Vysoká škola ekonomická v Praze

Fakulta informatiky a statistiky

Katedra informačních technologií

Student	: **Lukáš Michálek**
Vedoucí bakalářské práce	: **Ing. Alena Buchalcevová PhD.**
Recenzent bakalářské práce	: **Ing. Iva Stanovská**

TÉMA BAKALÁŘSKÉ PRÁCE

Použití metod testování softwaru v praxi

ROK : 2006

Prohlášení

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a že jsem uvedl všechny použité prameny a literaturu, ze kterých jsem čerpal.

V Praze dne …….. ....
Podpis

Poděkování

Rád bych touto cestou poděloval paní Ing. Aleně Buchalcevové PhD. za čas, který mi věnovala a za cenné rady a znalosti, kterými mě obohatila.

Anotace

Cílem této práce je položit čtenáři znalostní základ v oblasti testování softwaru a to takovým způsobem, aby znalosti zde načerpané byly co nejsnáze a nejefektivněji použitelné v praxi.

První část práce obeznamuje s teoretickým základem testování software, na kterém jsou následně stavěny praktické přístupy. Teoretický základ zde spočívá v definování a objasnění základních pojmů v oblasti testování softwaru, v popisu různých přístupů k vývoji softwaru, v kategorizaci a charakterizování metod testování a také rolí, které v rámci testování vystupují.

Druhá část práce popisuje činnosti, které v rámci testování softwaru probíhají, a také jak k těmto činnostem přistupovat, aby testování bylo co nejefektivnější. Jedná se o činnosti jako testování specifikace softwaru, plánování testů, návrh testových případů a ohlašování nalezených chyb. Na závěr následuje kapitola, která dříve popsané činnosti rekapituluje a naznačuje uspořádání, jak na sebe logicky navazují.

Annotation

The aim of this Bachelor thesis is providing a basic knowledge ground in the field of software testing in a way, which would allow using this knowledge in real projects in the most effective way.

The first part of the thesis describes the theoretical basics of software testing which are followed by practical approaches based on them. The theoretical basics consist of the definition of basic terminology, description of various approaches towards software development, categories and characteristics of testing methods and also roles, which take part in testing.

The second part of the thesis describes activities which are included in software testing and also the way how to run these activities in the most effective way. Activities described in this part are software specification testing, test planning, designing test-cases and errors reporting. Finally there is a chapter summarizing these activities and suggesting the way they can be logically ordered.

Obsah:

1 Úvod ... 9
1.1 Vymezení tématu práce ... 9
1.2 Důvod výběru tématu ... 9
1.3 Cíl páce ... 9
1.4 Předpoklady a omezení práce ... 9
2 Obecná východiska ... 10
2.1 Základní pojmy v oblasti testování software ... 10
2.1.1 Chyba ... 10
2.1.2 Správnost a přesnost ... 10
2.1.3 Verifikace a validace ... 11
2.1.4 Kvalita a spolehlivost ... 11
2.1.5 Testování a zajišťování kvality ... 11
2.2 Okénko do oblasti vývoje softwaru ... 11
2.2.1 Model velkého třesku ... 11
2.2.2 Model „Programuj a opravuj“ ... 12
2.2.3 Model vodopádu ... 12
2.2.4 Spirálový model ... 12
2.3 Role testování v rámci metodiky RUP [RUP] ... 13
3 Úvod do testování softwaru ... 15
3.1 Techniky testování podle způsobu provedení testů ... 15
3.1.1 Testování černé skříňky a bílé skříňky ... 15
3.1.2 Manuální a automatizované testování ... 16
3.1.3 Statické a dynamické testování ... 16
3.1.4 Testování splněním a selháním ... 16
3.2 Techniky testování podle úrovně vývoje ... 17
3.2.1 Testování v průběhu vývoje (Developer Testing) ... 17
3.2.2 Jednotkové testy (Unit Testing) ... 17
3.2.3 Integrační testy (Integration Testing) ... 17
3.2.4 Systémové testy (System Testing) ... 17
3.2.5 Akceptační testy (Acceptance Testing) ... 17
3.2.6 Nezávislé testování (Independent Testing) ... 18
3.2.7 Independent StakeholderTtesting ... 18
3.3 Testování podle dimenze kvality [RUP] ... 18
3.3.1 Funkčnost (**F**unctionality) ... 18
3.3.2 Použitelnost (**U**sability) ... 19
3.3.3 Spolehlivost (**R**eliability) ... 19
3.3.4 Výkon (**P**erformance) ... 20
3.3.5 Podporovatelnost (**S**upportability) ... 20
3.3.6 Ostatní (nefunkcionální) dimenze ... 21
4 Role v rámci testování ... 22
4.1 Manažer testů (Test Manager) ... 22
4.1.1 Požadované schopnosti ... 22
4.1.2 Odpovědnosti ... 22
4.1.3 Zaměření při vývoji na testovatelnost ... 22
4.2 Analytik testů (Test Analyst) ... 23
4.2.1 Požadované schopnosti ... 23
4.2.2 Odpovědnosti ... 23
4.3 Návrhář testů (Test Designer) ... 23

4.3.1 Požadované schopnosti ... 23
4.3.2 Odpovědnosti ... 24
4.4 Tester (Tester) ... 24
4.4.1 Požadované schopnosti ... 24
4.4.2 Odpovědnosti ... 24
4.5 Poznámka k rolím ... 25
5 Specifikace softwaru ... 26
5.1 Revize a zkoumání specifikace ... 26
5.1.1 Kde ve specifikaci hledat chyby? ... 27
5.1.2 Testování úplnosti specifikace ... 28
6 Plánování testů ... 29
6.1 Náležitosti plánu testů ... 29
6.1.1 Co se bude testovat a jakých výsledků se má dosáhnout ... 29
6.1.2 Lidé, místa a věci ... 29
6.1.3 Definice a názvosloví ... 30
6.1.4 Povinnosti a odpovědnost jednotlivých skupin ... 30
6.1.5 Co se bude testovat ... 31
6.1.6 Fáze testování ... 31
6.1.7 Strategie testování ... 31
6.1.8 Požadavky na prostředí ... 32
6.1.9 Pověření konkrétních osob ... 32
6.1.10 Časový plán testů ... 32
6.1.11 Zprávy o chybách ... 33
6.1.12 Rizika a možná omezení ... 33
7 Návrh testových případů ... 34
7.1 Optimální objem testů ... 34
7.2 Třídy ekvivalentních případů ... 35
7.3 Jak tvořit testové případy ... 36
7.3.1 Testování dat ... 36
7.3.2 Testování operací ... 38
8 Ohlašování chyb ... 42
8.1 Náprava chyb ... 42
8.2 Jak oznamovat chyby ... 43
8.2.1 Relevance ... 43
8.2.2 Jednotlivost ... 43
8.2.3 Jasná identifikace a reprodukovatelnost ... 43
8.3 Priorita a závažnost chyb ... 44
9 Postup testování ... 46
9.1 Testování (revize) specifikace ... 46
9.2 Strategie testování ... 47
9.3 Plán testů ... 47
9.4 Návrh a implementace testových případů ... 47
9.4.1 Testování splněním ... 47
9.4.2 Testování selháním ... 48
9.5 Zprávy o provedení testů ... 48
9.5.1 Zprávy o chybách ... 48
9.5.2 Závěr z testování ... 48
9.6 Opakované provádění testů ... 48
10 Závěr ... 49
10.1 Závěrečné shrnutí ... 49

10.2 Využití práce v praxi 50
11 Terminologický slovník 51
12 Seznam použité literatury 53

1 Úvod

1.1 Vymezení tématu práce

Tématem této práce je Použití metod testování softwaru v praxi. Zaměřuji se na obecná východiska, kde definuji základní pojmy, na možnosti klasifikace metod testování softwaru a poté bude následovat praktičtější pohled na testování, který zahrnuje plánování testů, návrh testových případů a různé rady a návody pro testování.

1.2 Důvod výběru tématu

Již několik měsíců mám to štěstí pracovat jako vedoucí testovacího týmu pro firmu Bellman Group s. r. o., která působí v oblasti vývoje software. Jsem zastáncem názoru, že teorie se neobejde bez praxe a praxe zase bez teorie. Zpracování tohoto tématu v rámci své Bakalářské práce mi umožní obě tyto poloviny propojit.

1.3 Cíl páce

Cílem této práce je jednak obeznámit čtenáře s teoretický základem testování software a jednak poskytnout pohled na testování poněkud prakticky, aby znalosti zde načerpané byly co nejsnáze využitelné v praxi.

Při psaní této práce budu vycházet z teoretických poznatků a závěrů týkajících se metod testování software a budu se snažit popsat a zhodnotit aplikaci těchto metod, jejich výhody a omezení při testování.

Myšlenky a informace budu čerpat z metodiky Rational Unified Process, z literatury a z osobních zkušeností a názorů.

Práci mohou využít zejména začínající softwaroví testeři k tomu, aby získali správný náhled na testování software a aby jejich práce byla již od začátku co nejvíce efektivní.

1.4 Předpoklady a omezení práce

Mezi předpoklady, které mám pro psaní práce na téma testování software řadím zejména praktické zkušenosti s testováním softwaru a fakt, že vývoj informačních systémů mě oslovil a rád bych v této oblasti pracoval i po ukončení studia.

2 Obecná východiska

2.1 Základní pojmy v oblasti testování software

Než se dostaneme k samotné definici testování software, je potřeba se seznámit se základními pojmy, se kterými budeme pracovat, a které nám pomohou samotnou definici sestavit.

2.1.1 Chyba

Chybou v softwaru můžeme nazvat skutečnost, že program vykazuje alespoň jeden z následujících příznaků *[Patton]*:

- software nedělá něco, co by dle specifikace produktu dělat měl,
- software dělá něco, co by podle specifikace dělat neměl,
- software dělá něco, o čem se specifikace nezmiňuje,
- software dělá něco, o čem se specifikace nezmiňuje, ale zmiňovat by se měla[1],
- software je obtížně srozumitelný, pomalý, těžko se s ním pracuje nebo vykazuje jiné vlastnosti, které mohou bránit jeho řádnému a efektivnímu použití.

2.1.2 Správnost a přesnost

Můžeme říci, že program pracuje správě, pokud se svými vstupy provádí správné operace – tedy „dělá to, co má dělat“ a to bez ohledu na přesnost výstupů.

Můžeme říci že program pracuje přesně, pokud po provedení nějaké operace poskytne přesné výstupy vzhledem k této operaci a to bez ohledu na to, jestli provedl správnou operaci.

Uveďme si rozdíl mezi přesností a správností na příkladu kalkulačky. Pokud zadáme například 10 x 2 = a dostaneme výsledek 20,01 – program pracuje správně, ale nepřesně. Použil sice správnou operaci - násobení, ale výsledek není přesný. Pokud by program vrátil jako výsledek například 12, znamená to, že pracuje nesprávně (použil operaci sčítání místo násobení), ale přesně (vzhledem k operaci sčítání je tento výsledek přesný). Pokud obdržíme jako výsledek číslo 20, program pracuje přesně i správně.

[1] Tento bod je jistě poněkud sporný. Z formálního hlediska se jedná o chybu, avšak v praxi se může vyskytnout situace, kdy z důvodu časového nátlaku není možné přepracovávat specifikaci. Potom může implementace bez specifikace zachránit situaci (ale bohužel také ještě zhoršit)

2.1.3 Verifikace a validace

Verifikací rozumíme ověření (kontrolu), zda program přesně odpovídá jeho specifikaci. Zatím co při Validaci si klademe otázku, zda program vyhovuje požadavkům uživatele a dalších zájmových skupin.

2.1.4 Kvalita a spolehlivost

Software je možné prohlásit za spolehlivý, pokud je stabilní, důvěryhodný a nevykazuje žádné chyby technického charakteru.

Pojem Kvalita software je nadřazený pojmu spolehlivost. Kvalita software má několik dimenzí, z nichž jedna je právě spolehlivost. Za ostatní dimenze kvality se obvykle považují funkčnost, použitelnost, výkon a podporovatelnost. Více se o dimenzích kvality pojednává v kapitole 3.3.

2.1.5 Testování a zajišťování kvality

Testování software můžeme definovat jako průběžnou činnost, která může probíhat po celé období a ve všech fázích vývoje software a je zaměřena na vyhledávání chyb, jejich identifikaci a zajištění jejich nápravy a to co nejdříve.

Zajišťování kvality spočívá v samotném vytváření a prosazování standardů a metod, které vedou ke zvyšování kvality software.

2.2 Okénko do oblasti vývoje softwaru

Vývoj softwaru jistě není věcí jednoduchou. Jedná se o proces velmi náročný na zdroje, plánování, koordinaci a mnohé další aspekty. Existuje celá řada tvrdých či měkkých metodik, které formalizovaným způsobem více či méně detailně popisují a dávají návod, jak efektivně řídit projekt vývoje softwaru. Tyto metodiky představují (většinou) velmi odborný pohled na vývoj a jsou mnohdy jistě velmi mocné. Je však nutno si uvědomit, že pro každou roli v rámci projektu (ať už se jedná o analytika, manažera projektu či testera) může být významný poněkud jiný pohled.

