Petr Roudenský, Anna Havlíčková

Řízení kvality softwaru

Průvodce testováním

Návrh testů a jejich realizace

Reportování chyb a management testování

Měření kvality softwaru

Automatizované testování a vhodné nástroje

computer press

Petr Roudenský, Anna Havlíčková

Řízení kvality softwaru
Průvodce testováním

Computer Press
Brno
2013

Řízení kvality softwaru
Průvodce testováním

Petr Roudenský, Anna Havlíčková

Obálka: Martin Sodomka
Odpovědný redaktor: Martin Herodek
Technický redaktor: Jiří Matoušek

Objednávky knih:
http://knihy.cpress.cz
www.albatrosmedia.cz
eshop@albatrosmedia.cz
bezplatná linka 800 555 513
ISBN 978-80-251-3816-8

Vydalo nakladatelství Computer Press v Brně roku 2013 ve společnosti Albatros Media a. s. se sídlem Na Pankráci 30, Praha 4. Číslo publikace 18 134.

1. vydání

ALBATROS MEDIA a.s.

Obsah

Úvod

Pokud se ohlédneme do minulosti, testování softwaru je staré jako vývoj softwaru sám. V dobách prvních programů, zhruba od padesátých let dvacátého století, bylo ověřování jejich správnosti velmi náročnou a obtížnou úlohou prováděnou důkladným manuálním zkoumáním samotných programů i jejich výstupů, neboť výpočetní síla byla velmi omezená a drahá. Jak se zvyšovala dostupnost a výkonnost počítačů, rostla i komplexita programů a brzy začalo být zřejmé, že kompletní otestování netriviálních programů není možné, především pro vysoký počet vstupů či možných průchodů kódem. K řešení tohoto problému začaly být používány různé techniky umožňující redukovat počet prováděných testů.

V průběhu šedesátých a sedmdesátých letech se chápání testování změnilo a původní cíl, prokazování správnosti softwaru, se změnil na cíl zcela opačný, totiž hledání chyb v programu. Během následujících dvou dekád začalo být na testování nahlíženo více i v souvislosti s měřením kvality a jeho aktivity byly rozšířeny také na další produkty vývoje, jako jsou specifikace či návrhové dokumenty. Zvyšující se potřeba systematického přístupu pro stále složitější systémy byla dočasně uspokojena novými přístupy v oblasti návrhu testovacích případů a také prvními nástroji pro automatizované testování.

A pokud se podíváme na současný svět, testování softwaru rozhodně nedosáhlo žádného statického stavu – nové platformy, programovací jazyky, technologie, trendy i možnosti, jaké máme k dispozici, se stále mění. Testování mobilních aplikací, testování využívající cloud, virtualizace nebo efektivní využití stále sofistikovanějších automatizačních nástrojů jsou jen některá témata posledních let.

Kvalita používaného softwaru je jednou z hlavních konkurenčních výhod a tím i atributem úspěšné existence mnoha ekonomických subjektů. Z toho důvodu bylo během posledních tří desetiletí vytvořeno mnoho různých standardů zabývajících se dosažením kvality, nástrojů pro efektivnější řízení testování, začaly být pořádány mezinárodní specializované konference pro pracovníky v oblasti testování a v neposlední řadě také vyšla řada knih.

Jak můžeme vidět, podoba testování softwaru se pochopitelně v čase mění a stejně tak i jeho chápání, ovšem bez ohledu na to zůstává stále nepostradatelnou součástí životního cyklu vývoje softwaru, umožňující měření a – přestože nepřímo – i zlepšování kvality vyvíjených systémů, jež jsou v určité podobě dnes prakticky neoddělitelnou součástí našich životů.

Poděkování

Rád bych poděkoval za pomoc v době, kdy vznikala tato kniha, MUDr. Jiřímu Piťhovi a prof. Dr. Mamede de Carvalho (PR).

Co je obsahem této knihy

Z námi provedených odborných šetření (AH) bylo zjištěno, že důraz na testování, vzdělávání pracovníků v této oblasti a používání moderních nástrojů je stále nižší, než jak je tomu v zahraničí. Svůj podíl na tomto stavu má jistě i dostupnost literatury. Přestože na zahraničních trzích lze nalézt množství knih zabývajících se problematikou testování softwaru a řízením a zajišťováním jeho kvality, u nás existuje pouze několik titulů, a to ve formě překladu známých prací zahraničních autorů. Tento fakt byl jedním z důvodů, proč jsme využili možnosti napsat tuto knihu.

Ačkoli především v angličtině je dostupná řada kvalitních knižních titulů z posledních let, jejich zaměření bývá často velmi teoretické anebo velmi specifické (automatizace testování, řízení kvality softwaru obecně, testovací techniky a podobně), chybí často zasazení do širšího kontextu, které nepředpokládá výbornou znalost dané problematiky.

S ohledem na výše uvedené bylo naším cílem v této knize nabídnout přehledný a ucelený pohled nejen na testování softwaru, ale i na související aktivity řízení kvality v rámci jejího zajišťování. Právě proto se v nezbytné míře obecněji zabýváme i vnímáním kvality softwaru a jejím chápáním, smyslem technických norem nebo vysvětlením rozdílů mezi zajišťováním a řízením kvality. Tematika testování není omezena jen na vymezení pojmů, představení jednotlivých technik a přístupů k návrhu testovacích případů – úměrně rozsahu knihy probíráme některá sporná tvrzení, zabýváme se praktickými otázkami souvisejícími s plánováním, prováděním i měřením testů, problematikou automatizace funkčních i zátěžových testů, a je-li to vhodné, upozorňujeme na odlišnosti klasického a agilního přístupu.

Teoretický výklad, zejména u technik návrhu testovacích případů, doplňujeme praktickými příklady i vyobrazením, jež by měly sloužit ke snadnému pochopení vyložené látky. Poslední kapitola umožňuje čtenáři vyzkoušet si krok za krokem práci se dvěma odlišnými nástroji automatizovaného testování.

Při psaní této knihy jsme využívali nejen osobních zkušeností, ale i dat z vlastních výzkumů, přednášek a také zajímavostí nebo doporučení z osobní korespondence s předními specialisty v této oblasti. V případě potřeby odkazujeme na literaturu nabízející podrobný výklad dané tematiky či dokládající určitá tvrzení, neboť mezi autory i normami lze v určitých místech nalézt výrazné rozdíly.

