující tento koeficient musí kromě množiny aplikací znát i množinu jejich dokumentací a porovnat kvality těchto dokumentací.

Testovací data a náročnost na jejich tvorbu lze promítnout do **koeficientů kvality a rozsahu aplikace**. Tvorba **test analýzy** je v případě tohoto porovnávání **zanedbatelný faktor** – tester ji vytváří v obou případech v přibližně stejné kvalitě a používá ji primárně pro své účely. Podobně je tomu i u **test reportu** – jím tester oficiálně dokumentuje výstup své práce a pokud dělal test report ve sjednoceném testingu, pokračuje v tom i jako tester produktového týmu.

Nefunkční testovací prostředí pro testování aplikace může být způsobeno plánovanou odstávkou, neplánovaným výpadkem nebo dalšími vlivy. Problémem jsou však další možné varianty nedostupnosti aplikace k otestování – vývojář omylem nasadí špatnou verzi aplikace, databázový server je zahlcen nebo může dojít výjimečně k výpadku internetového připojení při testování webových aplikací. Na všechny tyto případy lze aplikovat souhrnný **koeficient dostupnosti testovacího prostředí** $\mathbf{P_d}$, který zohlední dobu, po kterou bylo během testování prostředí dostupné.

Jednotnost metodiky **reportování chyb** zajišťuje procesně **metodika testování v bance**, podle které by se měli všichni testeři i ostatní účastníci procesu testování řídit. Pokud je podle metodiky vhodné reportovat více drobných chyb v rámci jednoho bugu[27], pak se tohoto pravidla musí držet testeři FAT, SIT, UAT i uživatelé nahlašující chyby v produkci a stejně jednotně musí zadávat chyby, pokud bude striktně definované, že každá chyba musí být zadána zvlášť.

U reportovaných chyb je důležitá jejich **závažnost**. Při zadávání chyby je tato závažnost udávána jako číslice reprezentující dopad neodstranění této chyby. Čím nižší tato číslice je, tím závažnější chyba – 1 je kritická chyba při které se náhle ukončí aplikace, 4 je např. nepodstatný drobný grafický nedostatek posunutí rámce okna. Tester, který najde závažné chyby způsobující ztrátu dat, pád aplikace apod. má mnohem větší přínos než tester který tyto chyby nenajde a nachází pouze drobné chyby, přesto že se při dalším testování nebo používání aplikace v provozu závažné chyby objeví. Nabízí se tedy **koeficient závažnosti**

[27] Bugem se myslí v tomto kontextu reportovaná chyba – chyba zadaná do nástroje pro reportování chyb.

reportovaných chyb, pro objektivní posouzení však nelze takto jednoduše tento koeficient stanovit, je třeba vzít v potaz další faktor. Tímto faktorem je, zda **bylo tyto chyby skutečně možné během testování nalézt**. V aplikaci i po důkladném otestování mohou zůstat chyby, které nejsou nikdy v životním cyklu užívání aplikace odhaleny – tento faktor však nelze příliš měřit ani brát v potaz. Velmi dobře je však měřitelný součet chyb nalezených během akceptačního testování a chyb nalezených během nasazení aplikace v produkci, pokud někdo objektivně posoudí, zda tester tyto chyby mohl reálně při procesu testování nalézt. Pokud je tester nalézt mohl, pak - se zohledněním závažnosti u každé jednotlivé chyby - vzniká klíčová a velmi použitelná metrika stanovující kvalitu testování. Modifikací obecně známé metriky efektivita odstraňování chyb DRE [1],[2], [10], kde platí

DRE = (Chyby nalezené během fáze testování / chyby nalezené v produkci) * 100 %

Úpravou DRE dostaneme v následující kapitole vzorec pro **koeficient kvality testování T_k**, který nám umožní objektivně posoudit kvalitu, s jakou bylo testování provedeno.

5.6.3 Koeficient kvality testování T_k

Mějme **koeficient závažnosti chyby K_z**, pro který platí:

$K_z = 5$ pro kritické chyby[28]

$K_z = 3$ pro závažné chyby

$K_z = 2$ pro středně závažné chyby

$K_z = 1$ pro méně závažné, drobné chyby

Pak pro chyby nalezené ve fázi x mějme koeficient chyb nalezených ve fázi x K_x, pro který platí, že máme-li celkové množství chyb nalezených ve fázi x

[28] $K_Z = 4$ nebyl definován z důvodu nalezení vhodných vah kritických chyb. Závažné chyby mají přibližně třikrát větší váhu než drobné chyby, proto $K_Z = 3$ pro závažné chyby. Pro kritické chyby by však zvýšení o jednu číslici na $K_Z = 4$ bylo příliš nízké, kritické chyby odpovídají spíše koeficientu závažnosti $K_Z = 5$.

$$M_x = M_{1x}, M_{2x} \ldots. M_{nx}$$

Kde M_{1x} značí první nalezenou chybu ve fázi x,

pak pro **koeficient chyb K_x** platí:

$$K_x = M_{1x}K_{z1x} + M_{2x}K_{z2x} + \ldots.. + M_{nx}K_{znx}$$

Kde $M_{1x}K_{z1x}$ značí první nalezenou chybu ve fázi x vynásobenou koeficientem závažnosti této chyby[29].

Tedy koeficient chyb nalezených ve fázi x označený jako K_x je dán součtem všech koeficienty závažnosti K_z vynásobených chyb, pokud byly tyto chyby nalezeny ve fázi x.

Pro koeficient kvality testování T_k pak platí:

$$T_k = (K_{fat} / (K_{sit} + K_{uat} + K_{prod}))$$

Kde:

K_{fat} je koeficient chyb nalezených ve fázi FAT testování

K_{sit} je koeficient chyb nalezených ve fázi SIT testování

K_{uat} je koeficient chyb nalezených ve fázi UAT testování

K_{prod} je koeficient chyb nalezených ve fázi nasazení aplikace v produkci

[29] Pro výpočet bude Mx vždy rovno 1, jedná se pouze o reprezentaci konkrétní chyby.

Vypovídací hodnotu pak budou mít zejména K_{uat} a K_{prod}, jejich součet znamená ve většině případů chyby, které nebyly nalezeny FAT testy. U všech koeficientů chyb kromě K_{fat} je třeba ještě zvážit, zda tyto chyby mohly být ve fázi funkčního akceptačního testování nalezeny - pokud nikoliv, pak by do těchto koeficientů neměly být započítány. Například chyba nalezená při UAT testech na systému, který je k dispozici pouze v UAT prostředí a FAT tester tudíž nemůže provést test, který daný systém vyžaduje, by se neměla do koeficientu K_{uat} započítat. K_{sit} bude nenulový pouze v případě, kdy po FAT testování určité oblasti testerem produktového týmu následovaly také SIT testy pro tuto oblast. V jiném případě by K_{sit} neměl smysl.

Otázka **podpory vývojářů a analytiků testovacímu týmu** by měla být předmětem interní metodiky vývoje software v bance. Základním cílem této podpory je vysvětlit testerovi podrobně fungování aplikace, pokud toto není zcela jednoznačně popsáno v dokumentaci. Analytik je tak zpravidla základním nositelem znalosti očekávaného funkčního i nefunkčního chování aplikace, vývojář by měl vysvětlit podrobněji technické detaily, případně pomoci s přípravou dat a simulováním některých situací. Měřitelnost toho, jak moc vývojáři a analytici podporují testera, není dost dobře možná, jedná se o velmi subjektivní posouzení závislé na vytíženosti vývojového týmu, osobních sympatií a přístupu testera. Pokud má tester pocit, že se mu **nedostává dostatečné podpory od analytiků** za předpokladu, že popis v analýze není jednoznačný nebo zcela neodpovídá chování aplikace, má k dispozici nástroj k eskalaci tohoto problému – reportování chyb na analýzu, což se projeví v **koeficientu kvality dokumentace D_k.**