Z pohledu softwarového testera není jistě tak důležité vědět, z jakých teoretických principů a poznatků se vychází při vývoji. Mnohem důležitější je, jaký byl skutečný praktický přístup při vývoji. Tady se dostáváme k poněkud praktičtějšímu pohledu na modely vývoje software. Zejména u prvních dvou modelů, které zde uvádím, je zřejmé, že by se jen těžko mohly dostat do odborných metodik, musíme však počítat s tím, že tu a tam k jejich aplikaci přece jen dojde. Jedná se o následující čtyři modely vývoje software *[Patton]*:

2.2.1 Model velkého třesku

Model velkého třesku by se jistě dal velmi výstižně nazvat improvizací. Snad jedinou výhodou tohoto přístupu je jeho snadnost. Vývojový tým se prostě vrhne do programování a programuje a

možná ani přesně neví, jak má výsledek vypadat. Tento model nezahrnuje žádnou analýzu, plánování, systematické testování, nic takového. Veškeré úsilí je věnováno samotnému psaní programového kódu.

Tento přístup ke psaní softwarru je, jak vidíme, naprosto neprofesionální a pro seriózní vývoj zřejmě nepoužitelný, a to především proto, že se do poslední chvíle přesně neví, jak produkt bude vypadat.

2.2.2 *Model „Programuj a opravuj“*

Tento model jistě znamená oproti předchozímu krok vpřed, ale tím netvrdím, že by se dal zařadit mezi efektivní modely.

Proces vývoje pomocí tohoto modelu začíná zpravidla neformálním a ne příliš detailním návrhem, kterého se vývojový tým po zbytek vývojového cyklu drží. Samotná tvorba spočívá v jakémsi cyklu – kus kódu naprogramovat a otestovat. Pokud se objeví chyby, opravit chyby a znova otestovat atd.

Nedostatky modelu „Programu a opravuj“ spočívají zejména v nedostatečně detailním návrhu a v nedostatečné dokumentaci. Přesto však by mohl být vhodný pro projekty s krátkou dobou použitelnosti, pro vývoj „na poslední chvíli“ nebo pro projekty sloužící jakémusi jednorázovému použití. Tento model je však tak přirozený, že i v sebelépe plánovaném projektu občas na nižší úrovni k vývoji typu „Programuj a opravuj“ dojde.

2.2.3 *Model vodopádu*

Model vodopádu je prvním modelem z těchto čtyř zmiňovaných, o kterém bychom mohli prohlásit, že je systematický. Skládá se z pěti fází – Nápad (specifikování požadavků, zadání), Analýza, Návrh, Vývoj, Testování.

Výhodou tohoto přístupu je naprosto jasná a detailní specifikace. Při samotném vývoji a testování není pochyb o žádném detailu, neboť vše bylo předem dohodnuto a přesně specifikováno[2]. Výsledek bývá velmi kvalitní. Nevýhody z pohledu testování lze spatřovat ve skutečnosti, že testování probíhá pouze na konci a také v tom, že specifikace produktu je velmi náchylná ke vzniku chyb.

2.2.4 *Spirálový model*

Spirálový model byl popsán v roce 1986 a od té doby se ukázalo, že se jedná o velmi efektivní způsob vývoje software. Spirálový (iterativní) model je jedním z principů, ze kterého vychází i metodika Rational Unified Process.

Základní myšlenkou modelu je to, že je téměř nemožné na samém začátku definovat celý software podrobně do nejmenších detailů. Začínáme tedy u základních obecných funkčností, po jejich zvládnutí jako by „nabalujeme“ (například na základě připomínek a návrhů zákazníků) další. Při tom

[2] To je ovšem velmi náročné a často zde mohou vznikat ve specifikaci chyby a mezery.

vycházíme ze základních činností, které se postupně opakují v jednotlivých vrstvách spirály vývoje (iteracích). Tyto jednotlivé fáze nemusí být vždy stejné. Mohou se lišit povahou vyvíjeného produktu i fází projektu, kde je přístup použit. Například se může jednat to tyto fáze [Patton]:

- určení cílů, alternativ a omezení,
- rozpoznání a řešení rizik,
- vyhodnocení alternativ,
- vývoj a testování aktuální úrovně,
- plánování další úrovně,
- rozhodnutí o postupu na další úroveň.

Tyto modely se samozřejmě nemusí vždy vyskytovat v ryzí formě, mohou se různě kombinovat a do sebe vnořovat a to jak mezi sebou, tak se dají použít i ve vývoji podle některé z odborných metodik. Mohou být i klíčovým přístupem v rámci odborné metodiky. Například Rational Unified Process je značně založen na spirálovém (iterativním) modelu – ten je stěžejním principem každé ze čtyř jejích základních etap.

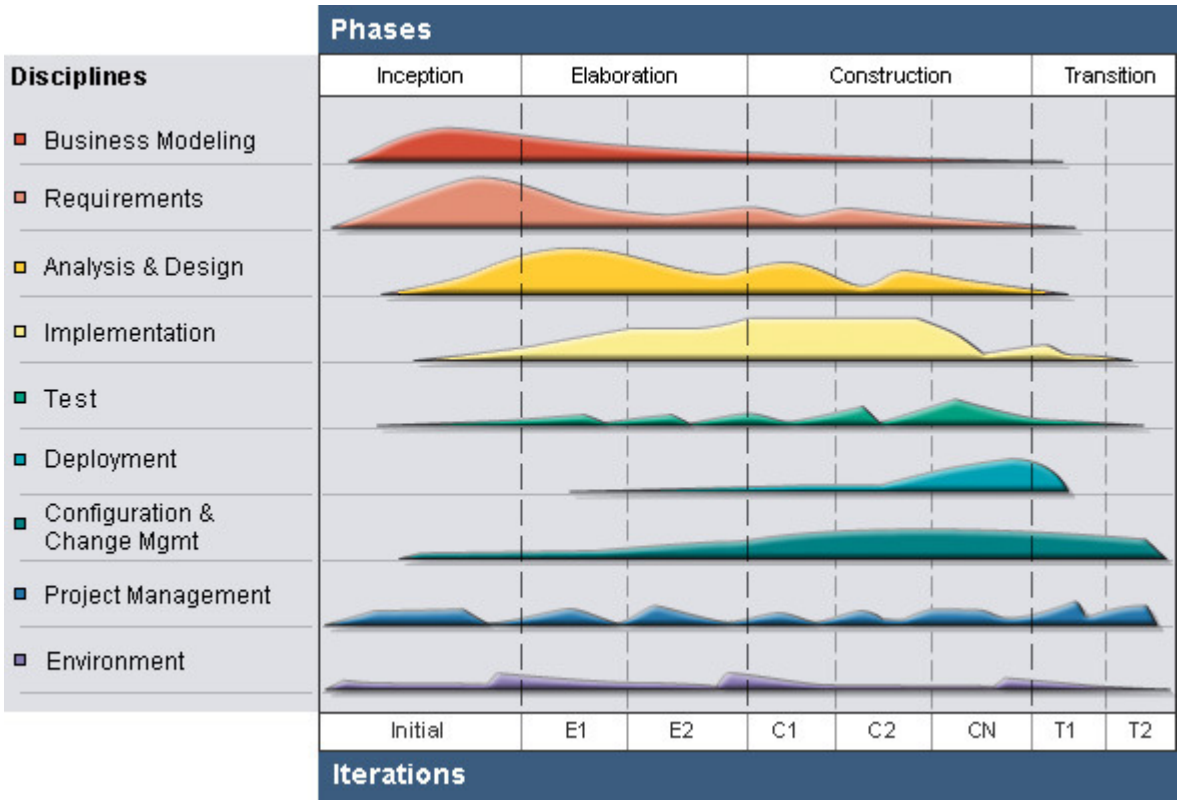
U menších či středních projektů mohou být modely aplikovány globálně na celý projekt, jinak i na vývoj jednotlivých částí software nebo dokonce při psaní jednotlivých funkcí a tříd.

Zcela evidentně se nabízí otázka, proč se i k takovýmto neefektivním modelům jako je „Velký třesk“ nebo „Programu a opravuj“ vůbec přistupuje? Odpovědí může být celá řada. Může to být podceněním složitosti celého procesu vývoje, ale také například může nastat situace, kdy na poslední chvíli vyjde najevo chyba ve specifikaci produktu a vývojář musí zaimprovizovat, navíc třeba ve stresu, pod časovým nátlakem. V takových situacích programátor občas uslyší od svého nadřízeného větu: „Nějak to udělej ať to funguje, zítra to musíme nainstalovat u klienta“.

Se vším tím musí tester počítat a použití těchto modelů jako přístupu k vývoji by jistě mělo být zohledněno při tvorbě plánu testování a pří návrhu jednotlivých testových případů.

2.3 Role testování v rámci metodiky RUP [RUP]

Jak již jsem výše zmínil, testování by mělo probíhat ve všech fázích vývoje softwaru. Na následujícím obrázku lze vidět, jak je zastoupeno testování v jednotlivých fázích metodiky Rational Unified Process.



Obrázek 1: činnosti v rámci metodiky Rational Unified Process

Jak jsem již zmínil, metodika RUP je postavena na iterativním modelu a v rámci každé iterace má testování svoje místo. Jsou zde přesně definovány role v rámci testování a dokonale rozdělené činnosti a odpovědnosti mezi tyto role. Testeři v rámci metodiky RUP pracují s množstvím formálních dokumentů a tím je celý proces dokonale zdokumentován. RUP je vhodné použít zejména na velké projekty. Při menších projektech by se mohl takovýto formalizovaný a podrobný přístup zbytečný a mohl by i brzdit samotný projekt.

3 Úvod do testování softwaru

Technik testování softwaru je celá řada a pro jejich lepší pochopení se neobejdeme bez jejich rozčlenění podle několika hledisek. Právě členění technik testování bude věnována tato kapitola. Kategorie testovacích metod se různě prolínají, proto samotný konkrétní test může současně spadat do více kategorií.

Již víme, že testování software není činností jednorázovou, která proběhne na konci projektu. Různé formy testů by měly být postupně aplikovány ve všech fázích projektu. Hlavním argumentem pro toto tvrzení jsou náklady spojené s chybami. Platí, že čím dříve je chyba odhalena, tím nižší jsou náklady na její odstranění. Za další argument lze považovat fakt, že výskyt příliš mnoha chyb současně znatelně znesnadňuje (někdy i zcela znemožňuje) i samotné testování, což může značně zdržet celý projekt. Proto je potřeba chyby odhalovat a opravovat průběžně.

Následující tři kapitoly popisují různá hlediska klasifikace technik testování. Jednotlivá členění jsou čerpána z několika zdrojů, které se v tomto ohledu doplňují: [Patton], [RUP], [Vrazelova], [Kunsky].

3.1 Techniky testování podle způsobu provedení testů

3.1.1 Testování černé skříňky a bílé skříňky

Testování černé skříňky (také provozní testování, funkcionální testování nebo Black-box test) je postaveno na myšlence, že tester nezná (nebo alespoň nevyužívá) vnitřní logiku testovaného software. Testování spočívá ve sledování vstupů a výstupů. Kontroluje se pouze, jestli po zadání konkrétních vstupů program vrátí správné a přesné výstupy. Ovšem co se děje s daty „uvnitř“ programu zde není předmětem zkoumání.

Oproti tomu testování bílé skříňky (také strukturální nebo White-box test) se opírá o znalost vnitřní logiky systému a zdrojových kódů. Netestuje se tedy jen, zda program podává správné a přesné výstupy vzhledem k daným vstupům, ale zejména, jak výstupy získá, zda se nějaké operace zbytečně neprovádí vícekrát, zda v kódu nejsou skryté chyby, které se provozním testováním jen těžko odhalí atd. Velkou výhodou strukturálního testování je to, že při znalosti vnitřní logiky systému a zdrojových kódů se dají lépe vytipovat a naplánovat testové případy (viz dále). Znalost vnitřní logiky systému však může mít pro testera i negativní vliv. To proto, že samotná znalost logiky systému jej již jaksi navádí k určitému chování při testování. Toto chování však může být poněkud jiné, než chování koncového uživatele. Z toho plyne, že testování by se měli zúčastnit jak testeři, kteří znají vnitřní logiku systému, tak i testeři, kteří tuto znalost nemají.

3.1.2 Manuální a automatizované testování

Další možnost rozdělení technik testování je manuální testování a automatizované testování. Rozdíl mezi nimi spočívá v tom, že při manuálním testování testuje člověk – tester, zatímco pří automatizovaném testování za něho testuje nějaký software. Pochopitelně obě tyto metody mají svojí povahou předurčeno použití. Někdy tam, kde je velmi dobře použitelné manuální testování je velmi těžkopádné, ne-li nemožné použít automatizované testování a naopak. Zatím co automatizované testování použijeme zejména na testování jednoduchých nebo dobře algoritmizovatelných operací ve velkém objemu, při testování složité logiky systému, kde je potřeba lidského mozku, se dá automatizované testování použít jen velmi obtížně, ne-li vůbec.

Jako typické příklady, kdy je vhodné použít automatizované testování, lze považovat například testování velkého množství najednou provedených požadavků na systém, větší množství současně přihlášených uživatelů, profilování výkonu (tedy zjišťování které operace zabírají jakou část výkonu), počítání dotazů na databázi apod.

3.1.3 Statické a dynamické testování

Dynamické testování spočívá v testování programu za běhu, ať už se jedná o testování provozní či strukturální, manuální i automatizované. Oproti tomu statické testování spočívá v testování programu, aniž by byl spuštěn – tedy zkoumání programového kódu, jeho syntaxe, přezkoumávání algoritmů, revize specifikace aj. Příkladem statického testování je i samotná kompilace programu, kdy překladač kontroluje syntaxi.