Pro koho je tato kniha určena

Obsah i uspořádání knihy je navrženo tak, aby byla vhodná primárně pro začínající i zkušenější pracovníky v oblasti řízení kvality, především pak pochopitelně testery, analytiky testování a částečně také pro manažery testování. Přínosná však může být i pro pracovníky zajišťování kvality – jednak proto, že testování jako proces spadá do oblasti jejich působnosti a také z toho důvodu, že nezřídka se některé jejich aktivity překrývají s aktivitami řízení kvality. Zvládnutí témat v této knize lze považovat za dostatečné pro práci na juniorských pozicích testerů a test analytiků běžných (myšleno nevyžadujících doménově specifické znalosti) systémů a zároveň slouží jako vhodný základ pro další specializaci v jednotlivých oblastech testování.

Zpětná vazba od čtenářů

Nakladatelství a vydavatelství Computer Press, které pro vás tuto knihu připravilo, stojí o zpětnou vazbu a bude na vaše podněty a dotazy reagovat. Můžete se obrátit na následující adresy:

Computer Press
Albatros Media a.s., pobočka Brno
IBC
Příkop 4
602 00 Brno

nebo

sefredaktor.pc@albatrosmedia.cz

Computer Press neposkytuje rady ani jakýkoli servis pro aplikace třetích stran. Pokud budete mít dotaz k programu, obraťte se prosím na jeho tvůrce.

Errata

Přestože jsme udělali maximum pro to, abychom zajistili přesnost a správnost obsahu, chybám se úplně vyhnout nelze. Pokud v některé z našich knih najdete chybu, budeme rádi, pokud nám ji oznámíte. Ostatní uživatele tak můžete ušetřit frustrace a pomoci nám zlepšit následující vydání této knihy.

Veškerá existující errata zobrazíte na adrese *http://knihy.cpress.cz/K2028* po klepnutí na odkaz Soubory ke stažení.

KAPITOLA 1

Co je třeba znát aneb důležité pojmy

V této kapitole:
- Krátce o požadavcích
- Stakeholdeři
- Technické normy
- Specifikace a návrhové dokumenty
- Agilní vývoj
- Minimum ze slovníku programátora

V této úvodní kapitole se blíže podíváme na některé pojmy, s nimiž se budeme v různé míře setkávat v následujících kapitolách této knihy. Její prostudování doporučujeme obzvláště těm, kteří v oblasti informačních technologií zatím nemají mnoho zkušeností, ovšem přínosná může být i pro ty zkušenější.

Budeme se stručně zabývat požadavky na software, technickými normami, významnými dokumenty používanými při vývoji systémů, základy agilního vývoje a také pojmy z oblasti programování. Ty jsou totiž užitečné nejen k pochopení některých příkladů, ale pro práci v oblasti testování a řízení kvality softwaru je považujeme za prakticky samozřejmé.

Krátce o požadavcích

V této části si shrneme to hlavní, co bychom měli vědět o požadavcích na softwarový systém. Důvodem pro zařazení tohoto tématu je – kromě zřejmého významu požadavků z pohledu testování – fakt, že v ideálním případě se pracovníci zodpovědní za testování účastní projektu již od samého počátku a provádí analýzu vytvářených dokumentů, aby případné problémy zachytili co nejdříve.

Požadavky vyjadřují přání zákazníka, respektive budoucího uživatele. Popisují určité funkce či vlastnosti, které od vytvářeného systému očekávají. Specifikují „co“, nikoli však „jak“ to bude realizováno.

Požadavky dělíme na dva základní typy:

- *Funkční požadavky* – popisují funkčnost služby poskytované systémem, tedy co by měl vykonávat.
- *Mimofunkční (nefunkční) požadavky* – popisují určité vlastnosti systému či omezující podmínky. V podstatě říkají, jaký by systém měl být.

Pro lepší představu si uvedeme několik příkladů různých funkčních požadavků, specifikujících chování softwarového produktu:

- Uživatel bude moci vytvořit záznam pro nového zákazníka.
- Program umožní uživateli vyhledávat zboží podle výrobního čísla.
- Systém automaticky odhlásí uživatele po 3 minutách nečinnosti.
- Program umožní uživateli exportovat data do textového souboru.

Oproti tomu mimofunkční požadavky se týkají např. bezpečnosti, spolehlivosti, použitelnosti, výkonnosti, kompatibility, přenositelnosti a podobně. Nemusí se týkat jen samotného produktu, ale i procesu jeho vývoje. Příklady mimofunkčních požadavků tak mohou být:

- Doba odezvy pro zobrazení detailu objednávky nepřekročí 3 sekundy.
- Modul „Správa klientů" bude přístupný jen uživatelům s rolí správce.
- Dokumentace k použití systému bude v takovém rozsahu, aby nebylo nutné školení uživatelů.
- Systém bude použitelný při zátěži 1 000 uživatelů.
- Ke komunikaci mezi serverem a klientem bude použito protokolu XYZ.

Co se týká zdrojů požadavků, tak nejde pouze o přání uživatelů, jak by se mohlo zdát. Ve skutečnosti jsou zdroje požadavků závislé na povaze projektu, a tak mezi ně patří také například:

- Zákony či nařízení čili legislativa
- Znalost dané domény a z toho vyplývající požadavky
- Existující (i konkurenční) software
- Pracovní aktivity uživatelů, implikující jejich potřeby
- Hardwarová a softwarová omezení
- Prostředí, kde bude systém provozován

Nezřídka se setkáváme s tím, že dodané požadavky nejsou formulovány správně. Vedle nekonzistence, neproveditelnosti či nekompletnosti požadavků se často setkáváme s následujícími vadami:

- Netestovatelnost. Příkladem může být požadavek typu „Podržení kurzoru myši dostatečně dlouho nad objektem zobrazí jeho název". Při implementaci a testování bychom

tedy mohli jen odhadovat, co je podle autora „dostatečně dlouho", a zda tedy produkt požadavek splňuje či nikoli. Takový požadavek tak samozřejmě nelze akceptovat.