5.6.4 Duplicita chyb

Další významný faktor je duplicita chyb. Je možné, že chyby zadané z produkce jsou zadány dvakrát. Obvykle je tomu tak buď z toho důvodu, že stejnou chybu podruhé zadal jiný uživatel, který o prvním zadání nevěděl, nebo chybu podruhé zadal stejný uživatel, který si již nepamatoval, že stejnou chybu předtím zadával. V každém případě je třeba duplicitní chyby do koeficientů nezapočítávat. Je však třeba citlivě ošetřit, co v tomto kontextu je duplicitní chyba a co není – nezřídka se stává, že jedna chyba ve zdrojovém kódu nebo v databázi způsobuje několik různých chyb z pohledu uživatele v aplikaci. V tomto případě se tyto chyby za duplicitní nepovažují, vycházíme z toho, jak se chyby v aplikaci projeví a jak je

vnímá uživatel, nikoliv z toho, čím byly chyby způsobeny a zda se tou samou opravou ve zdrojovém kódu opraví několik chyb naráz. Někdy je i otázka duplicity diskutabilní, např. pokud se v menu A zobrazují špatně nadpisy, bude duplicitní chybou i to, že se nadpisy zobrazují špatně i v menu B? V takovém případě ano – jednalo by se o duplicitu, uživatel by měl zadat chybu, že se špatně zobrazují nadpisy a vypsat v ní všechna okna, kde se tyto nadpisy špatně zobrazují. Otázku zadávání duplicitních chyb by opět měla řešit metodika testování v bance a není proto třeba na ni vymýšlet další koeficient, postačí informace že duplicitní (samozřejmě i odmítnuté) chyby se nezapočítávají.

Problematika oblasti **zadávání duplicitních chyb** spadá obecně pod otázku způsobu reportování chyb. Způsob reportování chyb **stanovuje metodika testování v bance**.

5.6.5 Pravidla pro přiřazení nalezené chyby do koeficientů K_x

Máme 3 základní podmínky, které musí chyba splňovat, než ji započteme do koeficientů:

- Chyba není duplicitní ani zamítnutá
- Pokud se jedná o chybu nalezenou mimo fázi FAT, pak tuto chybu bylo možné nalézt během funkčního akceptačního testování
- Chyba je v aplikaci, nikoliv ve zvláštní dokumentaci[30]

[30] Pokud je dokumentace součástí aplikace např. jako nápověda, pak se jedná o chybu v aplikaci. Zvláštní dokumentací se myslí analýza aplikace, u které platí, že chyby v ní nalezené obvykle upřesňují kvalitu dokumentace která aplikaci popisuje a kterou zadavatel schvaloval. Pro konečné uživatele však nemá kvalita technické dokumentace tak zásadní význam jako kvalita vlastní aplikace.

5.6.6 Pokrytí odlišností dvou různých testování koeficienty a pravidly

Na začátku kapitoly „Způsob porovnání nákladů a výnosů testování“ jsou v bodech vypsány **otázky**, vedoucí k zamyšlení nad **odlišnými podmínkami dvou různých testerů testujících dvě různé aplikace**. Z těchto otázek vycházejí některé parametry testování. Abychom měli jistotu, že tyto **parametry jsou plně pokryty** buď pomocí koeficientů, nebo pomocí interních pravidel a norem, bude zde uvedena pro přehled tabulka tohoto pokrytí.

Parametry které musejí být přibližně stejné pro účely porovnání dvou různých testování	Zajištění rovnosti parametrů
Složitost aplikací	Koeficient složitosti aplikace As
Rozsah aplikací	Koeficient velikosti aplikace Av
Odezva aplikací	Koeficient složitosti aplikace As
Existence dokumentace	Koeficient kvality dokumentace Dk
Kvalita dokumentace	Koeficient kvality dokumentace Dk
Existence testovacích dat	Koeficient velikosti aplikace Av
Náročnost tvorby testovacích dat	Koeficient složitosti aplikace As
Kvalita test analýzy	Slouží primárně testerovi, zanedbatelný faktor
Kvalita test reportu	Slouží primárně testerovi, zanedbatelný faktor
Dostupnost testovacího prostředí	Koeficient dostupnosti testovacího prostředí Pd
Způsob reportování chyb	Metodika testování v bance
Důležitost nalezených chyb	Koeficient kvality testování Tk (součástí jeho výpočtu koeficient závažnosti chyby Kz)
Reálná možnost odhalení neodhalených chyb ve fázi FAT	Koeficient kvality testování Tk (jedno z jeho tří pravidel)
Podpora testingu od analytiků a vývojářů	Metodika vývoje v bance, Koeficient kvality dokumentace Dk

Tab.1 Pokrytí parametrů testování koeficienty a pravidly [autor]

5.6.7 Povaha a vypovídací hodnota hlavních koeficientů

Celý systém koeficientů je zaváděn pro zpřesnění výstupu práce testera a co nejobjektivnější posouzení kvantitativního i kvalitativního výstupu práce testera v čase.

Máme v zásadě 4 hlavní kvantitativní koeficienty: A_s, A_v, D_k a P_d a jeden kvalitativní: T_k, k jehož stanovení slouží další pomocné koeficienty K_x pro testování ve fázi x a K_{znx} pro každou chybu M_{nx}. Kvantitativní povaha je u koeficientů A_s a A_v zřejmá, D_k je pak kvantitativním z toho důvodu, že oprava dokumentace představuje pro testera vyšší množství práce, která není pro zadavatele aplikace většinou nikde vidět a přitom sama dokumentace kvalitu aplikace přímo neovlivní. P_d je kvantitativním z toho důvodu, že nedostupnost prostředí znamená nemožnost testovat, z toho vyplývající prostoje příp. režijní a administrativní práci pro testera, která mu ubírá čas potřebný k testování a rovněž se v konečném výstupu neprojeví.

Koeficient kvality testování T_k má pak jednoznačně kvalitativní charakter, neboť představuje poměr chyb nalezených při FAT testování a později zohledněný jejich závažností. Tento koeficient je zde pouze pro představu o kvalitě FAT testování v původním a novém stavu, z důvodu omezení, která by tuto kvalitu zkreslovala tak není příliš dobrým ukazatelem celkového smyslu a efektivity testování[31].

5.6.8 Data pro stanovení koeficientů

Porovnávané období, během kterého nashromáždíme relevantní data, činí celkem 8 měsíců – **4 měsíce v původním stavu a 4 měsíce v novém stavu**. Tato délka byla vybrána ze dvou důvodů:

- Období je dost dlouhé na to, aby data za toto období nebyla příliš zkreslena krátkodobými výjimečnými událostmi

[31] Koeficient kvality testování má omezující pravidla a nezohledňuje např. chyby, které se nalezly ve fázích UAT a PROD a zároveň je nebylo možné nalézt ve fázi FAT testování. Pokud však takové chyby existují a je jich mnoho, nastává otázka čím je to způsobeno a jak zlepšit a zefektivnit FAT testování aby zachytilo i tyto chyby. Může se jednat o přiblížení testovacího prostředí produkčnímu, zapojení integrace nebo přiblížení testovacích dat na FAT prostředí těm, které jsou k dispozici v té fázi, ve které se chyby objevily, nejdůležitější bývá fáze PROD, kde mají vyskytující se chyby nejdražší dopad.

- Období není příliš dlouhé na to, aby v něm hrála významnou roli změna seniority všech účastníků procesu testování

Nasbíraná data mají dva hlavní **zdroje:**

1. **Reportovací nástroje a databáze pro sběr dat ke kvalitativním koeficientům**

    - Reportovací nástroj pro prostředí FAT nebo sjednoceného testování, ve kterém jsou reportovány všechny chyby včetně jejich závažnosti.

    - Další reportovací nástroje a databáze, ve kterých lze dohledat chyby nalezené v pozdějších fázích testování – SIT, UAT a PROD.

2. **Přímé účastníky procesu testování pro stanovení kvantitativních koeficientů**

    - Těmito účastníky mohou být tester, vývojář, analytik příp. správce testovacího prostředí nebo manažer dodávky.