3.1.4 Testování splněním a selháním[3]

Při začátku testování programu nebo jeho části je vhodné nejprve otestovat základní funkčnost, provést ty nejpřirozenější testové případy – tedy zjistit, zda program nebo jeho testovaná část vykazuje základní známky funkčnosti. Tomuto způsobu testování se říká testování splněním (test-to-pass). Účelem zde není pokoušet se systém nějak shodit. Jedná se spíše o simulaci běžného použití systému, a nezachází se do extrémních či okrajových situací.

Test selháním (test-to-fail) provádíme až tehdy, když je zabezpečena základní funkčnost systému – tedy test splněním proběhl úspěšně. V této fázi již vyvíjíme na systém extrémní zátěž, testujeme neobvyklé testové případy, stabilitu systému a podobně. Jako bychom se pokoušeli systém nějak shodit. Zaměříme se zejména na slabá místa v softwaru a volíme takové testové případy a v takovém pořadí, abychom si „vynutili“ projevení maximálního množství chyb, které se v systému nachází.

[3] Některé zdroje uvádí překlad testování splnění a testování selhání

3.2 Techniky testování podle úrovně vývoje

Další možností, jak rozčlenit testování software je podle úrovně vývoje. Nemáme zde na mysli fáze vývoje jako analýza, návrh, implementace … ale na jaké úrovni vývoje se nachází předmět našeho testování.

3.2.1 Testování v průběhu vývoje (Developer Testing)

Testování v průběhu vývoje obvykle provádí sám programátor při psaní programového kódu a to formou jednotkových testů, integračních testů, někdy i systémových testů. Testování v průběhu vývoje není prací testerů.

3.2.2 Jednotkové testy (Unit Testing)

Jednotkové testy jsou zaměřeny na testování nejmenších testovatelných částí systému. Sledují se chování a datové toky – zda odpovídají definicím. Obvykle se provádějí hned po implementaci nově dokončeného modulu. Jednotkové testy zpravidla ještě provádí programátor.

3.2.3 Integrační testy (Integration Testing)

Zatím co jednotkové testy zjišťují funkčnost jednotlivých částí systému, integrační testy ověřují, zda tyto části spolu dohromady správně spolupracují, zda spolu správně komunikují apod. Zaměření je zde na nedokončené, opomenuté části a na rozhraní jednotlivých komponent.

Integrační test lze považovat za velmi významný milník v projektu vývoje systému, neboť zde se již přikračuje k testování případů užití (use case).

Integrační testování by mělo obsahovat testování jak testerů, kteří jsou součástí vývojového týmu – ti by prováděli white-box testy, tak i nezávislých testerů, kteří nejsou součástí vývojového týmu – ti by prováděli black-box testy (Independent Testing – viz dále v této kapitole). Každý z těchto dvou týmů by měl vymezeno jakési „pole působnosti“ tak, aby se navzájem příliš nepřekrývala [RUP].

3.2.4 Systémové testy (System Testing)

Zde se systém testuje jako jeden celek. Systémové testy mohou začít jakmile jsou implementovány nejdůležitější části systému (tedy alespoň některé z případů užití).

3.2.5 Akceptační testy (Acceptance Testing)

Akceptační testy představují finální testy před distribucí systému zákazníkům. Zjišťuje se, zda je software připraven k používání. Důraz je zde kladen na dříve zjištěné problémové oblasti systému, tj. oblasti, které byly v uplynulých testech zdrojem většího množství chyb.

Existují tři formy akceptačních testů:

- **Formální akceptační test** – je dopředu přesně naplánován, testové případy jsou vybrány ze systémových testů, test je přesně zdokumentován. Provádí interní testeři.

- **Neformální akceptační test** – není přesně naplánován. Každý tester si zvolí sám co a jak bude testovat. Tento test je taktéž dobře zdokumentován. Provádí interní testeři.
- **Beta test** – od předchozích testů se liší v tom, že jej neprovádí interní testeři či vývojáři. Tento test provádí nezávislé osoby například z řad zákazníků. Jedná se tedy o test na konečných uživatelích. Test není nijak naplánován ani zdokumentován, jedná se však o velmi efektivní způsob testování, neboť se softwarem je zde zacházeno tak, jak s ním budou zacházet koneční uživatele – je to tedy jakási simulace ostrého provozu. Výsledkem beta-testů bývá odhalení mnoha skrytých chyb.

Rational Unified Process v rámci svých úrovní testování (Levels of Test) definuje ještě další dvě úrovně testování.

3.2.6 Nezávislé testování (Independent Testing)

Jak již bylo zmíněno, nezávislé testování provádí nezávislí testeři, kteří nejsou součástí vývojového týmu a nemají interní znalost systému. Tím pádem jejich testování má formu black-box testů a tím mohou vnést do testování poněkud více uživatelský přístup než do něho vnesou interní testovací týmy. Nezávislé testování by mělo obsahovat jak verifikaci, tak i validaci systému.

Záměrem nezávislého testování je poskytnout odlišný pohled na věc a tím i odlišné testy, navíc v bohatším (rozmanitějším) prostředí, než které má k dispozici člen vývojového týmu.

3.2.7 Independent StakeholderTtesting[4]

Independent Stakeholder Testing je alternativním pohledem na nezávislé testování a je zaměřeno na potřeby a zájmy všech, kterých se má projekt či výsledný produkt týkat. Týká se tedy zejména o validaci systému nejen z pohledu uživatele, ale i dalších zájmových skupin.

3.3 Testování podle dimenze kvality [RUP]

Jedním z modelů, který popisuje dimenze kvality je model FURPS+, který je jako východisko řízení kvality použit i v metodice Rational Unified Process. Tento model zahrnuje pět základních (funkcionálních) dimenzí kvality a dále libovolný počet ostatních (nefunkcionálních) dimenzí. Při vývoji a testování software by měl být kladen důraz na každou z těchto dimenzí a jejích aspektů.

Jedná se o následující dimenze a jejich aspekty:

*3.3.1 Funkčnost (**F**unctionality)*

Při testování funkčnosti je potřeba se zaměřit na tři oblasti, kterými jsou:

[4] Vzhledem k obtížnosti přeložení slova „Stakeholder“ do českého jazyka ponechávám pouze původní anglický termín

- **Funkce** - test je zaměřen na ověření funkcí systému jako celku i jeho částí a to za „normálních“ podmínek. Předmětem testování je ověřit, zda systém pracuje správně.
- **Bezpečnost** – test spočívá v ověřování bezpečnosti dat a ochrany proti neoprávněnému použití systému.
- **Kapacita** – test zaměřen na schopnost systému obsluhovat velké množství vstupů a poskytovat velké množství výstupů, obsluhovat velkou databázi apod.

3.3.2 *Použitelnost (**U**sability)*

- **Uživatelské rozhraní** - kontroluje se přívětivost a konzistence uživatelského rozhraní, uživatelská dokumentace, nápověda, průvodci a další aspekty, se kterými přichází do přímého kontaktu koncový uživatel systému
- **Uživatelská dokumentace a školící materiály** – předmětem testování uživatelské dokumentace a školících materiálů je zkoumání, zda obsahují vše potřebné, zda neobsahují něco, co by obsahovat neměly, dále je potřeba ověřit, že dokumentace je konsistentní se samotným produktem, že sedí názvosloví i samotné popisované skutečnosti. Jako velmi efektivní způsob testování uživatelské dokumentace je nejen testování členy vývojového týmu, ale i testování na samotných uživatelích.

3.3.3 *Spolehlivost (**R**eliability)*

- **Náchylnost k chybám** – test ověřuje odolnost systému proti selhání a to na jednotlivých částech systému. Při testování náchylnosti k chybám je na místě použití zátěžových testů (Stress test), které ověří chování systému v extrémních podmínkách jako například velké množství najednou přihlášených uživatelů, nedostatek pamětí, špatná dostupnost datových zdrojů apod.
- **Zotavitelnost systému** – u každého systému, byť by byl sebelepší, je potřeba počítat se situací, že dojde k jeho „spadnutí“. V takovém případě je potřeba jej obnovit (zotavit). Právě obnovení činnosti systému by také mělo být předmětem testování. Pravděpodobně nejdůležitějším aspektem obnovení činnosti systému je zálohování a obnovování dat a proto by se této části měla věnovat dostatečná pozornost.
- **Předvídatelnost** – dalším z mezníků spolehlivosti systému je předvídatelnost jeho chování a to právě také v nepříznivých výjimečných situacích. Může být užitečné, v některých případech dokonce nezbytné některé simulovatelné nepředvídatelné situace simulovat a ověřit při nich chování systému.
- **Přesnost** – předmětem testování je ověřit, zda systém poskytuje přesné výstupy vzhledem k prováděným operacím.

- **Průměrný čas mezi selháními (Mean Time Between Failure)** – jedná se o statistické vyhodnocení jak často v průměru dochází k nefunkčnosti systému.

3.3.4 Výkon (Performance)

- **Zátěžový test** – sledují se doby reakce systému různých konfiguracích a různých úrovní zátěže. V případě stanovených výkonnostních limitů se sleduje jejich dodržení.
- **Rychlost** – Rychlost systému je jedním z nejdůležitějších aspektů výkonu. Rychlost se měří nejčastěji pomocí srovnávacích testů tzv Benchmarků. Jedná se o testy, které srovnávají softwarový nebo hardwarový systém s jiným systémem pracujícím v podobných podmínkách.
- **Dostupnost** – pravděpodobně neexistuje systém, který by byl v provozu a fungoval nepřetržitě po celou dobu jeho životnosti. U každého systému je potřeba počítat s výpadky jeho funkčnosti, ať už způsobené závadami nebo jen běžnou údržbou, zálohováním atp. Dostupnost systému se udává v procentech a znamená, jakou část roku je systém funkční. Samozřejmě na každý systém jsou kladeny jiné nároky co se týká dostupnosti. Tyto nároky musí být předem specifikovány.
- **Doba odezvy** – S rychlostí systému značně souvisí doba odezvy. Při optimalizaci doby odezvy systému je potřeba vycházet z výkonnostního profilu systému. Zjednodušeně můžeme výkonnostní profil charakterizovat jako časy potřebné k provedení různých fází akce, jejíž dobu odezvy sledujeme. Následná optimalizace by pak měla začít u těch nejpomalejších fází neboť tam se dá očekávat největší zrychlení při optimalizaci. Profilování výkonu je typickým příkladem použití automatizovaného testování.
- **Doba zotavení** – v této kapitole již bylo zmíněno zotavení systému. Vedle způsobu provedení zotavení, snadnosti zotavení a dalších aspektů je neméně důležité hledisko času – tedy jaká doba je potřeba k zotavení systému, obnovení dat apod.
- **Používání zdrojů** – Výkonnost systému značně ovlivňuje sdílení zdrojů, zejména datových. Sdílení zdrojů může mnohdy znamenat velmi značné zpomalení systému, proto je potřeba jeho optimalizaci věnovat dostatečnou pozornost. Současné přístupy ke stejným datům je vhodné simulovat automatizovaně, kdy testovací nástroj v jeden okamžik simuluje více uživatelů přistupujících ke stejným datům nebo jiným zdrojům.

3.3.5 Podporovatelnost (Supportability)

- **Kompatibilita** – test kompatibility zjišťuje použitelnost jednak s různým software, s různým hardware a dokonce i použitelnost software samého se sebou – tedy například sdílení vstupů a výstupů mezi různými verzemi téhož software

- Testovatelnost
- **Rozšiřitelnost** – architektura systému do jisté míry předurčuje, jak snadné bude rozšiřování systému v budoucnosti. Pokud se již v úvodních fázích projektu s budoucím rozšiřováním počítá, je toto rozšiřování potom jednodušší a výsledek kvalitnější.
- **Přizpůsobivost (flexibilita)** – neboli pružnost systému – testování ověřuje zejména chování systému po změně prostředí, například přechod na jiný typ databáze, jiný operační systém a další.
- Spravovatelnost
- **Konfigurovatelnost** – ověřuje se funkčnost na různém hardware a při různých softwarových konfiguracích
- **Instalovatelnost** – ověřuje chování instalace za různých podmínek – různá konfigurace, hardware, nepříznivé podmínky jako např. nedostatek místa na disku apod.
- **Lokalizovatelnost** – většina systémů je vyvíjena za účelem realizace na více než jednom trhu, v různých geografických a hlavně jazykových prostředích. Přizpůsobivost systému těmto různým prostředím by měla být brána v úvahu již při samotném návrhu systému. Z důvodu pozdější možné lokalizace by systém neměl mít texty, které zobrazuje uživateli přímo v kódů, ale umístěné mimo kód v různých jazycích a v kódu jen odkazy. Při testování lokalizace je zejména potřeba ověřit překlady všech zobrazovaných výstupů z hlediska věcné i jazykové správnosti a také zkontrolovat správné zobrazování. Při zobrazování si zejména všímáme zda vyhovuje znaková sada, zda jsou lokalizovány i obrázky a schémata.

3.3.6 Ostatní (nefunkcionální) dimenze

Kromě základních pěti dimenzí kvality, které jsem právě popsal je potřeba brát v úvahu takové dimenze, které jaksi nejsou na výsledném produktu přímo „vidět“. Ty jsou v modelu FURPS+ reprezentovány právě znakem „+“. Jedná se o takové dimenze, které jsou důležité zejména při vývoji produktu nebo jeho dalších úpravách. Zahrnují například standardy při psaní kódu, správně zvolenou architekturu databáze, nebo libovolný počet dalších věcí, které jsou pro projekt důležité.