- Nejednoznačnost. Uvažujme požadavek typu „Při souboji s koncovým nepřítelem bude po zničení hráč přesunut do lokace XYZ". Měl autor požadavku na mysli zničení hráče, nebo nepřítele? Požadavky musí být jasné a jednoznačné – pokud je lze interpretovat více způsoby, jde o chybu.
- Více požadavků zapsaných jako jeden. Například požadavek „Uživatel bude moci vložit obrázek nebo odkaz na obrázek" není správný. Pokud bude fungovat jen vložení obrázku, ale nikoli odkazu, je splněna polovina požadavku…

Je třeba říct, že správně zvládnuté aktivity související se správou požadavků silně ovlivňují úspěšnost celého projektu. Jsou-li požadavky nesprávné (ať už vinou chybné interpretace analytikem, nedostatečného zapojení uživatelů či neúčasti klienta v tomto procesu vůbec), systém bude budován na špatných základech. Je zřejmé, že odstraňování takto vzniklých problémů bude velmi časově i finančně náročné, jak bude zmíněno později v souvislosti s náklady na odstranění defektů.

Stakeholdeři

Často používaným termínem, se kterým se budeme setkávat zejména v managementu testování, je slovo „*stakeholder*", pro který čeština nemá ustálený překlad. Ve své podstatě jsou to všichni ti (jednotlivci, skupiny nebo i organizace), kteří mají na produktu určitý zájem či mohou ovlivnit nebo být ovlivněni jeho použitím anebo vývojem. Mohli bychom tedy říct, že jde o zainteresované strany nebo zájmové skupiny.

Zákazník, projektový manažer, vedoucí vývoje, marketingové oddělení a tak dále, ti všichni jsou stakeholdery, avšak jejich zájmy budou poněkud odlišné. Zapojení správných stakeholderů je klíčové především v oblasti hodnocení rizik.

Technické normy

U některých témat v této knize naleznete odkazy na konkrétní technické normy. Je tedy na místě zmínit, co to technické normy vlastně jsou, jaký je jejich smysl a které organice se zabývají jejich tvorbou.

Technické normy jsou společně – za účasti zástupců všech zainteresovaných stran – dohodnutá doporučení stanovující základní požadavky na kvalitu, vhodnost použití k zamýšlenému účelu a případně i bezpečnost (či slučitelnost apod.) výrobku, procesu nebo služby. Technické normy tak obsahují především definice a pravidla, technické charakteristiky či řešení, specifikace nebo další pokyny.

Poznámka: Texty norem nejsou veřejně přístupné zdarma, jak by se mohlo zdát.

Protože jde o doporučení, nejsou technické normy samy o sobě závazné, a jejich používání je tak čistě dobrovolné. Závaznými se však mohou stát, jestliže na ně odkazuje právní předpis nebo pokud jsou uvedeny ve smlouvě, což začíná být stále častější i u nás. Dodržování technických norem je však výhodné, neboť výrobci poskytuje určitou jistotu týkající se jeho produktu (kompatibilita, srovnání s ostatními produkty). Stejně tak ale přináší výhody pro zákazníka/uživatele, který tak má zaručenu jistou míry kvality či záruku týkající se bezpečnosti a podobně. Soulad s normou je tedy pochopitelně v zájmu zákazníků.

O normalizační proces na nadnárodní úrovni se starají především tyto organizace:

- *ISO* – Mezinárodní organizace pro normalizaci (International Organization for Standardization)
- *IEC* – Mezinárodní elektrotechnická komise (International Electrotechnical Comission)
  - Společná technická komise *ISO/IEC JTC1* Informační technologie (Information Technology)

Další významnou organizací vydávající globální normy je *IEEE* – Institut pro elektrotechnické a elektronické inženýrství (Institute of Electrical and Electronics Engineers), původně působící pouze v USA.

V České republice normy vydává ČNI – Český normalizační institut. V oblasti IT normy pouze přejímá, a to překladem či převzetím originálu s nutnými údaji či přílohou v češtině, ale s původním textem.

Specifikace a návrhové dokumenty

K vývoji softwaru v *tradičním* pojetí patří i řada dokumentů, jejichž význam je vhodné znát. V této části se tak v krátkosti seznámíme s těmi, jež se nezabývají přímo testováním – ty budou rozebrány později v příslušných kapitolách.

- *Studie proveditelnosti (Feasibility Study – FS)* – Existuje-li záměr vytvořit nový či modernizovat stávající informační systém, mezi počátečními kroky je i zpracování analýzy s názvem studie proveditelnosti. Jejím cílem je zjistit, zda lze projekt uskutečnit s danými omezeními, riziky a v časovém rámci a zda je to vůbec výhodné z ekonomického hlediska.
- *Specifikace požadavků na software (Software Requirements Specification – SRS)* – Analýza, specifikace a dokumentace požadavků končí vypracováním dokumentu nazývaného specifikace požadavků na software. Ten obsahuje kromě všeobecného popisu systému (účel, kontext…) i funkční a mimofunkční požadavky, tedy kompletní popis chování budoucího systému. Správnost tohoto dokumentu je klíčová, neboť slouží jako vstup či reference pro ostatní dokumenty.

- *Návrhový dokument (Design Document)* – Po dokončení specifikace požadavků na software je ve fázi návrhu vypracován návrhový dokument, který popisuje, jak by měl být software implementován, aby vyhovoval požadavkům. Návrhový dokument obsahuje například návrh databáze, datových struktur, objektů a podobně.

Agilní vývoj

V předchozí části zaměřené na dokumenty jsme u softwarového vývoje záměrně zvýraznili slovo tradiční, neboť dnes se velmi často setkáváme s *agilním* vývojem, jehož hodnoty a principy jsou poněkud odlišné. Co je tedy agilní vývoj a jaký je hlavní rozdíl oproti tradičnímu přístupu, to ve stručnosti zmíníme v následujících odstavcích.