    - Účastníci stanoví koeficienty jako velikost aplikace, složitost aplikace a kvalita dokumentace na základě množiny aplikací, v rámci které toto posouzení probíhá – jedná se o všechny aplikace, na kterých sledovaný tester během porovnávaného období testoval.

    - Pro všechny účastníky, kteří se podílejí na procesu stanovování příslušných koeficientů, je vyžadována hluboká znalost všech aplikací v porovnávané množině, zejména pak jejich vlastností ovlivňujících příslušný koeficient. Př.: Pokud analytik přispěje svým vyjádřením k tomu, jak složitá a veliká je jeho

aplikace oproti ostatním aplikacím v porovnávané množině, je nezbytně nutné, aby znal důkladně všechny tyto aplikace, alespoň z uživatelského hlediska.

- Za stanovení těchto koeficientů zodpovídá po sběru pohledů od ostatních účastníků procesu testování vždy FAT tester, který se při testování nejvíce seznámil s analýzami i ovládáním všech testovaných aplikací a je tak nejlépe schopen posoudit jejich vzájemnou velikost, složitost, posoudit dokumentace a zohlednit dostupnost testovacího prostředí.

5.6.9 Stanovení kvantitativních koeficientů

Množina posuzovaných aplikací bude představovat všechny aplikace, které příslušný FAT tester testoval za vybraný časový interval během původního i nového stavu – tedy z celého časového období osmi měsíců. V rámci této množiny bude provedeno stanovení kvantitativních koeficientů. Tyto koeficienty budou platné pouze v rámci posuzované množiny aplikací, protože jejich střední, minimální a maximální hodnoty budou vycházet právě z této množiny.

Pro koeficienty velikosti aplikace, složitosti aplikace a kvality dokumentace bude aplikován postup, který uvedeme na příkladu stanovení koeficientu pro velikost aplikace:

Tester stanoví množinu všech sledovaných aplikací a zvolí mezi nimi aplikaci, která má oproti ostatním aplikacím v této zkoumané množině zdánlivě průměrnou velikost[32]. Na základě velikosti této aplikace vytvoří jednotkovou velikost, které přiřadí hodnotu 1 – bude odpovídat velikosti této průměrné aplikace. Poté stanoví velikosti ostatních aplikací tím způsobem, že je porovná s touto jednotkovou velikostí a určí poměr, kolikrát jsou aplikace rozsáhlejší nebo menší. Tudíž aplikace s jednotkovou velikostí bude mít koeficient velikosti aplikace $A_v = 1$, aplikace, která bude 1,5 krát rozsáhlejší bude mít $A_v = 1,5$ a aplikace, která bude o 18% méně rozsáhlá, bude mít $A_v = 0,85$[33].

[32] Stanovení proběhne nejen na základě jeho posouzení, ale po konzultaci s ostatními účastníky procesu testování, kteří mají podrobnou znalost o velikosti všech aplikací ve zkoumané množině. Tato podmínka bude platit i pro ostatní kvantitativní koeficienty.
[33] Aplikace, která bude o 18% méně rozsáhlá se vypočítá jako 1 / 1,18 = 0,85.

Stejným způsobem, jakým tester stanovil A_v, stanoví i A_s a D_k. Je zřejmé, že zatímco ve velikosti i složitosti jednotlivých aplikací mohou být výrazné rozdíly, u dokumentací budou tyto rozdíly menší, už z důvodu závazné metodiky pro psaní dokumentací, kterou předepisují interní normy banky.

Velikost koeficientů A_v i A_s je přímo úměrná velikosti a složitosti aplikace, u D_k je velikost tohoto koeficientu nepřímo úměrná kvalitě dokumentace – čím vyšší kvalita dokumentace, tím menší bude D_k. Je tomu tak z toho důvodu, že čím méně kvalitní dokumentace je, tím více práce to testerovi přidělá, zatímco čím méně rozsáhlá a složitá aplikace, tím méně práce stojí ji otestovat.

Koeficient dostupnosti testovacího prostředí P_d se stanoví odlišným způsobem. Odstávky prostředí jsou oficiálně dokumentovány a archivovány, podstatné však je, které tyto odstávky skutečně zasáhly do konkrétního testování – pouze ty budou brány v úvahu. Jak již bylo popsáno, nemusí jít pouze o odstávky prostředí, ale budou zde zahrnuty i případy, kdy byla testerovi nasazena špatná verze aplikace, tudíž čas strávený testováním a příp. reportováním tester promarnil a představoval pro něj zbytečnou a nadměrnou zátěž neprojevující se ve výstupu procesu testování. Čas, který tester strávil testováním špatné verze, prostojem kvůli nedostupnosti prostředí nebo administrativními či režijními úkony kvůli nedostupnosti části testovacího prostředí se porovná s celkovým časem, který testováním konkrétní aplikace strávil a vyjde poměr představující nedostupnost vs. dostupnost a aktuální testování.

Pokud bylo prostředí se správně nasazenou verzí testované aplikace k dispozici po 80 % času celkově stráveným testováním této aplikace, bude pro tuto konkrétní aplikaci $P_d = 1.2$. Pokud nedošlo k žádným výpadkům ani chybně nasazeným verzím, bude $P_d = 1$.

Při nedostupnosti může tester připravovat data, promýšlet scénáře nebo testovat jinou aplikaci, ve skutečnosti k prostoji zpravidla nedochází, protože vždy je nějaká jiná práce. Přesto tato neplánovaná změna práce a úkony s ní splněné ubírají čas, nutí dělat si vedlejší poznámky o tom, co právě testera napadlo otestovat a není to součást scénáře apod. Nedostupnost také nebývá nijak výrazná, takže tento koeficient nebude mít takový význam jako např. velikost a složišti aplikace – bude se pohybovat v hodnotách od 1.0 do 1.3. Je ale

možné že při jiném testování bude docházet k výpadkům podstatně výraznějším a algoritmus pro výpočet koeficientu dostupnosti prostředí se pak může změnit.

Všechny 4 výše popsané kvantitativní koeficienty dávají určitou představu o tom, jak náročné je aplikaci testovat. O skutečně otestovaných funkcionalitách nám však nic neříkají. Je proto třeba zjistit, kolik procent z celkové aplikace bylo reálně testováno během zkoumaného období. Číslo představující otestované množství O_m pak získáme tak, že všechny 4 kvantitativní koeficienty mezi sebou roznásobíme, vydělíme je stem a vynásobíme procentuelním vyjádřením reálně otestované aplikace O_{pr}. Pokud změny v některých částech aplikace proběhnou vícekrát, pak bude vícekrát započítána i O_{pr}, může se tedy teoreticky stát, že bylo reálně testováno více než 100 % aplikace.

Příklad: Pokud by se polovina aplikace testovala právě jednou, bude $O_{pr} = 50\ \%$. Pokud by se musela otestovat celá aplikace, poté se ještě během zkoumaného období stihla celá přeprogramovat a musela se znovu otestovat podle kompletně změněných scénářů, pak bude $O_{pr} = 200\ \%$.

Vzorec pro výpočet otestovaného množství

$$O_m = ((A_s * A_v * D_k * P_d) / 100) * O_{pr}$$

Kde:

O_m je reálně otestované množství jedné konkrétní aplikace

O_{pr} je kolik procent této aplikace bylo ve sledovaném časovém období skutečně testováno

5.6.10 Stanovení kvalitativních koeficientů množiny posuzovaných aplikací

Kvalitativní koeficienty jsou jednoznačně vypočteny z dat, která obsahují reportovací nástroje pro správu chyb, kde jsou chyby i jejich závažnosti.

Z těchto nástrojů jsou po aplikaci třech základních podmínek[34] vybrány ty chyby, které platí stále i po aplikaci těchto podmínek. U nich se zaznamená pro každou z nich závažnost a ta se přepočte podle pravidel popsaných v kapitole Koeficient kvality testování (pokud se jedná o kritickou chybu, pak $K_z = 5$, pokud o drobný překlep, pak $K_z = 1$ apod.).

Pro každou fázi testování se stanoví její koeficient chyb K_x.