4 Role v rámci testování

V rámci testování softwaru existuje řada činností, od plánování testů, návrh testů až po samotné vykonání testů a podání zprávy o nalezených chybách. Všechny tyto činnosti, musí být v testování obsaženy a z nich vyplývají i role v rámci testování. Je zřejmé, že každý konkrétní postup či metodika si svým charakterem může vyžádat poněkud jiný přístup a členění, ale pro účely této práce budu v tomto ohledu opět vycházet z metodiky Rational Unified Process, která definuje následující čtyři role v rámci testování softwaru [RUP].

4.1 Manažer testů (Test Manager)

Manažer testů je role, která máá na starosti tvorbu a obhajobu standardů kvality, metriky posuzování kvality, řízení nákladů a je celkově odpovědna za testování v rámci konkrétního projektu.

4.1.1 Požadované schopnosti

- obecné znalosti v oblasti softwarového inženýrství,
- zkušenosti v oblasti testování softwaru a znalost technik testování a testovacích nástrojů
- schopnost komunikace s lidmi a přesvědčovací schopnosti
- znalost testovaného systému
- případně zkušenosti s programováním nebo vedením týmu programátorů

4.1.2 Odpovědnosti

- vyjednávání o činnostech a výstupech testování
- zajištění plánování testů a řízení zdrojů potřebných k testování
- hodnocení efektivity a chodu testování
- obhajoba úrovně kvality
- obhajoba úrovně zaměření při vývoji na testovatelnost

4.1.3 Zaměření při vývoji na testovatelnost

Stejně, jak je tomu v mnoha problémech, pokud se s něčím počítá předem a tým se na to řádně připraví, je to potom v budoucnu mnohem jednodušší. Samozřejmě opět mám na mysli testování. Způsob, jakým je programový kód napsán může značně ovlivnit, jak obtížné bude jej efektivně otestovat, ať už na úrovni jednotkových či integračních testů, nebo na vyšších úrovních testování. Kód například může obsahovat navíc několik řádků, které budou zapisovat právě probíhající činnost, nebo

výsledky nějakých akcí do logu. Při identifikaci a lokalizaci chyb potom tyto údaje mohou být k nezaplacení.

Zapisování do logu je samozřejmě jen jeden z mnoha a mnoha možností, jak testovatelnost softwaru zlepšit. Slouží zde pouze jako ilustrativní příklad pro lepší pochopení termínu testovatelnost.

4.2 Analytik testů (Test Analyst)

Analytik testů identifikuje a definuje, jaké testy je potřeba provádět. Také pravidelně sleduje celkový pokrok v rámci testování, jeho výstupy a celkovou kvalitu softwaru. Analytik testů také reprezentuje osoby, které jsou do projektu nějak zainteresované (stakeholders), ale nejsou přímo členy vývojového týmu.

4.2.1 Požadované schopnosti

- analytické myšlení,
- průbojnost,
- smysl pro detail,
- porozumění běžným selháním softwaru,
- znalost testovaného systému,
- zkušenosti s testováním.

4.2.2 Odpovědnosti

- identifikace oblastí systému, které je potřeba testovat,
- definování vhodných testů a testovacích dat,
- zhodnocení výstupu každého testového cyklu.

4.3 Návrhář testů (Test Designer)

Návrhář testů tvoří konkrétní přístupy k testování a zajišťuje jejich řádnou implementaci. Jeho činnost spočívá ve volbě vhodných technik testování a nástrojů, vše v souladu se zdroji, které má testovací tým k dispozici.

4.3.1 Požadované schopnosti

- zkušenosti s testováním,
- schopnost rozpoznání a řešení problémů,
- hluboké znalosti v oblasti hardware i software,

- zkušenosti s nástroji automatického testování,
- programování,
- schopnost vedení týmu programátorů,
- velmi detailní znalost testovaného software.

4.3.2 Odpovědnosti

- identifikace a popis konkrétních vhodných postupů testování,
- identifikace nástrojů podpory testování,
- definice a udržování architektury automatických testů,
- specifikace a kontrola konfigurace prostředí.

4.4 Tester (Tester)

Tester provádí navržené testy a dokumentuje výsledky testů.

4.4.1 Požadované schopnosti

- znalost navržených postupů testování,
- schopnost rozpoznání a řešení problémů,
- znalost testovaného softwaru,
- schopnost práce v používaných nástrojích pro automatické testování,
- zkušenosti s nástroji automatického testování,
- základy programování,
- debugování a schopnost lokalizace chyb.

4.4.2 Odpovědnosti

- implementace jednotlivých testů,
- sestavování a spouštění testů,
- dokumentace výsledků testů a kontrola řádného proběhnutí testů,
- analýza a chybových stavů a návrat z nich.

4.5 Poznámka k rolím

Na závěr popisu rolí v rámci testování bych rád upozornil na to, že se jedná opravdu o role v rámci testování, nikoli o konkrétné osoby. Jedna osoba může vykonávat i více těchto rolí v rámci jednoho projektu a zároveň více osob může vykonávat stejnou roli.

Není tedy vyloučeno, že například vedoucí projektu zastává roli manažera testů a analytika testů a dalších pět lidí zastává současně roli návrháře testů a testera. Role tedy nekopírují organizační strukturu, jsou pouze jakýmsi seskupením podobných činností prováděných v rámci testování.

5 Specifikace softwaru

Specifikace je základním pokladem pro testování software, proto je potřeba věnovat jí řádnou pozornost. Specifikace může být buď ve formě slovního popisu nebo popisu pomocí obrázků, tabulek, diagramů, může obsahovat use-case diagramy apod. a nebo jako neformální, většinou fyzicky (ani elektronicky) nevyhotovená – tedy pouze ve formě znalostí a představ členů podílejících se na projektu. Tato neformální forma však s sebou přináší řadu problémů, proto rázně doporučuji držet se řádně vyhotovené formální specifikace.

Tvorba specifikace softwaru není předmětem činnosti testovacího týmu, proto psaní specifikace nebudu v této práci věnovat pozornost.

Je známo, a já osobně to mohu z vlastní zkušenosti potvrdit, že nedokonalá specifikace je nejčastějším zdrojem chyb softwaru. Právě z tohoto důvodu je nanejvýše vhodné podrobit i specifikaci testování a řádné revizi. Pokud se totiž najde chyba ve specifikaci před tím, než je implementována, náklady na odstranění této chyby jsou minimální, stejně tak i časové ztráty.

5.1 Revize a zkoumání specifikace

Každý software se tvoří proto, aby jej používali nějací uživatelé. Jedna z prvních testovacích činností v rámci projektu vývoje software by tedy měla směřovat k ověření, zda produkt, který se plánuje vyvíjet, bude uspokojovat potřeby jeho budoucích uživatelů a dalších osob, které jsou do projektu zainteresované (tzv. stakeholders). U specifikace produktu je tedy potřebné provést její validaci (viz kap. 2.1.3). Tester by se tedy měl vcítit do role budoucího uživatele a zhodnotit specifikaci očima uživatele.

Dalším aspektem, který by měl být zhodnocen již na počátku zkoumání a revize specifikace v rámci její validace je zkontrolovat, zda jsou dodržovány všeobecné standardy a zásady, které by měl software splňovat. Příkladem mohou být standardy návrhu uživatelského rozhraní, hardwarové standardy, standardy komunikačních protokolů a další.

Teprve po provedení validace specifikace je na místě věnovat se velmi důkladně její obsahové stránce. Aby mohla být specifikace považována za dobrou, musí mít následující vlastnosti [Patton]:

- **Úplnost** – specifikace obsahuje vše, co by obsahovat měla. Žádná vlastnost či funkce nebyla opomenuta
- **Správnost** – řešení, které specifikace navrhuje, opravdu povede k vytčenému cíli
- **Přesnost a jednoznačnost** – formulace použité ve specifikaci nesmí být příliš vágní či nejednoznačné. Specifikace má jen jednu možnou interpretaci.

- **Konzistence** – popis jednotlivých funkcí nesmí být ve vzájemném rozporu. Ve specifikaci je použito jednotného názvosloví.
- **Relevance** – specifikace neobsahuje irelevantní informace, které nejsou důležité pro implementaci. Takové informace by mohli být matoucí. Ve specifikaci by se měla dát vyhledat původní potřeba zákazníka (uživatele).
- **Proveditelnost** – implementace specifikace je proveditelná v rámci omezení, která projekt má, v rámci jeho rozpočtu, časového plánu, personálního obsazení apod.
- **Bez kódu** – specifikace se věnují výhradně definici produktu, nikoli návodu na jeho implementaci. Neměla by obsahovat architekturu softwaru či dokonce programový kód.
- **Testovatelnost** – specifikace by měla obsahovat dostatek informací pro to, aby na jejím základě tester mohl naplánovat testování, navrhnout testové případy a testování provést.

5.1.1 Kde ve specifikaci hledat chyby?

Jelikož psaní specifikaci zdaleka není exaktní disciplínou a každá specifikace je svojí povahou a povahou výsledného produktu unikátní, není snadné (snad ani možné) popsat univerzální postup, jak spolehlivě zjistit, zda je specifikace dobrá či nikoli. Existuje však několik skupin slov, které když se ve specifikaci objeví, je dobré se nad touto oblastí specifikace zamyslet hlouběji a představit si je v širších souvislostech. Mnohdy totiž věstí nějakou chybu či nepřesnost [Patton].

- **Absolutní výrazy** – pokud se ve specifikaci vyskytnou slova jako například: vždy, každý, nikdy apod., je potřeba se zamyslet nad tím, jestli jsou opravdu právem tyto výrazy použity – jestli přece jen neexistuje případ, kdy tomu tak není.
- **Přesvědčující výrazy** – některá slova jako: evidentně, patrně, zřejmě … jako by vnucovala nějaký názor či myšlenku, ale je tato myšlenka skutečně korektní a pravdivá?
- **„A tak dále“, „Jako například“, „Třeba“** – takováto slova jasně signalizují neúplnost, nejasnost a nepřesnost specifikace[5]. Takto definované výčty či popisy jsou naprosto netestovatelné. Je potřeba přesně vyjmenovat, nebo jasně popsat, aby nebylo pochyb, co měl autor na mysli.
- **Nekvantifikovatelné výrazy** – příkladem by mohla být slova jako: malý, velký, rychlý, levný … Tyto výrazy jsou nejednoznačné a proto netestovatelné. Musí být specifikovány přesněji, například rychlý = doba odezvy maximálně 2 sekundy.

[5] Pokud neslouží jen k uvedení příkladu pro jasně a jednoznačně definovanou funkci

- **Nízká podrobnost** – výrazy jako například: Zpracováno, ošetřeno, nedovoleno … v sobě mohou skrývat řadu dalších funkcí, které je potřeba také specifikovat. Ve specifikaci by se neměla bez dalšího vysvětlení vyskytovat například věta: „Chybná vstupní data budou ošetřena“ Je nutné alespoň stanovit, jak se má systém zachovat když budou zadána chybná data a jak mají vypadat správně zadaná data.
- **Jestliže … pak** – pokud v textu specifikace nalezneme spojení „jestliže … pak“, měli bychom ověřit, jestli je také definováno „jinak“ – tedy co když neplatí podmínky v „jestliže“.

5.1.2 Testování úplnosti specifikace

Testování úplnosti specifikace je pravděpodobně nejobtížnější částí revize (testování) specifikace. Zřejmě neexistuje univerzální postup, jak úplnost dokonale ověřit. Opět ale existuje několik záležitostí, ze kterých se může vyjít.

- **FURPS+** - již byla řeč o dimenzích kvality. Každá dimenze kvality by měla být ve specifikaci řádně specifikována
- **Use-case diagram** – specifikaci softwaru často předchází use-case diagram, nebo podobný dokument. Je každý use-case detailně specifikován?

6 Plánování testů

Plánování testů bývá často považováno za první fázi testování (nepočítáme-li testování specifikace) a má několik aspektů, kterým by měla být věnována pozornost. Hlavním podkladem pro plánování testů je specifikace, která by již měla být řádně otestována.

Smyslem plánu testů je podle standardu ANSI/IEEE Standard 829/1983 for Software Test Documentation (překlad doslovně převzat z [SSTD]):

> *„Předepsat rozsah, postup, prostředky a časový plán aktivit spojených s testováním. Identifikovat jednotlivé testované položky, testované funkce a úkoly prováděné při testování, konkrétní osoby odpovědné za každý z úkolů a rizika spojená s definovaným plánem“*

Výsledkem plánování testů je plán testů – dokument, který by měl být fyzicky nebo elektronicky vyhotoven a všichni členové vývojového týmu (tedy programátoři, testeři, management projektu, analytici atd.) by s ním měli být seznámeni. Plán testů by měl být celým týmem řádně projednán a odsouhlasen.

Některé metodiky definují jakousi univerzální šablonu plánu testování. Existují ale i názory, že plán testů by podle žádné šablony vyhotovován být neměl, neboť každý projekt je svým způsobem unikátní a v každém projektu je potřeba zaměřit se na jiné oblasti a aspekty. Dokument vytvořený podle nějaké šablony by tedy potom mohl opomíjet některé významné zvláštnosti konkrétního projektu, nebo naopak obsahovat takové oblasti, které nejsou pro ten konkrétní projekt natolik důležité.

6.1 Náležitosti plánu testů

Ať už je plán tvořen pomocí nějaké šablony či nikoli, měl by vždycky mít alespoň následující náležitosti [Patton].