Agilní přístup, jak napovídá samotný název, znamená adaptabilitu, flexibilitu či přizpůsobivost změnám, které velmi často přichází během vývoje, což bývá pro tradiční přístupy často problematické. Hodnoty a principy agilního vývoje byly v roce 2001 formulovány IT odborníky pracujícími s alternativními způsoby vývoje do tzv. Agilního manifestu. Ten jako hodnoty uvádí následující:

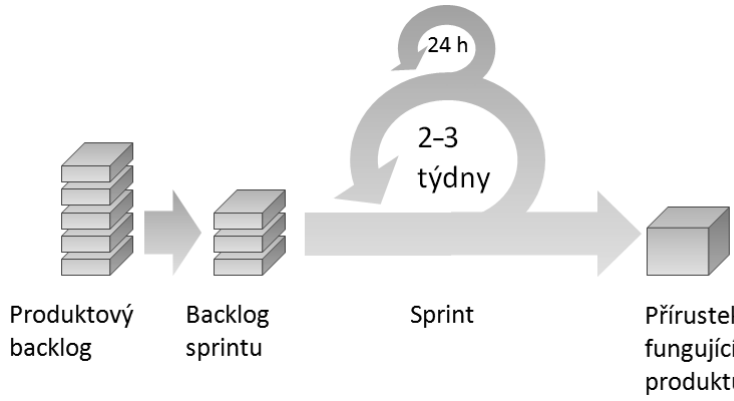
- Lidé a jejich vzájemná interakce jsou důležitější než procesy a nástroje.
- Fungující software je důležitější než vyčerpávající dokumentace.
- Spolupráce se zákazníkem má přednost před vyjednáváním smluv.
- Je důležitější reagovat na změny než dodržovat plán.

Oproti tradičnímu způsobu vývoje je tak agilní přístup silně orientovaný na jednotlivé členy týmu, jejich schopnosti a interakci. Do pozadí naopak ustupuje striktní dodržování procesů, používání daných nástrojů či vytváření jiné než nezbytně nutné dokumentace. Agilní vývoj je proto jen málo formální a klade především důraz na doručování funkčního softwaru, což slouží i jako metrika pokroku na projektu.

Poznámka: Není však pravda, že v agilním vývoji se vůbec nedokumentuje či neplánuje, jak se někdo mylně domnívá.

Agilní přístupy využívají kratších iterací, kdy jednotlivé funkční celky jsou dodávány v časovém rámci většinou 2 až 4 týdnů. Požadavky se implementují podle zákazníkem určené priority. Zásadní je pak cena a časová náročnost odvozená z jejich odhadnuté obtížnosti. To tvoří omezení, za kterého má tým během jednotlivých iterací doručit co nejvíce implementovaných požadavků.

Mezi frameworky agilních přístupů patří například velmi známý a populární *Scrum* nebo *Extreme Programming*.



Obrázek 1.1 Scrum – základní schéma. Produktový backlog obsahuje požadavky na funkce systému řazené dle priority a s odhadovanou náročností na implementaci. Do jednotlivých iterací (tzv. sprintů) si tým odebírá úkoly tak, aby je (ideálně) vždy stihl dokončit, takže na konci sprintu je přírůstek fungujícího kódu. Každý den probíhá rychlé setkání týmu, kde každý člen shrne svou práci od posledního setkání, práci na daný den a případně překážky, které mu brání v pokračování.

Minimum ze slovníku programátora

Při probírání určitých témat není možné zcela se vyvarovat popisu, ukázky či vysvětlení obsahujícího ryze technické pojmy, se kterými se setkáme především při programování. Jak se ale dozvíme v následujících kapitolách, při testování je dnes často nutné mít o programování více než jen povědomí, a tak alespoň minimální znalost některých pojmů je často nezbytností. V této části se tedy podíváme (bez zbytečné teorie) na ty základní z nich, které považujeme za důležité a ve své podstatě univerzální. Čtenáři znalí programování mohou samozřejmě pokračovat rovnou další kapitolou.

- Bit je základní jednotkou informace, nabývající hodnot 0 a 1.
- Byte je jednotka informace složená z 8 bitů, může tedy nabývat 256 hodnot (2^8).
- Proměnná je úsek v paměti sloužící pro uchování informací. Je určitého datového typu a má programátorem přiřazen jednoznačný identifikátor – jméno, pomocí kterého je pak možné přistupovat k hodnotě v ní uložené.
- Datový typ určuje, jakým způsobem je proměnná uložena v paměti a jaké operace s ní lze provádět. Například mezi primitivní datové typy jazyka C# patří:
  - *bool* – obsahuje logickou hodnotu „true“ (pravda) či „false“ nepravda
  - *short* (či int16) – celočíselný typ (2^{16} hodnot)
  - *long* (či int64) – celočíselný typ (2^{64} hodnot)

- *string* – řetězec Unicode znaků čili text
- *char* – v jazycích, jako je Java či C#, jde o 16 bitový znak Unicode, v C/C++ o 8bitový znak ASCII.

Poznámka: Názvy datových typů se mezi jednotlivými programovacími jazyky mohou lišit.

- *Ladění (Debugging)* je proces umožňující mimo jiné hledat logické chyby za běhu programu. Vývojář může procházet program příkaz po příkazu („krokování“) či se zaměřit jen na určitou jeho část, prohlížet hodnoty proměnných a mnoho dalšího.
- *Výjimka (Exception)* je často používaný termín. Jde o mimořádnou (tedy výjimečnou) událost vzniklou za běhu programu a většinou ovlivňující jeho další pokračování. Výjimka tedy představuje určité selhání, jde o důsledek chyby.
- *Funkce a metoda*. *Funkce* je pojmenovaný blok kódu, který jako sled příkazů vykonává určitou konkrétní úlohu, typicky popsanou právě v jejím názvu. Funkce může přijímat různé parametry, po dokončení vracet hodnotu a lze ji volat opakovaně. V objektově orientovaném programování mluvíme o *metodách*, což jsou vlastně funkce, které však vždy náleží určité třídě (funkce třídy či členské funkce).
- *Parametr* a *argument* jsou vcelku často vzájemně zaměňované pojmy, ovšem jejich význam je rozdílný. Chceme-li, aby funkce přijímala data určitého typu, při její deklaraci zavedeme proměnnou s určitým názvem a datový typem, kterou nazýváme parametr. Ten v podstatě definuje, jakou hodnotu funkce očekává. Při volání této funkce jí pak musíme předat hodnotu, jež odpovídá typu parametru, přičemž tato hodnota je právě argumentem. Někdy se také můžeme setkat s označením formální a skutečný parametr.
- *Sestavení (build)* označuje proces, který ze zdrojových kódů a případných ostatních zdrojů projektu vytvoří výsledný produkt (například spustitelnou aplikaci, v binární podobě). Stejným názvem je ale v češtině také označován výsledný produkt.
- *Integrované vývojové prostředí* (Integrated Development Environment - IDE) je software sloužící k vývoji aplikací, integrující všechny potřebné nástroje pro psaní kódu, jeho ladění, kompilaci (překlad), sestavování a často i např. jednotkové testování a podobně do uživatelsky přívětivého prostředí.
- *Log.* Záznam aktivity systému za jeho provozu umožňující zpětnou analýzu jeho činnosti. Logování je často využíváno k diagnostice chybových stavů.