Příklad : Pokud budou nalezeny tři chyby v produkci, z nichž první má v reportovacím nástroji prioritu 1 (kritická) a zbylé dvě 4 (drobný překlep), pak

$$K_x = M_{1x}K_{z1x} + M_{2x}K_{z2x} + M_{3x}K_{z3x}$$

Tudíž $K_{prod} = 1*5 + 1*1 + 1*1 = 7$.

Poté se ze všech koeficientů chyb K_x vypočte výsledný koeficient kvality testování T_k podle vzorce

$$T_k = (K_{fat} / (K_{sit} + K_{uat} + K_{prod}))$$

Kde pro každou fázi testování x[35], která neproběhla, bude $K_x = 0$. Nutnou podmínkou je, že K_{fat} bude vždy nenulové, což je zajištěno tím, že fáze FAT testování je výchozí, ze které

[34] Kdy chyba není zamítnutá ani duplicitní, pokud se jedná o chybu nalezenou jinde než ve FAT, pak bylo možné reálně tuto chybu ve FAT nalézt a jedná se o chybu v aplikaci, nikoliv v dokumentaci.

[35] Běh aplikace na produkci nebývá označen za fázi testování, ale funkčnost aplikace tam používáním svým způsobem testována je, v některých případech se na produkčním prostředí testují funkcionality, které nemají vliv

sbíráme množinu aplikací, pro které koeficienty stanovujeme. V původním stavu bude fáze FAT znamenat funkční akceptační testy v rámci sjednoceného testování, v novém stavu bude znamenat FAT testy prováděné testerem v rámci produktového týmu.

T_k není absolutní údaj jednoznačně vypovídající o kvalitě testování, ale pokud bude aplikován výše popsaným postupem a poměří výstupy práce jednoho konkrétního testera navzájem ve stavu původním a novém, pak nám dává objektivní představu o tom, v jaké kvalitě testování probíhalo v porovnání mezi těmito dvěma stavy. Stejně jako všechny ostatní koeficienty, i T_k se bude lišit pro každou aplikaci.

5.6.11 Výsledek kvantitativních koeficientů a vyhodnocení dat

V této kapitole jsou zobrazena data kvantitativních koeficientů ve stavu původním i novém. Dále je provedena analýza a vyhodnocení dat.

Původní stav - sjednocené testování kvantitativní						
Aplikace	Av	As	Dk	Pd	Opr	Om
Aplikace A1	0,6	0,9	1,6	1,3	130	1,46
Aplikace A2	1	1,3	0,7	1,1	21	0,21
Aplikace A3	1,3	0,7	1	1,2	33	0,36
Aplikace A4	0,5	0,6	1,3	1,1	17	0,07
Součet za všechny aplikace	3,4	3,5	4,6	4,7	201	2,10

Tab.2 kvantitativní koeficienty testera za časové období 4 měsíce pro původní stav [autor]

na klientská ani jiná data, příp. jsou vidět jen pro interní zaměstnance banky a mají je označena jako neplatná data jen pro testovací účely.

Nový stav - produktové týmy kvantitativní						
Aplikace	Av	As	Dk	Pd	Opr	Om
Aplikace A5	1,2	0,8	1,3	1,1	14	0,19
Aplikace A2	1	1,3	0,7	1,1	172	1,72
Aplikace A6	0,7	1	0,9	1	51	0,32
Aplikace A7	1,5	1,5	1,1	1,1	16	0,44
Aplikace A8	0,5	0,3	0,8	1,2	7	0,01
Součet za všechny aplikace	4,9	4,9	4,8	5,5	260	2,68

Tab.3 kvantitativní koeficienty testera za časové období 4 měsíce pro nový stav [autor]

Z důvodu anonymizace dat byly konkrétní názvy aplikací změněny na jednotlivé kódy. Z nich lze usoudit, že sledovaný tester v původním i novém stavu testoval aplikaci označenou jako A2, ostatní aplikace testoval buď pouze jako tester sjednoceného testování nebo jako tester produktového týmu.

Značně se liší i rozsah testovaných aplikací, A4 a A8 jsou malé aplikace, na kterých se navíc testovala pouze malá část, tudíž práce na nich výsledky příliš neovlivňuje. Oproti tomu aplikace A1 není příliš rozsáhlá, ale testovala se během prvních 4 měsíců kompletně celá a to včetně několika změn, tudíž práce na ní má podíl významný.

Složitost aplikace nemusí být přímo úměrná její velikosti, přesto většinou u aplikací v bankovním prostředí roste v případě jejich vylepšování postupem času velikost i složitost současně spolu s tím, jak jsou do aplikace implementovány nové funkčnosti. Malé aplikace nebývají příliš složité, existují výjimky, ale v případě našich dat se nevyskytly.

Z výsledků je vidět, že dostupnost testovacího prostředí se lišila velmi málo a její podíl je rovněž zanedbatelný, pouze aplikace A6 byla vždy nasazena správně na funkčním prostředí bez jakýchkoliv výpadků.

Kvalita dokumentace však dostála významných změn. Nejzajímavější je případ aplikace A1, kde je její koeficient kvality dokumentace silně nadprůměrný a výrazně zvyšuje podíl pracnosti této aplikace. Je tomu tak z toho důvodu, že při testování A1 se v bance vystřídalo několik analytiků, kteří ukončovali pracovní poměr během krátké doby z různých důvodů. Po celou dobu testování tak neexistovala kvalitní analýza ani analytik se znalostí aplikace A1, tester tak strávil spoustu času zjišťováním toho, jak přesně má aplikace fungovat obíháním

zadavatelů a vlastníků této aplikace, pro které to znamená práci navíc, na kterou v některých případech jen velmi obtížně nacházejí čas. Je vidět, že oproti aplikaci, kde je analýza nejkvalitnější to znamená více než dvakrát větší celkovou pracnost.

Z celkového otestovaného množství O_m je vidět, že tester v produktovém týmu otestoval o 27,6% větší množství aplikací než jich otestoval jako tester ve sjednoceném testingu za stejné časové období. Protože celkové O_{pr} se oproti původnímu stavu zvýšilo ještě více, tedy o 29,4%, lze odhadnout, že rozdíl v otestovaném množství není způsoben tím, že by se testovaly v produktovém týmu rozsáhlejší a složitější aplikace, ale skutečně se otestovalo více požadavků.

Protože právě aplikace A2, která byla testována v původním i novém stavu, má v novém stavu největší váhu, mohlo by se zdát, že rozdíl je způsoben tím, že tester poznal během první fáze tuto aplikaci více do detailu a ve druhé z toho těžil. Tato domněnka je mylná, aplikace má v původním stavu velmi nízkou váhu a byla v té době testována pouze z jedné pětiny, tester se seznámil jen s její malou částí. Až v novém stavu se s ní seznámil kompletně, navíc aplikace doznala natolik výrazných změn, že bylo třeba většinu věcí testovat jinak a nebyly pro tyto testy použity scénáře vytvořené v původním stavu.

5.6.12 Výsledek kvalitativních koeficientů a vyhodnocení dat

V této kapitole jsou zobrazena data kvalitativních koeficientů ve stavu původním i novém. Dále je provedena analýza a vyhodnocení dat.

Původní stav - sjednocené testování kvalitativní				
Aplikace	Kfat	Kuat	Kprod	Tk
Aplikace A1	84	132	36	0,50
Aplikace A2	37	22	23	0,82
Aplikace A3	51	68	0	0,75
Aplikace A4	26	34	17	0,51
Arimetický průměr za všechny aplikace	49,5	64	19	0,65

Tab.4 kvalitativní koeficienty testera za časové období 4 měsíce pro původní stav [autor]

Nový stav - produktové týmy kvalitativní					
Aplikace	Kfat	Ksit	Kuat	Kprod	Tk
Aplikace A5	14	2	15	4	0,67
Aplikace A2	52	7	20	16	1,21
Aplikace A6	27	0	4	11	1,80
Aplikace A7	15	0	33	12	0,33
Aplikace A8	8	0	0	12	0,67
Aritmetický průměr za všechny aplikace	23,2	1,8	14,4	11	0,94

Tab.5 kvalitativní koeficienty testera za časové období 4 měsíce pro nový stav

Ve sjednoceném testingu nemá K_{sit} význam, protože SIT testing neexistoval a vnější integrace byla testována v rámci UAT. Z dat je patrné, že v původním stavu bylo nacházeno vyšší množství chyb v oddělení testingu, nicméně i tak bylo chyb nalezených v dalších fázích ve vyšším poměru, než jak tomu bylo u testingu v produktovém týmu.