6.1.1 Co se bude testovat a jakých výsledků se má dosáhnout

Na samém počátku plánu testování je potřeba definovat a přesně vymezit produkt, který se bude testovat a dále definovat cíle tohoto produktu v oblasti spolehlivosti a kvality.

Výsledkem je tedy jasná, jednoznačná a všem srozumitelná definice cílů produktu. Jednotlivé cíle musí být jasně vymezeny, aby bylo možné jednoznačně určit, zda jich bylo dosaženo či nikoli.

6.1.2 Lidé, místa a věci

Další fází plánování testů je určení osob, které budou na testování pracovat, stanovit režim a způsob práce a způsob komunikace mezi členy. Je vhodné takto popsat nejen členy testovacího týmu,

ale všechny členy projektu, kteří s danou testovanou oblastí mají něco společného, tedy i vývojáře, kteří produkt vyvíjeli a kteří budou opravovat případné chyby, analytiky, manažery atd.

Co se týká míst a věci, je potřeba sesumarizovat, jaké vstupní prostředky budou používány, kde se tyto prostředky nachází, jaký je způsob jejich použití a přístupu k nim.

6.1.3 Definice a názvosloví

Jedním z nejdůležitějších předpokladů pro úspěch projektu jsou jasné definice a názvosloví pro všechny testované oblasti. Všichni členové, kteří se projektu zúčastní, musí s těmito definicemi a názvoslovím být řádně obeznámeni. Nesmí vznikat situace, kdy si každý člen týmu pod stejným slovem představí něco jiného. Tato nedorozumění bývají zdrojem mnoha velmi nepříjemných chyb. Je tedy nanejvýše vhodné, aby plán testování obsahoval i definice a názvosloví, i když názvosloví by mělo být definováno samozřejmě ještě dříve než začne vznikat plán testování.

Příklad z praxe

Ve firmě, ve které pracuji, vyvíjíme systém pro řízení požadavků. Každý požadavek, který je do systému zadán prochází svým „životním cyklem“ – prochází různými stavy v závislosti na nastavení systému. Při vývoji logiky stavů požadavků došlo právě k nedorozumění v názvosloví. Každý stav měl svůj interní název, ale z důvodu nedokonalosti ve specifikaci si každý člen týmu pod různými názvy představoval jiné stavy. Důsledkem tohoto nedorozumění bylo to, že každý z programátorů vyvíjel podle jiné logiky a každý z testerů testoval podle jiné logiky. Když nakonec celé nedorozumění „vyplavalo na povrch“, bylo potřeba vše sjednotit a dát do pořádku. Vývoj byl zpožděn o několik týdnů a přitom by celou situaci býval vyřešil hodinový meeting, kde by se názvosloví jasně definovalo a každý by se s ním seznámil.

6.1.4 Povinnosti a odpovědnost jednotlivých skupin

Na projektu se zúčastňuje více skupin – testeři, programátoři, management, a další skupiny. Jednou z klíčových záležitostí při plánování čehokoli je stanovení odpovědností. Jinak tomu není ani při plánování testování software. V této fázi plánu se zatím nestanovují odpovědnosti jednotlivých členů týmu, ale odpovědnosti jednotlivých skupin, tedy co je v odpovědnosti týmu testerů, co programátorů atd. Zejména je potřeba se zaměřit na takové činnosti, které by svojí podstatou mohly být předmětem činnosti různých skupin.

Asi nejvhodnější formou zobrazení je tabulka, která informuje o činnostech, které budou probíhat v rámci projektu a o skupinách, které jsou za činnost odpovědná a skupinách, které se činnosti zúčastní. Tabulka může vypadat například takto:

Činnost	Vedoucí projektu	Programátoři	Testeři	Tvůrci dokumentace	Marketing	Podpora produktu
Návrh produktu a jeho funkcí	-				X	
Interní architektura produktu	-	X				
Návrh a kódování		X				
Obecné testování			X			
Revize tištěných materiálů	-	-	-	X	-	-
Vytvoření seznamu konfigurací					X	-

Tabulka 1: Odpovědnost a účast skupin na jednotlivých činnostech
Vysvětlivky k tabulce:
Tabulka představuje pouze ilustraci, nikoli kompletní výčet činností v rámci projektu
Znak „X“ symbolizuje, že daná skupina je odpovědná za danou činnost
Znak „-„ symbolizuje, že daná skupina se zúčastní dané činnosti, ale není za tuto činnost odpovědná

6.1.5 Co se bude testovat

V další fázi plánu testování je potřeba rozhodnout, co se bude testovat a co ne. Pokud se u některých komponent rozhodne, že nebudou předmětem testování, mělo by následovat zdůvodnění tohoto rozhodnutí. Důvodem by například mohlo být, že některé části produktu byly implementovány už jako hotové a otestované a jejich základní funkčnost již tedy není potřeba testovat.

6.1.6 Fáze testování

Stanovení fází testování značně vyplívá z toho, jaký model nebo metodika vývoje softwaru byla použita. Metodiky totiž většinou testování také obsahují a fáze testování jsou v nich tedy definovány. Každá fáze musí mít jasně stanovená objektivní kritéria, aby bylo možné prohlásit, že fáze byla ukončena a začala další fáze.

6.1.7 Strategie testování

Strategie testování definuje, jaký postup bude použit při testování jako celku a v jednotlivých fázích. Jsou zde stanoveny metody, které budou použity pro různé fáze testování. Je potřeba učinit například následující rozhodnutí:

- testovat vlastními silami, nebo využít služeb specializované testerské firmy?
- na které části aplikovat black-box testy a na které white-box testy?
- kdy použít manuální testování a kdy automatické?
- vytvořit vlastní software pro automatické testování nebo koupit komerční řešení – v tom případě které?

6.1.8 Požadavky na prostředí

Po stanovení strategie testování následuje výčet prostředků, které bude potřeba pro provedení testů dle zvolené strategie.

Je potřeba požadavky předem pečlivě specifikovat. Spadají sem požadavky na personální obsazení, software, hardware a ostatní vybavení, prostory, externí firmy a další.

Je potřeba také počítat s tím, že požadavky, které nevyly vzneseny v této fázi se obvykle později velmi těžko získávají. Management už totiž má vytvořeny a implementovány finanční plány, které nechtějí, nebo dokonce nemohou už měnit.

6.1.9 Pověření konkrétních osob

Dalším z klíčových částí plánu testování je další rozdělení činností a odpovědnosti, tentokrát však už ne jen na jednotlivé skupiny, ale na konkrétní členy týmu. Každý člen týmu musí vědět, za jaké oblasti je odpovědný a musí mít dostatek informací k tomu, aby mohl navrhovat testové případy.

Tato fáze by měla končit kontrolou, jestli neexistuje žádná oblast, která by měla být předmětem testování a za kterou není nikdo zodpovědný.

6.1.10 Časový plán testů

Další fází plánování je promítnutí všech činností souvisejících s testováním do časového plánu v rámci projektu.

Jak již bylo několikrát zmíněno, projekt vývoje software i samotné testování má několik fází. Každá z těchto fází je na testování jinak náročná a tedy má i jiná požadavky na zdroje, především lidské.

Počáteční fáze projektu sice také obsahují testování (například specifikací) a objem testovacích prací s časem a pokročilejšími fázemi projektu roste. Největšího objemu obvykle dosahují v konečné fázi projektu, kdy se provádějí akceptační testy. Tato struktura objemu testovacích prací by měla být hlavním východiskem při sestavování časového plánu.

Bohužel, plánování je věc jedna, ale realita je věc druhá. Často dochází k tomu, že se jedna fáze protáhne na delší časové období, než jaké je jí vyčleněno v časovém plánu. To samozřejmě naboural časový plán i těch fází, které na tuto fázi navazují. Pokud jsou jednotlivé fáze naplánovány pomocí absolutních časových údajů (například: začátek 13.8.2006 a konec 20.8.2006) a předchozí fáze se protáhne o 4 dny, potom na další fázi zbývají místo jednoho týdnu pouze 3 dny a dochází k stresovým situacím. Takovýto problém může výt vyřešen například tím, že se fáze časově nevymezí absolutním začátkem a koncem, ale relativním začátkem a trváním, například: začátek – dokončení alfa-verze produktu, trvání – 7 dní.

6.1.11 Zprávy o chybách

V plánu testů by měl být také stanoven způsob, jakým budou testeři podávat zprávy o chybách, jak tedy bude probíhat komunikace mezi testery a programátory, kteří budou chyby opravovat.

6.1.12 Rizika a možná omezení

Každý projekt či činnost s sebou přináší i řadu rizik. Testování software není výjimkou. Je velkou výhodou, když jsou rizika známa už při plánování činnosti, v tomto případě testování. Identifikovaná rizika je potřeba zakomponovat do časových plánů a obeznámit s nimi celý tým, včetně managementu.

7 Návrh testových případů

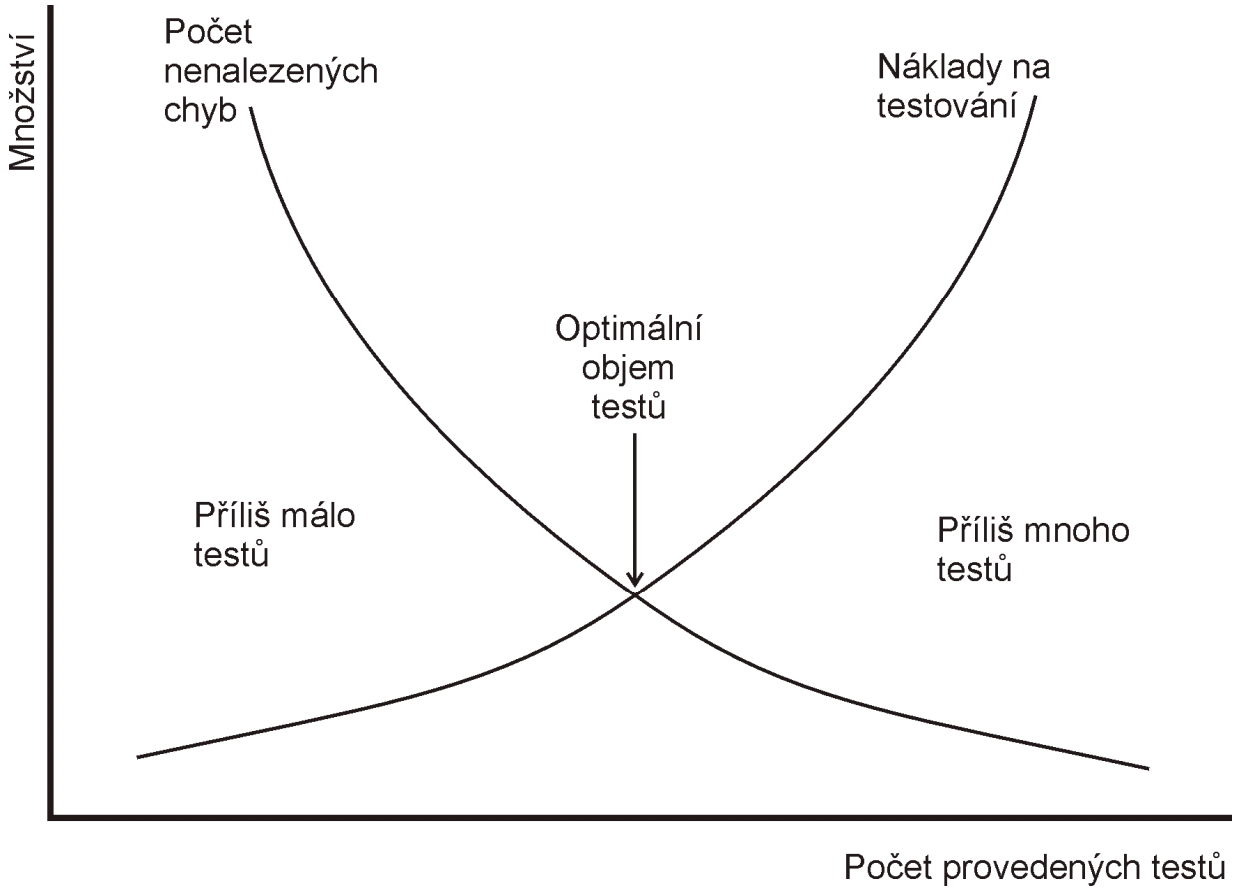
Drtivou většinu softwaru nelze kompletně otestovat. Existuje totiž tak obrovské množství různých vstupů a situací, že otestování úplně všech by bylo naprosto nereálné, ne-li nemožné. Na druhé straně ale, pokud neotestujeme software kompletně, nemůžeme odpovědně prohlásit, že neobsahuje chyby. Pracujeme tedy s určitým rizikem, že některé chyby, které v softwaru jsou, neodhalíme. Snížení tohoto rizika můžeme docílit právě správným návrhem testových případů, čemuž je věnována tato kapitola.

7.1 Optimální objem testů

Čím větší objem testů (tedy množství testových případů) provedeme, tím více snižujeme riziko, že v softwaru zůstanou chyby, které neodhalíme. Je ale zřejmé, že testování nemůžeme provádět do nekonečna. Těžko bychom asi před managementem obhajovali, že testování ne příliš rozsáhlého softwaru bude trvat dva roky. Při testování totiž není jediným milníkem dosažení kvality softwaru, ale také časový a rozpočtový plán projektu. Je tedy nutné zvolit optimální objem testování.