KAPITOLA 2

Základní koncepty v oblasti kvality softwaru

V této kapitole:
- Kvalita softwaru a role testování
- Zajišťování a řízení kvality
- Verifikace a validace
- Statické a dynamické techniky

Protože se tato kniha zabývá testováním softwaru, považujeme za samozřejmé nejprve seznámit čtenáře s některými základními koncepty a pojmy, které se týkají kvality softwaru a způsobů, jak ji lze hodnotit.

Vzhledem k zaměření naší knihy nejsou témata v této kapitole rozebírána detailně, avšak dostatečně na to, aby čtenář získal o dané problematice dobré povědomí. Nutno zmínit, že literatura zabývající se testováním a kvalitou softwaru často není jednotná v používání terminologie i chápání některých pojmů, proto jsme se snažili vycházet z akceptovaných technických norem, bylo-li to možné. Avšak nakonec je třeba si uvědomit, že klasifikace či definice pojmů a technik není v praxi tolik důležitá, jestliže jim porozumíme a dokážeme je správně aplikovat.

Kvalita softwaru a role testování

Software je v určité podobě dnes nedílnou součástí našeho každodenního života, ať přímo či nepřímo. Se stále hlubší integrací a zvyšující se složitostí i nákladností jak podpůrných, tak kritických systémů logicky rostou i požadavky na jejich kvalitu. Jak lze ale kvalitu softwarového produktu chápat?

Ačkoli slova „kvalita“, „kvalitní“ či „nekvalitní“ používáme běžně, každý si pod těmito pojmy představuje něco poněkud odlišného. Koncept kvality je závislý na kontextu a velmi

komplexní ve svém pojetí, jak názorně ilustruje práce založená na rozboru vnímání kvality zpracovaném Davidem Garvinem [Kitchenham, Pfleeger, 1996].

Kvalita je zde analyzována z pěti různých pohledů:

- *Transcendentálního*: Na kvalitu nahlíží jako na něco, co lze rozpoznat, ale jen velmi obtížně definovat. Jinými slovy - poznáme, když je něco kvalitní, přestože svůj úsudek nedokážeme zcela zdůvodnit.
- *Uživatele*: Z této perspektivy kvalitou rozumíme, že produkt je vhodný pro zamýšlené použití, tedy že splňuje požadavky a potřeby uživatele.
- *Výroby*: Úroveň kvality je v tomto případě určena mírou rozsahu, v jakém produkt splňuje specifikace.
- *Produktu*: Z tohoto pohledu se kvalita váže na inherentní charakteristiky produktu (vnitřní kvality), které určují kvality vnější.
- *Cenového*: Kvalita závisí na tom, kolik je zákazník ochoten za ni zaplatit.

Snahy definovat kvalitu softwaru a její atributy daly vzniknout různým modelům kvality v rámci mezinárodních norem, především řady ISO/IEC 9126 (Softwarové inženýrství – jakost produktu). Dalšími normami pro tuto oblast byla řada ISO/IEC 14598 (Softwarové inženýrství – hodnocení softwarového produktu) a norma ISO/IEC 12119 (Softwarové balíky - Požadavky na jakost a zkoušení). Uvedené normy však trpí vážnými nedostatky (zastaralé, nekonzistentní…), a tak jsou postupně nahrazovány jednotným systémem norem ISO/IEC 25000–25099 v rámci projektu *SQuaRe* (*Software Quality Requirements and Evaluation* – možno přeložit jako „Požadavky na kvalitu softwaru a její hodnocení“).

Poznámka: Pojmy „jakost“ a „kvalita“ jsou zde používány jako synonyma, avšak vyjma citací budeme nadále používat jen pojem „kvalita“.

Podívejme se velmi stručně na již vydanou normu ISO/IEC 25010, podle níž je kvalita softwaru *míra, do jaké softwarový produkt splňuje stanovené a implicitní potřeby, je-li používán za stanovených podmínek*. Tato norma definuje model kvality produktu (vnitřní a vnější) a model kvality užití. Kvalita softwarového produktu zde sestává z osmi charakteristik (namísto původních šesti):

- Funkčnost
- Účinnost
- Kompatibilita (přidáno)
- Použitelnost
- Bezporuchovost
- Bezpečnost (přidáno)
- Udržovatelnost
- Přenositelnost

Poznámka: Překlady definic a pojmů v této části nejsou ustálené, a nelze je tak považovat za oficiální.

Každá charakteristika má navíc další podcharakteristiky, které jsou předmětem měření, přičemž jednou z měr je i testování. *Testování tedy poskytuje informace o kvalitě produktu.*

FURPS

Známý model kvality nazývaný FURPS je výsledkem přístupu k definici a měření jednotlivých atributů kvality dodaného systému, jenž byl vytvořen společností Hewlett-Packard. Název FURPS je akronym složený z prvních písmem pěti atributů kvality, které popisuje a jimiž jsou:

- *Functionality (funkčnost)*. Soubor požadované funkcionality, schopností a bezpečnostních aspektů systému.
- *Usability (použitelnost)*. Vnímání systému člověkem – snadnost použití, konzistence, estetika, dokumentace a podobně.
- *Reliability (spolehlivost)*. Četnost a závažnost selhání, doba bezporuchového provozu, správnost výstupu, zotavení…
- *Performance (výkonnost)*. Odezva systému, výkon za různých podmínek, požadavky na systémové prostředky a tak dále.
- *Supportability (rozšiřitelnost/podporovatelnost)*. Zde jde o kombinaci různých vlastností, jako je například škálovatelnost, udržovatelnost, testovatelnost či snadnost konfigurace.

Jak můžeme vidět, z pohledu požadavků jde o jednu skupinu funkčních a čtyři skupiny mimofunkčních požadavků.