Aplikace A1 obsahuje jednoznačně nejvyšší množství chyb ve všech fázích. Zajímavé je, že ačkoliv byla zvláštností této aplikace především absence kvalitní dokumentace, chyby promítnuté do těchto koeficientů jsou výhradně chybami v aplikaci, nikoliv v analýze. Produktové týmy mají tu výhodu, že mají jednoho až dva stálé analytiky, kteří mezi sebou sdílejí znalost aplikací. Minimalizuje se tak riziko toho, jak situace vypadala při testování aplikace A1, která byla oficiálně vedena jako projekt, a ačkoliv ji vyvíjel produktový tým, tento tým neobsahoval ani testera ani analytika, neboť v původním stavu byli součástí produktových týmů pouze vývojáři. Vysoká chybovost je tak u této aplikace způsobena z části také tím, jak moc si dokumentaci vývojáři přizpůsobovali a jaké to mělo konečné důsledky.

Problém s režijní náročností kvůli nízké kvalitě dokumentace však byl promítnut u této aplikace do její celkové pracnosti u kvantitativních koeficientů, tudíž za vysokou chybovostí nelze hledat problém pouze v dokumentaci. Tímto problémem byla při testování také velká poziční vzdálenost, kterou musel tester urazit pokaždé, když potřeboval řešit nějaký nález s vývojovým týmem – toto by se nestalo, pokud by byl tester součástí produktového týmu.

Další zajímavostí je aplikace A3, která dopadla tak špatně, že dosud vůbec nešla do produkce, proto tam také nebyla nikdy reportována sebemenší chyba. Není to však kvůli špatnému

testingu, tato aplikace nebyla v zájmu zadavatelů a vlastníků a částečně byla vyvíjena z důvodu technologického vylepšení, které však jejím uživatelům přinášelo podle jejich názoru více práce než užitku. Na samotné chybovosti se tento fakt neprojevil, všechny nálezy v UAT musely být v rozporu s analýzou, kterou zadavatel musel napřed schválit.

V novém stavu jsou pak pouze A5 a A2 testovány na všech čtyřech prostředích, pokud počítáme užívání na produkci za uživatelské testování.

Aplikace A8, jež má podle kvantitativních koeficientů nejmenší váhu, je velmi malá pomocná aplikace sloužící pouze interním zaměstnancům banky a neovlivňující žádné ostatní systémy, tudíž nemá žádné testovací prostředí pro SIT ani UAT.

V koeficientu kvality T_k jsou vidět největší rozdíly v novém stavu. Aplikace A2 a A6 vykazují nejvyšší kvalitu FAT testování. Tyto aplikace vyvíjí produktový tým a v krátké době předal testerovi hlubší znalost databáze i vnitřní funkčnosti těchto aplikací. Z toho důvodu jsou ve fázi FAT z hlediska typu testů v nejvyšší míře prováděny testy typu bílá skříňka a výborně u těchto aplikací zafungoval testing díky spolupráci s produktovým týmem. Z velké části je tomu však i díky bližší spolupráci se zadavatelem, kdy v průběhu vývoje tester produktového týmu přidal do testování prvky agilního vývoje, jako např. iterační ukázky zadavateli již dokončené části aplikace a možnosti podrobení připomínek, případně změn v analýze. K něčemu takovému je však potřeba i osobnost zadavatele, která o kvalitní otestování stojí a je ochotná k užší spolupráci, což se v případě obou dvou aplikací výrazně splnilo, ačkoliv oba mají jiného zadavatele.

Aplikace A7 má pak koeficient kvality nejhorší. Jedním z důvodů je přesně opačná osobnost zadavatele než u aplikací A2 a A6. Zadavatel zde striktně odmítá spolupracovat s produktovým týmem více, než nezbytně musí podle nařízení interních norem. Další příčinu stavu nízké kvality A7 lze vyčíst z kvantitativních koeficientů – ačkoliv je tato aplikace ze všech největší i nejsložitější, testování na ní proběhlo jen na velmi malé části. Tester proto neměl prostor seznámit se s touto aplikací jako s celkem, což vzhledem k její velikosti, složitosti a především vzájemné vnitřní provázanosti představuje jistou míru rizika.

Přestože aritmetický průměr koeficientu kvality u nového stavu je výrazně vyšší než u stavu původního – o 69,15%, je tento údaj velmi orientační. Prvním důvodem je, že nezahrnuje

kvantitativní ukazatele, tudíž nijak nezobrazí, jak dobře byly testovány právě ty aplikace, které byly největší, nejsložitější a nejvíce se toho na nich měnilo. Druhým důvodem je vysoká citlivost výsledku aritmetického průměru T_k na extrémní hodnoty jednotlivých koeficientů kvality. Je proto vhodné kombinovat informativní hodnotu T_k s dalším faktorem, kterým je sledování množství nálezů v konkrétních fázích u jednotlivých aplikací.

Zatímco v původním stavu bylo v UAT testech kromě jednoho případu vždy reportováno výrazně více chyb (vážených závažností) než ve FAT testech, v novém stavu je tomu spíše naopak. V jednom případu z pěti je chyb v UAT výrazně více než ve FAT, ve dvou případech je jich výrazně méně a v jednom případu zhruba stejně. Pátý případ je A8 která jednak UAT nemá a navíc je tato aplikace pouze interní a velmi malá.

Chyby v produkci jsou v původním i novém stavu vzhledem k chybám v UAT v přibližně podobném poměru. Z těchto závěrů již lze vyvozovat, že kvalita testování v produktových týmech byla vyšší než ve sjednoceném testingu. Nelze vyvodit jednoznačný závěr, že v každém případě je sjednocené testování méně kvalitní než produktové týmy, záleží na mnoha faktorech a v prostředí banky se v námi zkoumaném případě produktové týmy s integrovaným testerem v agilním vývoji osvědčily [11].

5.7 Vyhodnocení úspory nákladů

Vraťme se znovu k označení testerů T_1 (sjednocené testování) a T_2 (produktový tým). Tester T_2 otestoval oproti T_1 o 27,62% vyšší množství O_m. Pro náklady jsme již dříve definovali za námi uvažované časové období $T_1 = T_2$. Dalším předpokladem je požadavek důsledného otestování všech aplikací v co nejrannější fázi cyklu vývoje SW - kvůli růstu ceny chyby v závislosti na čase, ve kterém je chyba odhalena. Proto také požadujeme, aby v té nejrannější fázi testování probíhaly FAT testy[36]. Dále jsme definovali, že průměrné měsíční náklady banky na interního testera jsou přibližně 86 000 Kč.

[36] Pokud bychom měli ve vývoji SW jako veškeré testování např. dva cykly po sobě jdoucích uživatelských akceptačních testů, pak by tou nejrannější fází byl první cyklus UAT. V případě naší banky je vždy nejranější fází FAT test, po kterém následuje SIT a poté UAT, tak jako to bylo popsáno v dřívějších kapitolách.

Z těchto předpokladů vyplývá, že změnou organizační struktury, při které přesuneme FAT testery sjednoceného testování do produktových týmů, může dojít ke snížení nákladů na FAT testery o 27,62%. Při tomto snížení bereme v úvahu pouze objem aplikací, který se otestuje. Z výsledků kvalitativních koeficientů však také víme, že se přesunem z T_1 do T_2 navíc zvýšila i kvalita testování. Protože se jedná o orientační ukazatel, hodnotit o kolik přesně kvalita vzrostla by nebylo objektivní, dá se však tvrdit, že kvalita FAT testů vzrostla významně.