Při rozhodování o objemu testování by se mělo vycházet především z požadavků na výsledný produkt (například software pro řízení letového provozu nebo jaderné elektrárny jistě bude mít daleko přísnější požadavky na spolehlivost než například textový editor), a také rozpočtový a časový plán projektu.

Následující graf ukazuje příklad, jak je možné stanovit optimální objem testování:



Obrázek 2: Optimální objem testů

7.2 Třídy ekvivalentních případů

Nejzákladnější technikou, jak se oprostit od úplného otestování software (což je nereálné nebo dokonce nemožné) a zároveň co nejvíce snížit riziko, že jsou v softwaru chyby, které nebudou odhaleny, je vytvoření tříd ekvivalentních testových případů a z těchto tříd potom otestovat jen některé případy.

Rozdělení testových případů do ekvivalentních tříd (pokud je správně provedeno) je tedy postupem, pomocí kterého snižujeme velkou nebo nekonečnou množinu testových případů na mnohem menší podmnožinu, která je však i přes to stejně efektivní.

Správná identifikace tříd ekvivalentních případů je tedy nejdůležitějším milníkem testování software, od kterého se přímo odvíjí kvalita a efektivita testování.

Jako kritéria pro rozdělení testových případů do ekvivalentních tříd lze použít následující tři:

- Podobnost vstupů,
- Podobnost výstupů,
- Podobnost operací.

Jistě každého napadne, že termín „Podobnost“ není zcela vypovídající. Nelze totiž exaktně definovat, co tento termín znamená. Podobnost či rozdílnost je při testování identifikována na základě dalších obecných znalostí, na základě znalostí konkrétního softwaru, na základě zkušeností, nebo dokonce i na základě subjektivních představ a názorů. Každopádně základní prostředky pro interpretaci jsou znalosti a zkušenosti a čím více jich tester má, tím lépe dokáže podobnost interpretovat pro konkrétní oblast, kterou testuje.

Některé rady a návody, jak podobnost v konkrétních případech správně interpretovat poskytuje následující kapitola.

7.3 Jak tvořit testové případy

Velmi populárním pohledem na software je pohled, který definuje jeho dvě části. Jedná se o data a operace, které se nad těmito daty provádí. Tento pohled se dá použít i pro účely testování.

7.3.1 Testování dat

Pokud řekneme testování dat, měli bychom si uvědomit, že tím nemáme na mysli pouze čísla, znaky či nějaké soubory na disku. Za data je potřeba pouvažovat i například pohyb či kliknutí myší, text a grafika zobrazené na monitoru, nebo obrázek vytištěný na papíře. Mějme tedy na mysli data v tom nejširším slova smyslu.

Každý software pracuje s určitou množinou dat, přičemž tato data spadají do jistého oboru hodnot. Testování dat ve své podstatě obnáší zjištění, zda program opravdu správně pracuje se všemi daty, které spadají do oboru hodnot, se kterými by program měl pracovat.

Při testování dat je velmi důležité identifikovat obory hodnot a hraniční podmínky dat. Hodnoty, které se pohybují v těsné blízkosti těchto hranic totiž jsou to hlavní, co je potřeba otestovat. Pokud totiž program dokáže pracovat správě na hranici svých možností, téměř určitě bude pracovat správně i za normálních příznivých podmínek.

Hraniční podmínky

Jak již bylo řečeno, software na základě své specifikace pracuje s určitou omezenou množinou dat, která má nějaké hranice. Při testování dat se doporučuje otestovat hranice a jejich blízké okolí. Uveďme si testování hranice na příkladu:

Příklad testování názvu ukládaného souboru

Představme si, že software, který testujeme ukládá své výstupy do souboru. Předpokládejme, že název souboru musí obsahovat minimálně jeden znak a maximálně 255 znaků. Pro zjednodušení předpokládejme, že název může obsahovat jakékoli znaky Pro otestování kontroly dat pro název souboru tedy zvolíme následující testové případy:

Spodní hranici otestujeme tak, že se pokusíme zadat jméno souboru, které neobsahuje žádný znak (je tedy prázdný), jméno souboru, které obsahuje jeden znak a jméno souboru, které obsahuje dva znaky.

V následujícím kroku otestujeme horní hranici. Zadáme tedy jméno souboru obsahující 254, 255 a 256 znaků, případně i 257.

Zde vidíme, jak jsme efektivně snížili nekonečný počet testových případů na sedm. Lze totiž předpokládat, že pokud program správně uloží soubor, jehož název obsahuje 1, 2, 254, 255 znaků, že téměř jistě správně uloží i soubor, jehož název obsahuje například 12, 36, nebo 168 znaků. Pokud program po zadání 256 a 257 znakového názvu zobrazil správnou chybovou hlášku dle specifikace, dá se předpokládat, že jí téměř jistě zobrazí i při zadání 258, nebo třeba 1237 znakového názvu.

Již bylo řečeno, že testování softwaru je postaveno na riziku. Tedy pokud jsme neotestovali všechny testové případy (což jsme neotestovali), pořád tu existuje riziko, že ještě existují chyby, které jsme neodhalili.

V našem příkladě existují pouze dvě hranice. U reálného software je však potřeba se nad hranicemi zamyslet poněkud podrobněji, neboť hranic a omezení pravděpodobně bude více.

Subhraniční podmínky

Subhraniční podmínky (interní hraniční podmínky) se testují stejně jako Hraniční podmínky, poněkud se však liší svým charakterem. Nejsou totiž tak snadno viditelné a identifikovatelné. Jedná se o hranice a omezení interních součástí systému, například vlastností proměnných, které jsou v programovém kódu použity. Každá proměnná totiž má svůj obor hodnot, a tedy své vlastní hranice.

Tester však obvykle od programátora nedostane seznam kdy, kde a jak použil které proměnné. Proto je potřeba se jaksi dovtípit a hranice, které by mohly takto vzniknout, otestovat.

Jednou ze skupin subhraničních podmínek jsou mocniny dvou. Jelikož jsou data v počítači realizována binárně, všechny proměnné jsou realizovány mocninou dvou bitů. Například:

- 1 bit – možnosti 0 nebo 1
- 4 bity – možnosti 0 až 15
- 8 bitů (1 byte) – možnosti 0 až 255
- 16 bitů (1 slovo) – 0 až 65535 apod.

Na mocniny dvojky je tedy potřeba nahlížet jako na další hraniční podmínky.

Příklad

Představme si, že testujeme zadání hodnoty do formuláře, která se dle specifikace musí pohybovat v rozmezí 1 až 100000. Kromě otestování jednotky a 100000 jako hraničních podmínek by ještě bylo vhodné otestovat 14, 15 a 16, 254, 255 a 256 a také 65534, 65535 a 65536

Další skupinu subhraničních podmínek může tvořit kódování znaků. Každý znak má přidělen svůj kód a problém by mohl nastat, pokud jsou nějaká data omezena pouze na určité znaky. Při specifikaci množiny znaků, které mohou data obsahovat, vlastně tvoříme další intervaly (intervaly kódů znaků) a tyto intervaly mají také své hranice. I tyto hranice je potřeba otestovat.

Příklad

Vraťme se nyní k příkladu s názvem ukládaného souboru. Pro jednoduchost jsme předpokládali, že název může obsahovat jakékoli znaky. Nyní však předpokládejme, že název může obsahovat pouze velká a malá písmena. Použito je kódování ASCII.

Jelikož velká písmena mají v kódování ASCII kódy 65 až 90 a malá písmena 97 až 122, bude také potřeba ověřit, jak se program chová, použijeme-li znaky s kódy 64, 91, 96 a 123, tedy „@“, „[„ , „‘ “ a „{„.

Nesmyslné údaje, výchozí, prázdná, nulová hodnota

Další situací, kde by mohly vznikat chyby, je moment, kdy software vyžaduje zadání určité hodnoty, uživatel hodnotu však nezadá, nebo zadá nesmyslnou hodnotu. Specifikace by měla definovat, jak se má v tomto případě software zachovat. Pro výchozí, prázdné a nulové hodnoty tedy vytvoříme další třídu ekvivalence.

7.3.2 Testování operací

Jak bylo řečeno na začátku kapitoly 6.3, další složkou softwaru po datech jsou operace, které se nad těmito daty provádí. Jedná se tedy o samotnou funkcionalitu softwaru.

V kapitole o metodách testování softwaru podle způsobu provedení testů bylo použito jako jedno z hledisek pro kategorizaci testování splněním a testování selháním. Jen pro připomenutí, při testování splněním (test-to-pass) jde o to abychom otestovali logiku systému a jeho běžnou funkčnost. Při testování selháním (test-to-fail) se potom snažíme vyvíjet na software tlak a simulovat nepříznivé podmínky, abychom otestovali chování na hranici jeho možností. Poněkud podrobněji se touto problematikou bude zabývat tato kapitola.

Testování splněním

Pro testování splněním je nezbytně nutné, aby tester dokonale ovládal logiku systému, nebo alespoň funkcí, které testuje. Jako obvykle, základní dokument, na kterém je potřeba testování postavit je specifikace produktu.

Logika softwaru se dá charakterizovat pomocí stavů softwaru, chování v těchto stavech a přechody mezi stavy. Přechody mezi stavy jsou dle specifikace realizovány za určitých podmínek. V této fázi tedy budeme zjišťovat, zda program vykazuje správné chování ve svých stavech a zda správně fungují přechody mezi stavy.

Pro testování splněním se jeví jako velmi užitečné nakreslit diagram stavů a přechodů, ze kterého bude tester při testování vycházet. Diagramu musí obsahovat všechny stavy softwaru (nebo testovaného modulu atp.) spolu s podmínkami, za kterých se do tohoto stavu smí dostat. Co se týká chování softwaru v jednotlivých stavech, jistě by nebylo efektivní opisovat k nim celé kapitoly specifikace, je však vhodné, aby byly do diagramu zaznamenány alespoň podmínky, které se mají nastavit při přechodu do stavu a při jeho opuštění. Při tvorbě diagramu není podstatné, jakou technikou je tvořen, ani zda je tvořen ručně na papír nebo elektronicky pomocí specializovaného softwaru. Důležité je, aby byl kompletní a srozumitelný[6].

Nyní se dostáváme k zásadní a rozhodně k jedné z nejsložitějších částí testování. Je potřeba snížit počet testových případů na přijatelnou úroveň, tedy zvolit třídy ekvivalentních případů.

Opět bychom jen těžko hledali univerzální exaktní metody, jak třídy ekvivalence zvolit. Vycházejme tedy jen z několika obecných doporučení, jak při navrhování testových případů postupovat.

- **Navštívíme každý stav alespoň jednou** – při tom nezáleží, jakým způsobem se do stavu dostaneme, důležité však je, abychom prošli všechny stavy a zkontrolovali, zda jsou dodrženy všechny vstupní a výstupní podmínky (tedy zda jsou například označena správná políčka, jestli je okno správně veliké, jestli se správně změnil kursor apod.) Připomeňme, že tímto testujeme stavy, nikoli podmínky přechodů mezi nimi.
- **Otestujeme nejobvyklejší přechody a funkce** – nejobvyklejší přechody testujeme zejména proto, že předpokládáme jejich největší využití. Klademe na ně tedy nejvyšší nároky co se týká funkčnosti a kvality. Hlavní a stěžejní funkce přece musí fungovat především. Výběr nejobvyklejších přechodů a funkcí je záležitostí značně subjektivní, avšak pomoci nám mohou například specifikace, nebo výstupy z fáze analýzy (například diagramy užití apod.)

[6] U většiny softwarů bude diagram obsahovat velké množství stavů a s nimi souvisejících náležitostí, takže je zřejmé, že „začtení se“ do diagramu nebude věc snadná, musí být však srozumitelná jeho notace

- **Otestujeme nejméně obvyklé přechody a funkce** – důvodem pro zvolení nejméně obvyklých přechodů a funkcí je fakt, že často bývají vývojáři podceňovány a není jim věnována dostatečná pozornost například co se týká jednotkových testů. Proto se zde dá předpokládat zdroj chyb.
- **Otestujeme všechny chybové stavy a návraty z nich** – předmětem zkoumání zde bude, jestli jsou všechny chybové stavy správně ošetřeny. Program by v ideálním případě neměl za žádných okolností skončit výjimkou. Také kontrolujeme, zda chybové hlášky, které program zobrazuje, opravdu popisují pravou příčinu chyby. Pro uživatele je totiž velmi matoucí, když program zobrazí jinou chybovou hlášku, než která popisuje uživatelovu chybu. Uživatel může být od používání softwaru pak velmi snadno odrazen.

Testy selháním

Proběhly-li úspěšně testy splněním, základní funkčnost softwaru je tedy ověřena a nastal čas přikročit k další fázi – k testům selháním. V této fázi budeme na software vyvíjet tlak a simulovat nepříznivé podmínky.

Testování selháním je přímo ideální oblastí pro použití technik automatizovaného testování.

Jednou ze skupin nepříznivých podmínek je takzvané závodění. Jedná se o situaci, kdy více programů nebo více součástí stejného programu se snaží současně použít stejné zdroje. Může se jednat jak o datové zdroje jako databáze, současné ukládání stejného souboru apod., tak i o systémové zdroje jako je procesorový čas, stejná část paměti a tak.