Dnes se většinou setkáváme s tímto modelem v jeho rozšířené podobě s názvem FURPS+, jež přidává několik dalších kategorií:

- *Design constraints (omezení návrhu)*. Stanovení určitých omezení, která jsou na návrh produktu kladena.
- *Implementation requirements (požadavky na implementaci)*. Vymezení použitých programovacích jazyků, standardů, omezení zdrojů a podobně.
- *Interface constraints (požadavky na rozhraní)*. Popisuje externí systémy, s nimiž bude produkt interagovat, a definuje potřebná omezení pro rozhraní (formátová, časová…).
- *Physical requirements (požadavky na fyzické vlastnosti)*. V případě hardwarových systémů může jít o rozměry, váhu, použitý materiál či tvar.

Zajišťování a řízení kvality

V této krátké části bychom rádi uvedli na pravou míru poměrně značně rozšířené přesvědčení, že zajišťování kvality softwaru (*Software Quality Assurance* – SQA) je vlastně jen jiným označením pro jeho testování, což neodpovídá skutečnosti.

Zajišťování kvality softwaru – kam budeme počítat i plánování - je totiž zaměřeno na kvalitu procesů *celého* životního cyklu softwaru, která ve výsledku pochopitelně silně ovlivňuje kvalitu samotného produktu. Mezi hlavní aktivity tak patří definice, zavedení procesů (včetně používaných norem, procedur, metrik, nástrojů…) a následně kontrola jejich dodržování a hodnocení s cílem nalézt případná zlepšení. Zajišťování kvality se tak snaží primárně předcházet vzniku defektů. Efektivním nástrojem zajišťování kvality je například audit procesů.

Oproti tomu *řízení kvality softwaru* (*Software Quality Control*) je zaměřeno na výstupy z jednotlivých procesů (dílčí produkty jako dokumentace, kód, spustitelný produkt…), u kterých ověřuje, zda odpovídají specifikacím a (i implicitním) požadavkům. Řízení kvality tak využívá kromě samotného testování produktu také techniky statické analýzy – revize, inspekce či strukturované procházení, které budou zmíněny dále. Řízení kvality je orientováno na nalézání defektů, jejich odstranění a následné ověřování správnosti provedených změn.

Poznámka: Někteří autoři – [Srinivasan, Gopalaswamy, 2005], [Agarwal, Tayal, 2010] – uvádí techniky statické analýzy jako nástroje zajišťování kvality, nikoli řízení kvality. Tyto techniky totiž umožňují vidět „dovnitř", a lze tak posoudit, zda je například kód v souladu se standardy a podobně, což odkazuje na hodnocení procesu. Na druhou stranu jde o zkoumání výstupu, což odpovídá kontrole kvality – [Malhotra, Tiple, 2008], [Kasse, 2008]. V praxi záleží spíše na tom, kdo danou aktivitu provádí a s jakým cílem,

Protože zajišťování kvality definuje a zavádí procesy a aktivity i pro řízení kvality, lze jej považovat za jemu nadřazené. Přestože tedy testování je z tohoto pohledu součástí zajišťování kvality (a poskytuje zpětnou vazbu pro zlepšení), nelze jej jako takové považovat za samotné zajišťování kvality.

Verifikace a validace

V souvislosti s kontrolou a zajišťováním kvality softwaru se často setkáváme s pojmy verifikace a validace, někdy zkráceně a souhrnně označované jako V&V aktivity. Jde o velmi blízké, avšak nikoli zaměnitelné koncepty, které jsou na sobě závislé.

Nejprve je však třeba upozornit, že chápání i definice obou pojmů se mezi autory odborné literatury často rozchází, což má za důsledek nezřídka velmi odlišně vymezené aktivity, které pod validaci či verifikaci spadají.

Verifikační a validační aktivity ověřují, zda software odpovídá specifikacím a jako celek poskytuje přesně to, co uživatel potřebuje – tedy zda splňuje svůj účel. Velmi populární se staly neformální definice podle Barryho W. Boehma, se kterými se setkáváme ve většině textů zabývajících se touto problematikou:

- Verifikace: Vytvářím produkt správně?
- Validace: Vytvářím správný produkt?

Verifikaci lze chápat jako proces, jehož cílem je ověření, že dílčí produkt vývoje softwaru náležitě odráží specifikované požadavky. Pokud bychom vycházeli z normy ISO/IEC 12207:2008, pak verifikační aktivity zahrnují:

- Verifikaci samotných požadavků. Ty musí být konzistentní, proveditelné, kompletní, testovatelné.
- Verifikaci návrhu. Ověření, že návrh systému je správně odvozen z požadavků, přičemž vazby je možné dohledat.
- Verifikaci zdrojového kódu. Kód odpovídá návrhu, je správný, testovatelný, v souladu se stanovenými standardy, umožňuje dohledat vazbu k jednotlivým požadavkům.
- Verifikaci integrace. Zde je cílem ověření, že jednotlivé moduly jsou správně a kompletně integrovány do systému jako celku.
- Verifikaci dokumentace. Ověření, že dokumentace je úplná, v souladu s požadavky, konzistentní.

Verifikace tak zajišťuje, že výstupy jednotlivých fází vývoje odpovídají vstupům – specifikacím odvozeným z požadavků. Pro výše uvedené verifikační aktivity lze využít následující techniky statické analýzy:

- Manuální revize různého stupně formálnosti (řazeno sestupně):
  - Inspekce
  - Technické revize
  - Strukturované procházení dokumentů
  - Neformální revize
- Kontrola využívající nástroje pro automatickou statickou analýzu

Poznámka: Více o statických a dynamických technikách se dozvíte níže.

Jestliže má verifikace potvrzovat soulad dílčího produktu se specifikacemi, může být třeba provést i testování jako formu dynamické verifikace. Cílem této činnosti je nalézt odchylky od specifikací.

Oproti tomu validací rozumíme potvrzení, že dílčí produkt vývoje softwaru je správný s ohledem na požadavky na jeho zamýšlené použití. Jinými slovy, že funguje dle očekávání zákazníka, přičemž z jeho pohledu je také daný produkt testován. Problémy nalezené během validačního testování indikují problém s požadavky – produkt může zcela odpovídat specifikacím, ale neodpovídá představě zákazníka.

Nejvýraznějším příkladem validace je akceptační testování, během kterého se ověřuje, že dodaný software vykonává skutečně to, co zákazník potřebuje. Jak jsme naznačili výše, může se stát, že verifikovaný software nebude validován. Proto je důležité, aby se validačních aktivit zákazník účastnil, co nejdříve je to možné, což je jedna z výhod týmů pracujících v agilním prostředí.