Měsíčně tedy banka může ušetřit organizačním přesunem na jediném FAT testerovi 23 753 Kč a zároveň zvýšit kvalitu testování, kterou tato organizační změna přinese. Protože máme celkem 5 produktových týmů, může tato úspora pro banku znamenat ročně přibližně 1 425 000. Protože tester, na jehož výstupech i vstupech bylo měření provedeno, reprezentuje svými výsledky průměrné hodnoty[11], na kterých se shodli ostatní testeři, kteří přešli ze sjednoceného testování do produktových týmů, znamená to, že měření právě jeho hodnot je objektivní a minimálně zkreslující. Přesto **není jednoznačně prokazatelné, že organizační změna ze sjednoceného testování do produktových týmů přinese vždy úspory nákladů**. Cílem této kapitoly bylo prezentovat přístup ke stanovení objektivních koeficientů a způsob, jakým lze objektivně měřit výstup práce testování. **Vždy záleží na prostředí konkrétní banky a teprve až po dlouhodobém používání se ukáže, zda je tento organizační model vhodný**.

6 Návrhy na vylepšení systému test management

V této kapitole budou aplikovány výstupy z předchozích kapitol za účelem zlepšení systému test managementu v bance. Rovněž budou podrobněji probrány obecné metriky jako hlavní ukazatelé výkonnosti a bude provedena analýza metrik, které jsou v bance aktuálně používané. Na závěr kapitoly budou identifikovány pomocí SWOT analýzy klíčové faktory hlavních vylepšení – zavedení objektivních ukazatelů výkonnosti a zavedení procesu výpočtu ceny testování.

6.1 Model zralosti softwarových procesů CMMI

Existuje vyšší množství modelů kvality organizace. Zde je použit Stupňovitý model zralosti CMMI, konkrétně CCMI-SW, který specifikuje úroveň zralosti procesu vývoje SW. Alternativou by byl standard ISO 9000:2001, který je však obecnější a není určen primárně k vývoji SW. Další alternativou je standard ISO 9001, který je však méně komplexní – narozdíl od modelu CMMI definuje ISO 9001 pouze cíle, zatímco CMMI definuje i stupně zralosti, tudíž vede k soustavnému zlepšování.

6.1.1 Stupně zralosti modelu CMMI

Model CMMI definuje 5 úrovní zralosti procesu [1]:

- **Úvodní (initial)** - Chaotické a nahodilé procesy, nepředvídatelné náklady a ukazatelé kvality.

- **Řízená (managed)** - Řízení na úrovni projektu, procesy jsou implementovány podle popisu a plánu.

- **Definovaná (defined)** - Procesní řízení na úrovni organizace, procesy jsou definovány pomocí standardů, nástrojů a metod. Procesy jsou přizpůsobovány podle typu projektu.

- **Kvantitativně řízená (quantitatively managed)** - Procesy jsou řízeny na základě metrik, probíhá vyhodnocování pomocí metrik.

- **Optimalizující (Optimizing)** - Neustálé vylepšování procesů.

Doposud v analyzované bance dosahovala procesní úroveň stupně zralosti Řízená. Řízení probíhalo na úrovni jednotlivých projektů. V současné době je částečně nastavena úroveň Definovaná, proces testování je již definován pomocí standardu platného pro celou organizaci. Pomocí vhodného užití metrik lze dosáhnout úrovně Kvantitativně řízená, tato

úroveň však stale není v bance plně implementována – metriky nejsou ustálené, dochází k jejich neustálým změnám a dosažení tohoto stupně úrovně zralosti by bylo možné, pokud by se při nastavování metrik použil komplexní přístup zvolený v kapitole 5.6 (stanovení výnosů), který cíleně identifikuje všechny podstatné faktory ovlivňující proces testování.

6.1.2 Metriky obecné

V této kapitole budou popsány některé základní obecné metriky, které mohou sloužit ke kvalitnímu měření a vyhodnocování KPI.

Efektivita odstraňování defektů

Základní metrikou pro porovnání efektivity testování během testovací fáze je poměr počtu defektů nalezených během testovací fáze a defektů nalezených v produkčním prostředí. Podle [1] se tato metrika nazývá **DRE** (Defect Removal Efectiveness) a platí pro ni jednoduchý vzorec

DRE = (Chyby nalezené během fáze testování / chyby nalezené v produkci) * 100 %

Takto jednoduše zjistíme pouze množství nalezených chyb, je vhodné přidáním vah zohlednit závažnost chyb (severitu). DRE má přímou souvislost s CMMI, čím vyšší úroveň zralosti, tím vyšší je efektivita odstraňování defektů – odhady jak moc se liší DRE v závislosti na CMMI se výrazně liší u různých organizací [1].

Efektivita testovacích scénářů

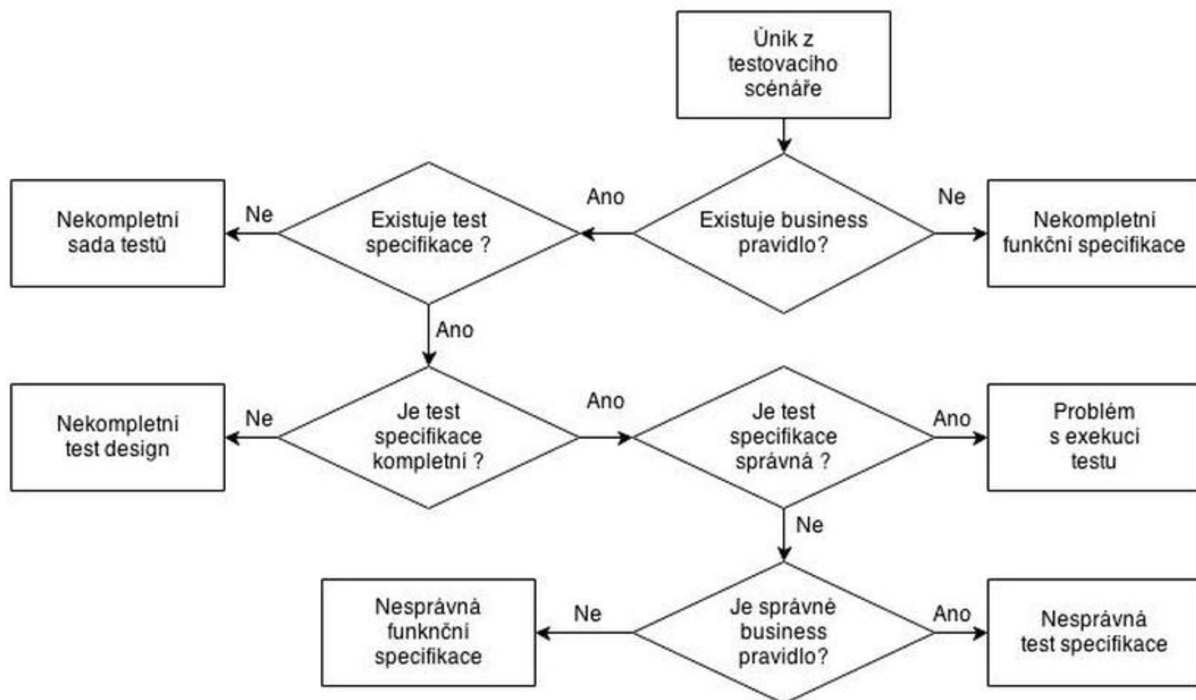
Podle [2] je metrikou pro efektivitu testovacího scénáře jednotka **TCE**, pro kterou platí

$$TCE = N_{tc} / N_{tot} * 100\ \%$$

Kde N_{tc} značí počet chyb nalezených pomocí testovacích scénářů a N_{tot} celkový počet všech chyb.

TCE (Test Case Effectiveness) nám tedy udává, kolik procent chyb bylo nalezeno díky testovaní podle postupů popsaných v testovacích scénářích, a kolik procent jsou defekty, které byly nalezeny postupem, který není popsaný ve scénářích.