Plánování testování závodění patří taktéž k poměrně obtížným disciplínám. Jednou z možností, jak testování naplánovat je procházet diagram stavů stav po stavu a u každého stavu se zamýšlet, jaké problémy by tu mohli vzniknout. Uvažovat nad tím, co by se mohlo stát, kdyby vyžádaný zdroj byl právě používán nebo například měněn. Rizikové může být také například sdílení tiskárny nebo databáze s jiným programem, nebo současné spouštění více instancí programu.

Další skupinou nepříznivých podmínek může být časté opakování stejné akce, například neustálé zapínání a vypínaná programu, spouštění stále stejné funkce apod. Zřejmě nejreálnější riziko, které zde hrozí je vznik takzvané díry v paměti. Díra v paměti vznikne tak, že některá funkce si alokuje určitý prostor v paměti, avšak při ukončení jej již neuvolní. Opakovaným spouštěním a vypínáním se tedy paměť postupně zaplňuje a volné paměti ubývá, až na konec může dojít k přetečení paměti.

Další skupinou testů, které testují nepříznivé podmínky jsou stresové testy (stress-test). Jedná se o testy, při kterých se ověřuje funkčnost systému s velmi omezenými hardwarovými prostředky, tedy například s malým množstvím paměti, místa na pevném disku, slabý výkon procesoru apod.

Zde je zejména za potřebí ověřit, zda systém opravdu správně pracuje i v minimálním hardwarovém prostředí a zda i v tomto prostředí dosahuje požadovaného výkonu a rychlosti. Pokud tomu tak není, je potřeba buď software v této oblasti zdokonalit, nebo přehodnotit reálnost požadavků na software.

Obdobný přístup jako u stresových testů aplikuje zátěžový test. Také se jedná o testování na hranici možností výkonu a stability, avšak zátěžový test nespočívá v minimalizaci systémových prostředků, ale v maximalizaci nátlaku vyvíjeného na software. Software je tedy nucen pracovat s velkými objemy dat a například s velkým množstvím současně přihlášených uživatelů. Zátěžové a stresové testy je vhodné i kombinovat, tedy vyvíjet velký nátlak na software běžící s minimálními systémovými prostředky.

Další techniky a rady

- **Předstírání nezkušeného uživatele** – Při testování software bývá někdy užitečné do finálních fází testování zapojit i takové uživatele, kteří počítačům příliš nerozumí. Může nás překvapit, jaká data se budou pokoušet do softwaru zadat o a jaký způsob práce se budou pokoušet a v neposlední řadě, kolik chyb možná objeví.
- **Hledáme chyby tam, kde jsme již nějaké našli** – bývá celkem obvyklé, že velká část chyb se nachází ve velmi malé části softwaru. Každý programátor je jen člověk a tak mohou existovat oblasti, které mu jdou programovat lépe a které hůře, nebo například může občas mít špatný den. V oblastech nebo třeba typově podobných funkcích, ve kterých jsme našli chyby tedy raději aplikujeme o několik testových případů více, můžeme totiž očekávat větší množství chyb.
- **Naslouchat předtuchám a intuici** – k testování software bychom se neměli stavět striktně a příliš exaktně. Mnohdy je velmi užitečné při testování naslouchat svým předtuchám a intuici.

8 Ohlašování chyb

Prací testera není jen plánovat testování a následně testy provádět, ale hlavním smyslem je oznamovat nalezené chyby. Může se zdát, že oznamování chyb je věci jasnou a jednoduchou, ale i oznamování chyb musí být efektivní a má svoje pravidla. Následující kapitola se tedy bude věnovat ohlašování nalezených chyb a komunikaci s programátory a ostatními členy týmu.

8.1 Náprava chyb

Jak jsme se dočetli v definici testování softwaru, práce testera není jen vyhledávání chyb, ale také zajištění jejich nápravy. Testovací tým je vlastně tím, kdo je odpovědný na kvalitu softwaru. Proto by měl dohlížet i na řádnou nápravu chyb.

Mnoho chyb, které tester najde a ohlásí zůstane neopraveno. Důvodů může být několik. Důvodem může být nedostatek času, nebo programátoři či vedení projektu nepovažují chybu za závažnou, a nebo by oprava chyby mohla prostě bít příliš obtížná nebo riskantní. Tyto příčiny sice tester asi moc ovlivnit nedokáže, avšak dalším důvodem, proč chyba občas zůstane neopravena je, že není oznámena efektivně a programátor buď dobře nepochopí, o jakou chybu se jedná, nebo se mu do opravy nechce a odsouvá ji a odsouvá. Zde může pomoci, když je chyba řádně a efektivně popsána.

Při ohlašování chyb se tedy držme následujících pravidel:

- **Chyby oznamujme co nejdříve** – čím dříve nalezené chyby oznamujeme, tím více času zbývá na jejich opravu. Je tedy velkou chybou aplikovat takový postup, kdy si tester „střádá“ chyby za celý týden testování a potom je najednou předá k nápravě.
- Chyby popisujeme efektivně – viz následující kapitola
- **Při oznamování chyb jsme nestranní** – při oznamování chyb dbáme na to, aby naše kritika byla konkrétní, objektivní a směřovala na program a jeho chyby, nikoli na programátora. Oznámení chyby by nemělo obsahovat věty jako: „Naprogramoval jsi to špatně“
- **Pokud předáme zprávu o chybách, tím to pro nás nekončí** – nejen že má tester vyhledávat chyby a podávat o nich zprávy, jeho povinností je i sledovat, zda jsou chyby řádně opravovány.

8.2 Jak oznamovat chyby

Efektivní oznámení chyby je tedy jedním z předpokladů, které napomáhají jejímu řádnému opravení. V této kapitole se pokusím sesumarizovat základní pravidla, která bychom měli dodržovat při oznamování chyb.

8.2.1 Relevance

Popis chyby by měl obsahovat jen relevantní informace bez zbytečných a s chybou nesouvisejících informací. Dbáme však na to, aby informace byli konkrétní a kompletní, avšak bez „informačního šumu“. Je vhodné například popsat posloupnost kroků a akcí, které vedly k chybě.

8.2.2 Jednotlivost

Chyby popisujeme jednotlivě. Pokud popíšeme v jedné zprávě více chyb, může při opravě dojít k opomenutí některé z nich a také z hlediska jejich dalšího sledování je vhodnější, když je na každou jednotlivou chybu vyhotovena zpráva.

Někdy může být celkem obtížné rozpoznat, zda se jedná o jednu chybu, nebo o více chyb. Z uživatelského pohledu se může jednat o dvě chyby a z hlediska programového kódu o jednu. V takovém případě raději nahlížíme na problém jako na dvě různé chyby, zajímá nás přece především jejich výskyt. Příčinu chyb již by měl vyhledat programátor.

8.2.3 Jasná identifikace a reprodukovatelnost

Když popisujeme výskyt chyby, měli bychom dbát na to, abychom jasně definovali, za jakých okolností k chybě dojde. Na základě tohoto popisu by měla jít chyba kdykoli nasimulovat.

Stává se, že některé chyby se zdají být velmi nahodilé a je obtížné přesně popsat, kdy se vyskytnou. Pro jejich nápravu je však jejich identifikace velmi užitečná. Chyby, o kterých programátor neví kdy se vyskytují se velmi těžko opravují a samotným programátorů se do jejich hledání většinou ani moc nechce, proto opravu chyby odloží na jindy.

Pokud hledáme takovouto nahodilou chybu, je potřeba si také uvědomit, že nemusí být způsobena chybou v testovaném softwaru, ale například špatnou funkcí hardwaru nebo třeba operačního systému. Zvážit bychom ale měli i otestování subhraničních podmínek (viz testování dat), zjistit, zda nedochází k dírám v paměti apod.

Velmi efektivní způsob hledání náhodných chyb je i úzká spolupráce s programátorem, který kód psal. Svůj kód totiž velmi dobře zná a možná ho během chvíle napadne, co by mohlo být příčinou.

Příklad nesprávného ohlášení chyby

„Včera večer, když jsem se vrátil z nákupu, zkusil jsem se zalogovat přes webový formulář jako uživatel s dlouhým uživatelským jménem (v polovině jsem udělal chybu, tak jsem se vrátil o tři znaky pomoci Backspace) a po kliknutí na Přihlásit program hodil výjimku. Když jsem se potom přihlásil jako jiný uživatel s kratším uživatelským jménem, přihlášení proběhlo dobře, ale úvodní stránka se špatně zobrazila."

Toto je popis chyby je na první pohled nesprávný. Pojďme si nyní poukázat na jeho nedostatky.

1) Měli bychom psát pouze relevantní informace, proto rozhodně větu o návratu z nákupu vypustíme. Raději specifikujme přesně verzi produktu, na které jsme testovali. Vzhledem k tomu, že webový formulář se odešle v aktuálním stavu, nemá žádný vliv, zda jsme uživatelské jméno napsali hned správně, nebo zda jsme mazali znaky pomocí backspace. Proto bychom neměli psát ani tuto zmínku.

2) Při popisu chyb jsme zásadně konkrétní. Proto co se týká „dlouhého" uživatelského jména, je potřeba informovat, jak přesně dlouhé to uživatelské jméno bylo a zda neobsahovalo nějaké speciální znaky, které by mohly chybu také způsobit. (Samozřejmě důkladně otestujeme, jestli chybu opravdu způsobila délka řetězce, nebo nějaké znaky). Podobně je potřeba lépe popsat, o jakou výjimku se jednalo a jaký přesně byl problém se zobrazením.

3) Pokud jsme zjistili, že původem výjimky byla skutečně délka řetězce, ještě také informujeme o tom, jakou délku řetězec musí mít, aby došlo k této výjimce.

4) Chyba s přihlášením a chyba se zobrazením úvodní stránky jsou dvě různé chyby a tak by na ně měly být vytvořeny dvě různé zprávy.

Popis těchto dvou chyb by tedy mohl vypadat například takto :

5) Ve verzi 1.7.3.110506 dochází při přihlášení přes webový formulář k následující výjimce: / … zde popsat výjimku …/. K chybě dochází, pokud je zadáno uživatelské jméno delší než 30 znaků.

6) Na úvodní stránce verze 1.7.3.110506 logo firmy překrývá nadpis „Vítejte v systému XYZ". Chyba se projevuje v prohlížeči Internet Explorer verze 5.5 a níže.

Podotýkám, že způsob evidence chyb se může projekt od projektu diametrálně lišit. Mohou být použity formuláře, systém pro evidenci chyb apod. Každopádně, tento příklad má sloužit pouze k demonstraci obsahové stránky ohlášení chyby, nikoli formální.

8.3 Priorita a závažnost chyb

Je zřejmé, že programátor nemůže opravit všechny vzniklé chyby najednou a obvykle se nestihne ani opravit všechny chyby před uvedením produktu na trh. Proto je potřebné chyby nějakým způsobem kategorizovat. Tedy dalšími vlastnostmi chyby, se kterými je potřeba naučit se pracovat je závažnost chyby a priorita jejího opravení.

- **Závažnost chyby** – jaké riziko či újma vznikne uživateli, když se chyba vyskytne.
- **Priorita** – jak je důležité chybu odstranit a kdy musí být odstraněna.

V každé firmě, podle toho, o jaký charakter softwaru se jedná, by měly být definovány stupně závažnosti a priority, a také pravidla, podle kterých se bude tvořit pořadí opravování chyb. Kategorie závažnosti a priority by mohly vypadat například takto [Patton]:

Závažnost

1. Havárie, ztráta či poškození dat
2. Funkční chyba, nesprávný výsledek
3. Kosmetické chyby jako např. jazykový překlep, špatné rozložení formuláře apod.
4. Připomínka

Priorita

5. Okamžitě opravit – chyba znemožňuje další testování nebo je velmi viditelná
6. Opravit před uvedením na trh
7. Opravit pokud na to zbude čas
8. Oprava může počkat na další verzi produktu

9 Postup testování

V předešlých kapitolách jsme se seznámili s teoretickým základem testování softwaru, rolemi, které jsou v rámci testování zastoupeny, revizí specifikace, plánováním testů a návrhem testových případů. Pro každý konkrétní projekt však musí být stanoven postup, v jakém pořadí provádět činnosti obsažené v těchto oblastech. Skutečně se tedy jedná jen o ilustrační příklad, který má sloužit jednak jako drobná rekapitulace a jednak pro utvoření lepší představy o logické návaznosti.

Poněkud složitější, ale zase o to efektivnější je v tomto ohledu spirálový (iterativní) model vývoje. V tomto modelu, jak již víme, neexistuje na počátku kompletní specifikace. Specifikace se postupně dotváří a upřesňuje v rámci „vrstev spirály“. I testování je zaměřeno téměř výhradně jen na tu oblast, která je předmětem této vrstvy a v každé této vrstvě může obsahovat různé typy testů, nebo dokonce relativně samostatný testový cyklus. Není proto tedy možné nadefinovat univerzální postup testovacích činností v rámci projektu. Tento postup musí být vždy sestaven na míru konkrétnímu projektu, nebo respektovat metodiku, podle které je software vyvíjen. Pokusím se ale alespoň popsat logickou návaznost činností v rámci testování software.

9.1 Testování (revize) specifikace

Specifikace softwaru vzniká někdy ve fázi analýzy[7] a je základním dokumentem, ze kterého se vychází při vývoji a samozřejmě i při testování. Není tedy možné, abychom plánovali testování softwaru, o kterém nevíme, jak bude vypadat a jaké na něj budou kladeny nároky. Jedná se tedy o testování, které není předmětem plánu testů, jak jsme si jej definovali v kapitole 6, je totiž podkladem pro jeho tvorbu.