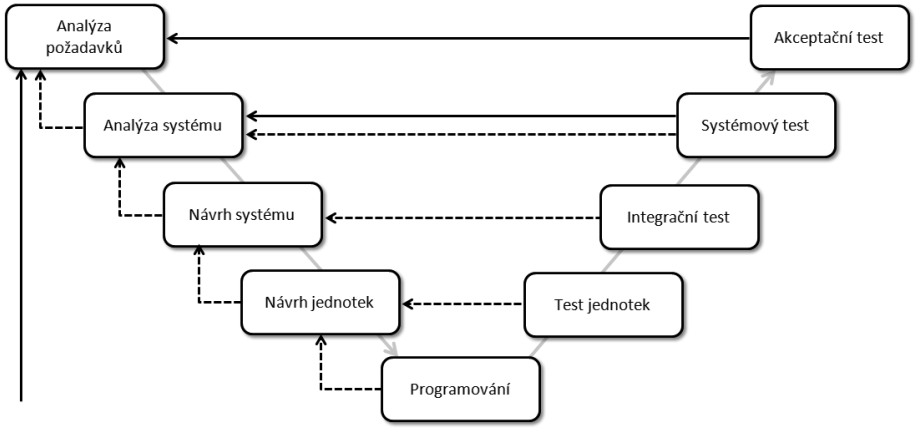
Můžeme uvést příklad z praxe, kdy předmětem vývoje byl software na zpracování a analýzu signálu elektromyografu (EMG), přístroje využívaného neurology k měření elektrické aktivity svalů, což umožňuje identifikovat a lokalizovat poškození nervů či jimi inervovaných svalů. Verifikační aktivity ověřovaly soulad se specifikacemi, normami a předpisy, správnost přenosu a zpracování signálu, jeho zobrazení ve formě elektromyogramu a tak dále. Avšak přestože hodnoty se jevily v pořádku, nikdo z vývojového týmu nemohl říct s určitostí, že systém vykonává to, co má – konečnou validací by v tomto případě provedl lékař-neurolog, který by vpichem umístil jehlu do svalu a interpretoval získaná data. Ve skutečnosti je však validace podobných systémů složitější.

Z uvedeného vyplývá, že jak verifikace, tak validace zahrnují testování, přestože je prováděno s odlišným cílem. Stejně tak validace není omezena pouze na kompletní produkt, jak je někdy uváděno. Podívejme se například na požadavky:

- Verifikace požadavků ověřuje jejich správnost, úplnost, proveditelnost apod.
- Validace požadavků naopak zjišťuje, zda skutečně zachycují potřeby uživatele.

Verifikační a validační aktivity lze ilustrovat pomocí původního V-modelu (jeden z modelů životního cyklu vývoje softwaru), zobrazujícího názorně vztah mezi jednotlivými fázemi vývoje softwaru a úrovněmi testování. Jak můžeme vidět, verifikace a validace by měly být prováděny již od počátku projektu.

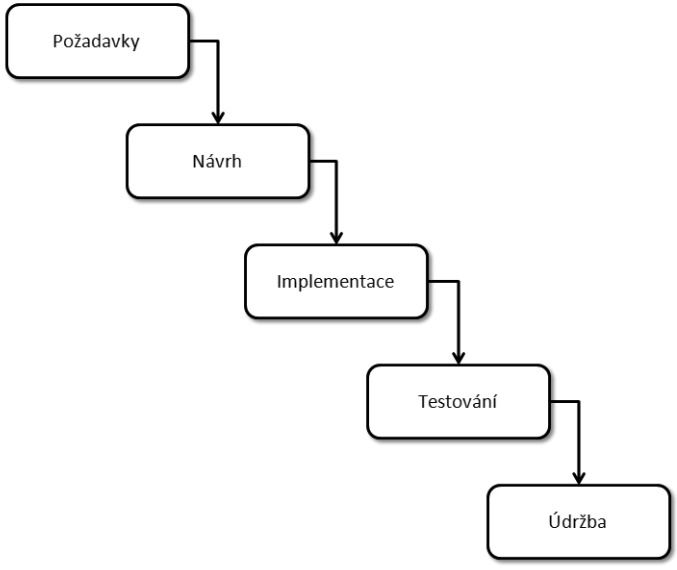
Abychom pochopili přínos V-modelu, je třeba podívat se na vývojový model vodopádu („waterfall model"), ze kterého vychází a jenž je vyobrazen níže. Vidíme, že každá fáze následuje až po dokončení fáze předcházející, což s sebou nese základní problém – jestliže např. došlo k chybné interpretaci požadavku, pak bude daný problém odhalen až ve fázi akceptace zákazníkem. Při opravě tak bude potřeba projít znovu celý cyklus (upravit požadavky, řešení, kód…), což je velmi nákladné. V-model tento nedostatek řeší tak, že výstup z každé fáze prochází ověřením, což umožňuje odhalovat chyby včas a snižuje tak náklady na jejich opravu.



Obrázek 2.1 V-model – všechny fáze vývoje mají odpovídající V&V aktivity. Plná čára představuje validační aktivity, přerušovaná pak ty verifikační

Významným rysem V-modelu je účast aktivit testování prakticky od počátku projektu a souběžně s vývojem, což se týká především produkce artefaktů testování (např. během fáze analýzy požadavků jde o návrh akceptačního testu a podobně).

Poznámka: Pro vodopád i V-model existují modifikované verze řešící některé jejich nedostatky.



Obrázek 2.2 Klasický vývojový model vodopád

Zejména u kritických projektů může být vyžadováno, aby verifikaci a validaci systému provedla nezávislá třetí strana; v takovém případě mluvíme o IV&V (*Independent Verification and Validation* čili *nezávislé verifikaci a validaci*). Ta je definována nezávislostí v následujících oblastech:

- Technická nezávislost vyžaduje, aby se V&V aktivit neúčastnil nikdo zapojený do vývoje ověřovaného systému.
- Finanční nezávislost znamená, že řízení financování V&V aktivit je svěřeno nezávislé straně. To zaručuje, že nedojde k rozpočtovým změnám a nežádoucímu tlaku, které by mohly negativně ovlivnit řádné dokončení všech V&V aktivit.
- Řízení V&V aktivit nezávislou stranou s objektivním, průběžným a neomezeným poskytováním svých zjištění vedení i vývojovému týmu.

Další podrobnosti je možné nalézt ve standardu IEEE 1012:2012.