Diagram zobrazující **doporučený postup pro vylepšení TCE**



Obr. 9 Důvody úniků z testovacích scénářů, přeloženo z [2]

Pomocí tohoto postupu lze identifikovat důvody, proč nalezené chyby nebyly nalezeny postupem, který popisují testovací scénáře a učinit podle toho vybraná opatření.

Přesnost odhadů pracnosti úkolů

Pro zpřesnění odhadu pracnosti úkolů nejen v procesu testování se používá podle [1] metrika **EEA** (Effort Estimation Accuracy), kde platí

EEA = (Skutečný čas / odhadnutý čas) * 100 %

Při používání této metriky je třeba nastavit způsob zamezení zneužití záměrně chybnými odhady. Rovněž je třeba zajistit, aby v případě příliš vysokých odhadů času nebyl měřený tester demotivován negativním vyhodnocením za vyšší efektivitu práce tím, že splnil daný úkol dříve.

6.1.3 Popis metrik používaných v bance pro hodnocení výkonnosti testování

Aktuálně jsou v analyzované bance zavedeny metriky popsané v této kapitole. Výstupem těchto metrik jsou ukazatele platné pro všechny členy vývoje aplikací v bance, kromě testerů platí i pro analytiky a vývojáře. Proces zavádění metrik trvá v bance již několik let a stále ještě není hotov a jsou nastavovány nové ukazatele [11]. Hlavním důvodem je nastavení objektivního měření výkonnosti a zároveň omezení vlivu faktorů, které mohou testeři, analytici a vývojáři jen těžko ovlivnit.

V bance jsou aktuálně používané tyto metriky [11] :

- **Termín** - hodnocen po každém hromadném nasazování aplikací. Termín je odvislý od toho tolik úkolů na vývoj aplikací stihlo IT dodat do plánovaného termínu. Pokud se stihlo dodat vše, je to ohodnoceno jako 100 % a znamená to „dobrá práce“. Vyšší hodnocení než 100 % je uděleno za flexibilní schopnost rychle dodávat opravené aplikace v případě poruch, výpadků nebo jiných neočekávaných událostí

  Termín = (počet dokončených úkolů pro daný termín /
  počet naplánovaných úkolů pro daný termín) * 100

- **Odhady** - hodnoceny po každém hromadném nasazování aplikací. Odhady měří přesnost původně odhadované doby, za kterou zaměstnanec v době známého zadání stanovil, jak dlouho mu bude daný úkol trvat, a zaměstnancem vykázanou dobou na tento úkol po jeho realizaci.

Odhady = Absolutní hodnota (1 - (celkově odhadnuto) / (celkově vykázáno)) * 100

- **Kvalita** - ukazatel množství chyb nalezených ve fázích FAT + SIT oproti chybám nalezených ve fázi UAT + na produkci

Kvalita = (UAT / (FAT + SIT + UAT)) * 100

Chyby jsou vážené jejich závažností:

  - Kritické chyby - 3
  - Střední chyby - 2
  - Drobné chyby - 1

- **Zpětná vazba** - výpočet pro tento ukazatel není definován, jedná se o vyjádření spokojenosti od obchodníků, zadavatelů a dalších útvarů na dodávky aplikací.

6.1.4 Vylepšení metrik používaných v bance pro hodnocení výkonnosti testování

Metriky používané v bance nejsou ustálené a často se mění. Aplikací postupu stanovování koeficientů pro ukazatele výnosů testování lze najít vhodnější metriky pro každý posuzovaný proces. V této kapitole budou analyzovány aktuálně používané ukazatelé a jejich nedostatky.

Termín - Aktuálně je tento ukazatel nastaven tak, že nezohledňuje žádné okolní vlivy. Nemožnost nasazení požadavku nemusí být nutně zapříčiněna IT. Způsob, jakým je identifikována „flexibilní schopnost rychle dodávat opravené aplikace v případě poruch, výpadků nebo jiných neočekávaných událostí“ není definovaný.

Odhady - Tento ukazatel měří schopnost odhadovat čas velmi dobře v případě, že zaměstnanec odhadující jak dlouho mu bude úkol trvat bude odhadovat svědomitě. Problém nastává v případě, že zaměstnanec odhadne příliš dlouhou dobu, úkol stihne mnohem dříve a poté vykáže čas, který tímto úkolem nestrávil tak, aby byl vyhodnocen tento ukazatel co nejlépe. Dobrým řešením by mohlo být hodnotit pouze překročení odhadnutého času, nikoliv mimořádně rychle zvládnutý úkol.

Kvalita - Podobný ukazatel byl v této práci řešen jako kvalitativní koeficient. Prvním problémem je identifikace možnosti, že chyba mohla být nalezena, ne vždy je to možné. Dále mají chyby 3 závažnosti, ale v některých reportovacích nástrojích banky jsou definovány jiná množství závažnosti. Bylo by vhodné stanovit způsob přepočtu vah. Rovněž tento koeficient nerozpozná, zda se jednalo o chybu v aplikace nebo jen v dokumentaci.

Zpětná vazba - Pro tento ukazatel by měl být stanoven jednotný způsob vyjadřování spokojenosti a identifikované parametry pro hodnocení této spokojenosti tak, aby se s nimi vždy seznámil hodnotitel i hodnocený.

6.2 SWOT analýza navrhovaných vylepšení

Strengths:

Protože způsob nastavení KPI kombinací mnoha metrik s důkladným promyšlením všech okolností bere v potaz vysoké množství faktorů působících na proces testování, je hlavní silnou stránkou tohoto vyhodnocování výkonnosti objektivita a komplexnost. Silnou stránkou všech řešení, zejména pak výpočtu vnitřní ceny testování, je optimalizace procesu testování a s ní spojená úspora nákladů.

Weaknesses:

Mezi slabé stránky nastavení měření a vyhodnocování KPI a vnitřní ceny testování patří především vyšší pracnost spojená s neustálou kontrolou aktuální platnosti všech metrik, cen chyb, nenasazení aplikací a dalších proměnných. Dále nastavení procesu pro stanovení hodnot koeficientů kvalitativních a kvantitativních ukazatelů výkonnosti testera je velmi závislé na lidském faktoru, objektivně vyhodnotit poměr kvality několika dokumentací vyžaduje vysoký stupeň kvalifikace a hlubokou znalost všech aplikací osoby, která tyto koeficienty stanovuje.

Opportunities:

Hlavní příležitostí je posunout na vyšší úroveň stupňovitý model zralosti stanovením měřitelných ukazatelů a tento proces aplikovat na objektivní vyhodnocování výkonnosti u vývoje zcela nových aplikací v případě interních i externích dodavatelů.

Threads:

Hrozbou pro stanovení vnitřní ceny testování je možná nedostupnost informací o cenách oprav chyb, nenasazení aplikace a dalších proměnných. Hrozba pro kvalitní stanovení měřitelných ukazatelů procesu testování je záměrné přecenění nebo podcenění hodnot těchto ukazatelů v případě nedostatečné revize a kontroly více účastníků procesu testování.

7 Závěr

Teoretická část práce byla zaměřena na studium procesů testování bankovních aplikací s cílem odstranění chyb. Ve vazbě na podrobnou analýzu reálných bankovních procesů bylo provedeno porovnání vlastních poznatků s klíčovými informacemi získanými z odborných publikací a expertních sdělení. V práci byly použity zejména vědecké metody analýzy, deskripce, komparace a syntézy.

Byly diskutovány základní procesní modely provádění testování bankovních aplikací. Na základě toho byly identifikovány dvě dominantní varianty organizace a řízení procesů testování. První variantou byl samostatný útvar provádějící testování, který spolupracuje s dalšími útvary banky při testování aplikací. Druhou variantou bylo vytvoření týmů sestavených účelově pro testování dané aplikace ze zaměstnanců s příslušnou odborností i vztahem k aplikaci, kdy členem tohoto týmu je také tester.

Jedním z cílů práce bylo najít a vhodným způsobem setřídit informace a fakta týkající se procesu testování a jeho optimalizace. Dále následovalo nalezením vzájemných vazeb a souvislostí. Metoda deskripce byla základem pro návaznou syntézu a formulaci odborných poznatků. Analýza a navazující syntéza poznatků získaných v jednotlivých etapách práce a jejich správná interpretace umožnila nalézt optimální variantu řízení procesů testování.