Pokud bychom měli testování specifikace začlenit do kategorií testování software, jednalo by se o statické testování černé skříňky neboli statický black-box test. Testujeme totiž software, který není spuštěn (ani nemůže být, když ještě neexistuje) a neznáme jeho interní strukturu (tou se specifikace nezabývá). Dále se jedná o validaci, neboť zjišťujeme, zda specifikace odpovídá požadavkům uživatele, nikoli jestli daný program přesně odpovídá specifikaci.

Specifikace softwaru má ale dvě stránky. Zatímco při validaci specifikace zkoumáme zda software jí popisovaný bude odpovídat požadavkům uživatelů (měla by popisovat všechny případy užití softwaru a žádné by neměly být navíc), revizí zkoumáme i ostatní aspekty, tedy například zda je specifikace reálná, jednoznačná, vhodně formulovaná, úplná apod. Dá se tedy říci, že pojem revize je nadřazený pojmu validace specifikace.

[7] Mám zde na mysli analýzu jako obecnou činnost na počátku projektu, nikoli konkrétní fázi nějaké metodiky. Ne každá metodika totiž musí mít fázi nazvanou „Analýza“

Při revizi specifikace by nejprve měla být provedena její validace a teprve když se specifikace usoudí za validní, teprve potom se budeme zabývat ostatními aspekty. Důvodem pro tento postup je fakt, že po validaci specifikace může být rozhodnuto například o vypuštění některých případů užití a jejich specifikaci je zbytečné náročně revidovat.

9.2 Strategie testování

Strategie testování je první fází plánování testů. Je potřeba rozhodnout o základních otázkách týkajících se budoucího testování a ze strategie testování následně bude vycházet plán testů.

9.3 Plán testů

Další činností, která by měla následovat a vycházet ze strategie testování je vyhotovení plánu testů. O obsahu plánu testů pojednává kapitola 6.

9.4 Návrh a implementace testových případů

Až doposud byl postup činností celkem intuitivní. Nyní je však velkou otázkou, jaký zvolit postup samotného testování, tedy, v jakém pořadí provádět navržené testové případy. Nebudu se zabývat postupem testů členěných podle úrovně vývoje, neboť ty jsou samozřejmě vázány na postup samotného vývoje a není tedy potřeba se jejich časovému sledu zde věnovat. Zrovna tak není jednoznačně definovatelný postup z hlediska dimenze kvality. Můžeme však doporučit postup technik členěných podle způsobu provedení testů.

Jedna z otázek, která zde vyvstává je, zda nejdříve naplánovat testové případy a následně je jen provádět, nebo jestli navrhovat a provádět testové případy průběžně. Univerzální odpověď zřejmě opět neexistuje. Z vlastní zkušenosti se mi však zdá nejefektivnější oba tyto způsoby skombinovat. Je dobré si testové případy dopředu naplánovat, neboť potom můžeme lépe rozhodnout o jejich pořadí, ale na druhou stranu, při samotném provádění testových případů téměř vždy narazíme na oblast, která jimi není dostatečně pokryta a v takovém případě je vhodné navrhnout další testové případy a hned je implementovat.

Ať tak nebo tak, návaznost testových případů by měla respektovat alespoň následující postup, tedy nejprve provádět testování splněním a až poté testování selháním.

9.4.1 Testování splněním

První kroky při samotném testování softwaru by měly směřovat k ověření základní funkčnosti. Nejdůležitější v této chvíli je zabezpečit, aby systém fungoval v tom smyslu, aby dokázal obstarávat případy užití v normálním provozu.

Když máme simulovat běžné používání systému, samo se vybízí k použití technik manuálního testování. Testujeme především samotnou logiku systému a k tomu je za potřebí lidského mozku. Obsluhu případů užití samozřejmě můžeme testovat pouze pokud je program spuštěn, jedná se tedy o dynamické testování.

9.4.2 Testování selháním

K vyvíjení tlaku na software přistupujeme až když je zabezpečena základní funkčnost (nebo alespoň vhodná část základní funkčnosti). Tak, jak byl u testování splněním důležitý lidský mozek a znalost logiky systému, je u testů selháním důležitá rychlost, přesnost a sáhodlouhé opakování. K tomu je ideální použití automatizovaného testování.

9.5 Zprávy o provedení testů

Fází, která logicky následuje po provedení testových případů, je podání zprávy o provedení testů. Zprávy, které jsou výsledkem testování jsou dvojího druhu.

9.5.1 Zprávy o chybách

Jak již zde bylo několikrát zmíněno, nalezené chyby by se měly ohlašovat co nejdříve, v ideálním případě ihned po jejich nalezení. S ohlašováním chyb nečekáme až provedeme všechny naplánované testy. Při jejich nápravě občas hrají roli nejen dny, ale i hodiny či dokonce minuty.

9.5.2 Závěr z testování

Po provedení některých typů testů, zejména akceptačních, ale i jiných, se očekává učinění nějakého závěru. Učinění závěru není nic jiného, než rozhodnutí na základě předem stanovených metrik, zda software, nebo jeho testovaná část, splňuje požadavky, které jsou na něj kladeny. Toto stanovisko může znamenat základní podklad pro rozhodnutí o uvedení softwaru do užívání, nebo například pro rozhodnutí přejít do další fáze vývoje nebo testování.

9.6 Opakované provádění testů

Pokud tester nalezne chybu a podá o ní zprávu, neměl by považovat tuto chybu za vyřešenou a uzavřenou. Měl by stav jím nalezených chyb i nadále sledovat. Pokud byla provedena nějaká oprava některé chyby, znova implementovat testové případy, kterými chybu odhalil, nebo i jiné testové případy, a ověřit, zda je chyba opravdu řádně opravena. Není vyloučeno ani opakované upozorňování vývojářů na chyby, které stále čekají na opravu.

10 Závěr

Testování softwaru, ostatně stejně jako mnoho dalších oblastí, není věc, kterou se člověk naučí jen tím, že si o ní přečte. K testování softwaru musí mít člověk předpoklady a hlavě, musí ho to bavit. Jindy tak negativně vlastnost jako hledání chyb v práci jiných je při testování softwaru velkým plus.

Testování softwaru je užitečné stavět na teoretických základech a aplikovat formalizované postupy, ale mnohdy daleko více udělá intuice a hlavně zkušenosti.

10.1 Závěrečné shrnutí

Mezi cíle této práce patří tedy jednak obeznámení čtenáře s teoretickým základem testování software a jednak pohled na testování praktickýma očima tak, aby znalosti zde načerpané mohl čtenář co nejsnáze uplatnit v praxi.

První z těchto cílů jsem se pokusil dosáhnout v kapitolách 2 – Obecná východiska, 3 – Úvod do testování software a 4 – Role v rámci testování.

V první části druhé kapitoly nazvané Základní pojmy v oblasti testování softwaru jsem se snažil o definování testování softwaru a o objasnění základních pojmů z této oblasti. Ve druhé části druhé kapitoly nazvané Okénko do oblasti vývoje softwaru se zabývám čtyřmi modely vývoje softwaru. Nebylo zde mým cílem popisovat nějakou metodiku, ale spíše praktické přístupy, které jsou při vývoji aplikovány. Třetí část druhé kapitoly nazvaná Testování softwaru v rámci metodiky RUP velmi stručně nastiňuje, jaké místo v projektu vývoje systému může testování zaujímat.

Ve třetí kapitole, nesoucí název Úvod do testování softwaru, se zabývám zejména kategorizací technik testování z hlediska způsobu provedení testů, z hlediska úrovně vývoje a dimenze kvality. Smyslem této kapitoly je kromě popisu samotných technik testování i snaha navést čtenáře na myšlenku, z jakých různých pohledů může být na software a jeho testování nahlíženo a že každý tento pohled má svoje specifika.

Ve čtvrté kapitole popisuji role, které jsou zastoupeny v rámci testování softwaru, jak je definuje metodika Rational Unified Process. Kapitola stručně popisuje náplň jejich práce a jejich odpovědnost.

Druhého z cílů by měly dosahovat kapitoly 5 – Specifikace softwaru, 6 – Plánování testů, 7 – Návrh testových případů, 8 – Ohlašování chyb a 9 – Postup testování.

V kapitole 5 nazvané Specifikace softwaru pojednávám o testování specifikace. Kapitola popisuje náležitosti, které by měla mít specifikace a také různé tipy, jak chyby ve specifikaci vyhledávat. Šestá kapitola pojednává o Plánování testů. Definuje náležitosti plánu testů podle

standardu ANSI/IEEE Standard 829-1998 for Software Test Documentation. Sedmá kapitola pojednává o návrhu testových případů a to jak při testování dat, tak i při testování operací. Vychází při tom ze stanovení optimálního objemu testů a z identifikace tříd ekvivalentních případů. Kapitola osm dává návod, jak efektivně oznamovat chyby v softwaru nalezené. Devátá kapitola slouží jednak jako drobná rekapitulace a jednak pro utvoření lepší představy o logické návaznosti činností v rámci testování softwaru.

10.2 Využití práce v praxi

Mimo začínajících testerů, kteří mohou v této práci načerpat základní znalosti z oblasti testování software, bude práce podkladem pro vytvoření systému testování ve firmě Bellman Group s. r. o. Testování v této firmě bude převedeno do formalizovanější podoby s cílem zefektivnění testování a dosažení co nejlepší kvality vyvíjeného softwaru.

11 Terminologický slovník

Termín	Vysvětlení
Správnost	Program se vstupními daty provádí správné operace
Přesnost	Program poskytuje přesné výstupy vzhledem ke vstupním datům operaci, které s daty provádí
Verifikace	Kontrola, zda program přesně odpovídá jeho specifikaci
Validace	Kontrola, zda program nebo jeho specifikace odpovídá požadavkům uživatele
Testování	Průběžná činnost zaměřená na vyhledávání chyb v softwaru
Zajišťování kvality	Vytváření a prosazování standardů a metod, které vedou ke zvyšování kvality softwaru
Plánování testů	Předpis rozsahu, postupu, prostředků a časového plánu aktivit spojených s testováním, identifikující jednotlivé testované položky, testované funkce a úkoly prováděné při testování, konkrétní osoby odpovědné za každý z úkolů a rizika spojená s definovaným plánem.
Testový případ	Soubor vstupů, testovacích podmínek a očekávaných výstupů pro testovaný software
Případ užití (use case)	Sekvence spolu souvisejících akcí prováděných systémem.
FURPS+	Model definující pět funkcionálních a libovolný počet nefunkcionálních dimenzí kvality
Třída ekvivalentních případů	Množina testových případů ,která je nahraditelná jakýmkoli obsaženým testovým případem, při čemž nedojde ke zvýšení rizika, že v systému zůstanou neodhalené chyby
Hraniční podminky	Prostor okolo hranic oboru hodnot dat, se kterými software pracuje, přičemž tyto hranice jsou z pohledu uživatele zřejmé.
Subhraniční podmínky	Prostor okolo hranic oboru hodnot dat, se kterými software

Termín	Vysvětlení
	pracuje, přičemž tyto hranice nejsou z pohledu uživatele zřejmé.
Test bílé skříňky (white-box test	Test, který je navrhován na základě znalosti vnitřní logiky systému
Test černé skříňky (black-box test)	Test, který je navrhován bez znalosti vnitřní logiky systému
Manuální test	Test, který je proveden ručně testerem, bez použití nástroje automatizovaného testování
Automatizovaný test	Test, který je proveden pomocí nástroje automatizovaného testování
Nástroj automatizovaného testování	Software, který byl vyvinut za účelem provádění automatizovaných testů

12 Seznam použité literatury

[Patton] PATTON, R.: Testování softwaru, vyd. Computer Press, Praha, 2002, ISBN 80-7226-636-5

[Vrazelova] Vráželová, L.: Diplomová práce: Testování software a nástroje pro jeho podporu, Vysoká škola ekonomická v Praze, Fakulta Informatiky a statistiky, Katedra informačních technologií, 2002

[Kunský] Kunský, F., Diplomová práce: Testování software a testovací role, Vysoká škola ekonomická v Praze, Fakulta Informatiky a statistiky, Katedra informačních technologií, 2005

[RUP] Rational Unified Process v7.0 zdroj: http://www-128.ibm.com/developerworks/downloads/r/rup/

[SSTD] ANSI/IEEE Standard for Software Test Documentation, zdroj: http://www.ruleworks.co.uk/testguide/documents/IEEE%20Standard%20for%20Software%20Test%20Documentation..pdf

[ZSI] Buchalcevová, A., Stavovská, I., Šimůnek, M.,: Základy softwarového inženýrství – základní témata, Vysoká škola ekonomická v Praze, Nakladatelství Oeconomica 2002, ISBN 80-245-0346-8

[STWik] Software Testing, zdroj: http://en.wikipedia.org/wiki/Software_testing

[TDDWik] Test-Driven Development: zdroj: http://en.wikipedia.org/wiki/Test-driven_development

[Pettichord] Pettichord, B., Testers and Developers Think Differently, STQE Magazine, 2000, zdroj: http://www.io.com/~wazmo/papers/testers_and_developers.pdf

[Hower] Hower, R., Software QA and Testing Frequently-Asked-Questions, 2006, zdroj: http://www.softwareqatest.com/qatfaq1.html, http://www.softwareqatest.com/qatfaq2.html

[Hower2] Hower, R., Software QA and Testing Less-Frequently-Asked-Questions, 2006, zdroj: http://www.softwareqatest.com/qat_lfaq1.html