Těžištěm aplikační části práce je případová studie porovnání a ocenění vybraných procesů testování na reálných datech, která jsou presentovaná tak aby nedošlo k porušení práv banky, která je poskytla.

Klíčovým výsledkem diplomní práce je navržená a ověřená metoda oceňování a ekonomického hodnocení procesu testování bankovních aplikací. V této oblasti je velmi důležitý způsob hodnocení a použití metrik.

Dalším významným přínosem jsou výsledky z ekonomického i věcného porovnání dvou základních modelů organizace a řízení procesu testování.

Souhrnně je možno konstatovat, že výsledky práce potvrzují, z pohledu věcného i ekonomického, správnost současného trendu přechodu od ad-hoc realizace jednotlivých projektů k systematickému procesnímu řízení, kdy jednotlivé projekty jsou integrální součástí řízených procesů.

8 Seznam použité literatury

[1] KAN, S. *Metrics and models in software quality engineering*. Vyd. 1. Boston: Addison-Wesley, 2003. ISBN 02-017-2915-6.

[2] Chernak, Y., *Validating and improving test-case effectiveness*, Software, IEEE , vol.18, no.1, pp.81,86, Jan/Feb 2001 doi: 10.1109/52.903172

[3] PAGE, Alan, Ken JOHNSTON a Bj ROLLISON. *How we test software at Microsoft*. Redmond, Wash.: Microsoft, c2009, xx, 423 p. ISBN 07-356-2425-9.

[4] BUNDSCHUH, Manfred a Carol DEKKERS. *The IT measurement compendium: estimating and benchmarking success with functional size measurement* [online]. Berlin: Springer, c2008 [cit. 2014-03-23]. ISBN 35-406-8187-6.

[5] PATTON, Ron. *Testování softwaru*. Vyd. 1. Praha: Computer Press, 2002, xiv, 313 s. ISBN 80-722-6636-5.

[6] Vienneau, R.L., *"The cost of testing software," Reliability and Maintainability Symposium, 1991. Proceedings., Annual , vol., no., pp.423,427, 29-31 Jan*

1991
doi: 10.1109/ARMS.1991.154473

[7] ZRALÝ, Martin. *Přednášky předmětu AD1M16CTR*. Dostupné pro studenty z oficiálních stránek katedry https://ekonom.feld.cvut.cz Praha, 2011.

[8] GROOD, Derk-Jan de. *TestGoal: result-driven testing*. Leiden, The Netherlands. ISBN 9783540788294-.

[9] BORBA, Paulo. *Testing techniques in software engineering: second Pernambuco Summer School on Software Engineering, PSSE 2007, Recife, Brazil, December 3-7, 2007 : revised lectures*. Berlin: Springer, 2010, viii, 312 p. Lecture notes in computer science, 6153. ISBN 36-421-4334-2.

[10] Louridas, P., *Test Management*, Software, IEEE , vol.28, no.5, pp.86,91, Sept.-Oct. 2011
doi: 10.1109/MS.2011.111

[11] Expertní sdělení - banka

[12] Integrovaný portál Ministerstva práce a sociálních věcí: Průměrné mzdy podle lokalit a zařazení. [online]. [cit. 2014-03-07]. Dostupné z: http://portal.mpsv.cz/sz/stat/vydelky/pra/ps?stat=2000000000070&obdobi=4&rok=2012&uzemi=19&send=1

[13] Český statistický úřad: Mzdy IT odborníků v České republice. [online]. [cit. 2014-05-07]. Dostupné z: http://www.czso.cz/csu/redakce.nsf/i/mzdy_it_odborniku_v_ceske_republice

[15] Simon, D.; Simon, F., *"Integrating Test and Risk Management," Quality of Information and Communications Technology (QUATIC), 2012 Eighth International Conference on the* , vol., no., pp.97,102, 3-6 Sept. 2012
doi: 10.1109/QUATIC.2012.34

Seznam zkratek

IT	Information technology
ICT	Information and communications technology
PC	Personal computer
QR	Quick response
CMMI	Capability maturity model integration
SWOT	Strengths, weaknesses, opportunities and threats
IEEE	Institute of electrical and electronics engineers
SW	Software
HW	Hardware
QA	Quality assurance
DEV	Development
FAT	Functional acceptance testing
UAT	User acceptance testing
SIT	System acceptance testing
PROD	Production
RUP	Rational unified process
ISTQB	International software testing qualifications board
CRM	Customer relationship management
DWH	Data warehouse
SQL	Structured query language
PLSQL	Procedural langauage structured query language
ČSÚ	Český statistický úřad
KPI	Key performance indicators

Slovník pojmů

Vývojář	osoba vyvíjející aplikace, programátor
Programátor	osoba vyvíjející aplikace, vývojář
Analytik	IT analytik vytvářející analýzu = dokumentaci aplikace
Tester	Osoba provádějící testy aplikace
FAT tester	Osoba provádějící funkční akceptační testování
SIT tester	Osoba provádějící systémové integrační testování
UAT tester	Osoba provádějící uživatelské akceptační testování
Unit test	Jednotkový test prováděný vývojářem
Produkce	Produkční prostředí banky, „live", živé prostředí
Nasazení	nahrání hotové aplikace nebo systému na určité prostředí
Testovací prostředí	Prostředí určené pouze pro testovací účely
Analýza	Dokumentace aplikace, dokument popisující aplikaci
Dokumentace	Analýza aplikace, dokument popisující aplikaci
Zadavatel	Osoba zadávající požadavky na IT na změnu aplikace
Vlastník	Osoba zodpovědná za vývoj a správnou funkčnost aplikace
Retail	Oblast maloobchodní klientely
Korporát	Oblast velkoobchodní klientely
Solus	Databáze neplatičů
Release notes	Poznámky k vydané verzi aplikace
Agilní vývoj	Moderní způsob vývoje SW, flexibilní, zaměřený na zákazníka
Manažer dodávky	„Delivery manažer", osoba zodpovědná za dodání aplikace řídící celý proces analýzy, vývoje a testování aplikace
Produktový tým	Tým složený z vývojářů, analytiků a testerů mající na starost určitou oblast z okruhu aplikací, dodávající určité produkty. Jeho výstupem jsou produkty – aplikace, které jsou posléze nasazeny (předány) dále.

Seznam tabulek

Tab.1 Pokrytí parametrů testování koeficienty a pravidly…………………………………..….78
Tab.2 kvantitativní koeficienty testera za časové období 4 měsíce pro původní stav…..…...85
Tab.3 kvantitativní koeficienty testera za časové období 4 měsíce pro nový stav………….86
Tab.4 kvalitativní koeficienty testera za časové období 4 měsíce pro původní stav……..….87
Tab.5 kvalitativní koeficienty testera za časové období 4 měsíce pro nový stav……....…...88

Seznam obrázků

Obr.1 Pravděpodobnost opravy chyby v závislosti na době jejího nalezení……………..….…..31
Obr.2 Rostoucí náklady na opravu chyby podle okamžiku nalezení……………………..….….32
Obr.3 Poměry nákladů na opravu chyby podle fáze…………………………………..…..…..32
Obr.4 původní model nasazování aplikace……………………………………………...……40
Obr.5 testing po zavedení testovacího oddělení……………………………………...……41
Obr.6 postupné nasazování aplikace v novém testingu zaměřeném na agilní vývoj…...……47
Obr.7 Typy testů v novém testingu zaměřeném na agilní vývoj……………………….………48
Obr.8 Časová škála běhu aplikace……………………………………………….……………..52
Obr.9 Důvody úniků z testovacích scénářů…………………………………………………….94

Seznam příloh

[CD] externí soubor: Diplomova_Prace.pdf
[CD] externí soubor: Zanonymizovana_Test_Analyza.xls
[CD] externí soubor: Zanonymizovany_Test_Report.doc
[CD] externí soubor: Komentovana_Reserse.doc