Statické a dynamické techniky

Kvalitu výstupů jednotlivých fází vývoje softwaru lze v zásadě hodnotit za použití dvou odlišných kategorií aktivit, které využívají buď techniky *statické analýzy*, nebo *dynamické* techniky – *dynamickou analýzu* a (dynamické) testování.

Upozornění: V moderním pojetí jsou techniky statické analýzy běžně označovány souhrnným pojmem „statické testování", přestože někteří autoři tento pojem nepovažují za zcela korektní. My jej v této knize nepoužíváme a z praktických důvodů budeme mezi jednotlivými technikami rozlišovat.

Jak je z názvu patrné, statická analýza má za cíl zkoumat produkt (respektive dílčí produkty, jako jsou např. návrhové dokumenty, databázové modely nebo zdrojový kód) manuálně či pomocí různých nástrojů, ovšem aniž by došlo k jeho vlastnímu spuštění.

Proto může být statická analýza prováděna i ve velmi raných fázích vývoje softwaru a je také, jak bylo uvedeno výše, typickou verifikační aktivitou. Vedle samostatné aktivity testera zkoumajícího jednotlivé produkty a jejich specifikace můžeme mezi běžné techniky statické analýzy zařadit:

- Nástroje *automatické statické analýzy* mohou být samostatnými produkty, ale dnes jsou v určité podobě již standardní součástí integrovaných vývojových prostředí (IDE). Ve většině případů je automatická statická analýza zaměřena na zdrojový kód a podle kvality daného nástroje umožňuje např. kontrolu dodržování definovaných konvencí a standardů, kontrolu toku řízení a toku dat grafické znázornění.

- *Neformální revize* je rychlý a nejméně nákladný proces, kdy autor prochází a vysvětluje obsah dokumentu jednomu či více kolegům, kteří jej komentují a hledají možné defekty či problematická místa. Většinou se o revizi ani nalezených defektech nevede žádný záznam a okamžitě se opravují. Přes svou jednoduchost jde o velmi efektivní způsob hledání defektů.

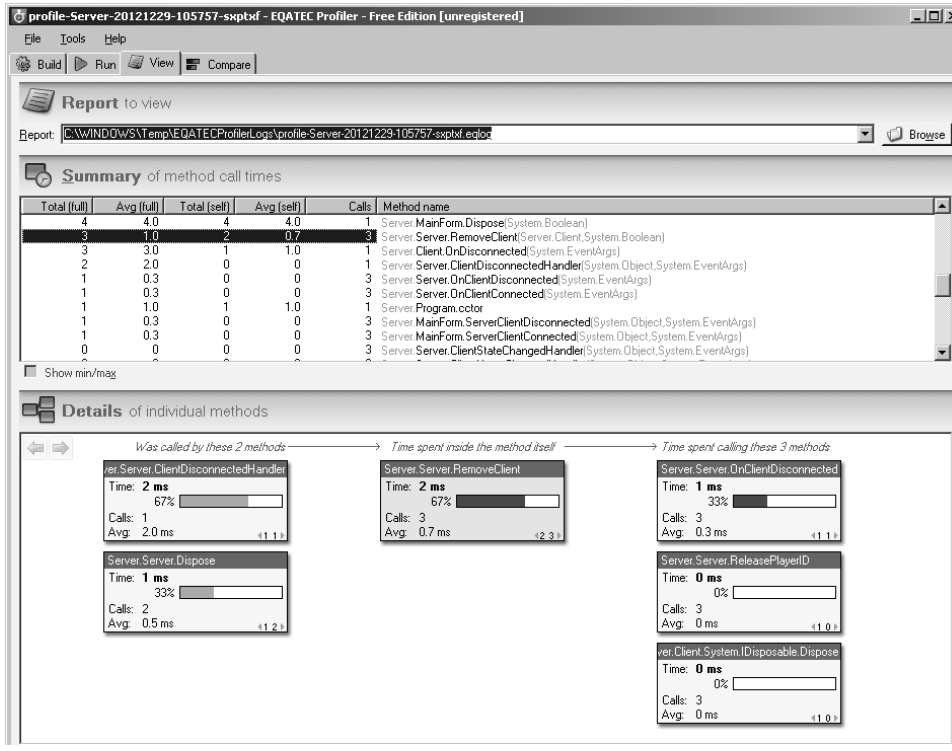
Poznámka: Neformální revizi provádí každý programátor, aniž o tom možná ví – krátká konzultace s kolegou je typickým příkladem.

- *Strukturované procházení* dokumentu či kódu bývá o něco více formální (ale stále do určité míry flexibilní) skupinovou aktivitou, kterou většinou řídí sám autor procházeného dokumentu za přítomnosti dalších osob s dostatečnou znalostí dané problematiky, jejichž role však nejsou přísně vymezeny. Účelem není pouze odhalit defekty, odchylky od standardů či zvážit možné jiné způsoby implementace, ale také sdílet znalosti o produktu v rámci týmu.
- *Technickou revizi* již lze označit za formální revizi. Jejím cílem je především zhodnotit, zda daný produkt splňuje požadavky pro zamýšlené použití, a identifikovat případné odchylky od specifikací a standardů či používaných konvencí. Výstupem není pouze seznam nalezených problémů, ale také doporučení, zda by měl být daný produkt přepracován, zamítnut nebo akceptován. Subjektem technické revize může být například specifikace požadavků či testovací dokumentace.

Poznámka: Především u technické revize se lze často setkat s odlišnými výklady. Výše uvedený je v souladu s normou IEEE 1028-2008.

- *Inspekce* je velmi formální a vychází z procesu, který vyvinul Michael Fagan v roce 1976 – odtud užívané označení *Faganovská inspekce*. Jde o plánovanou aktivitu s předem určenými rolemi jednotlivých členů (moderátor, průvodce, autor, oponenti, zapisovatel). Průvodce prezentuje kód (či například specifikaci požadavků) s případnou pomocí autora, oponenti vznáší připomínky a hledají defekty či problémy, které zaznamenává zapisovatel. Moderátor usměrňuje a vede diskuzi. Tento postup je velmi efektivní, avšak náročná na čas a zkušenosti zúčastněných, proto se často využívá jen v nutných případech.

Oproti tomu dynamická analýza zkoumá chování a vlastnosti produktu – systému či jeho komponentu – za běhu pomocí speciálních k tomu určených nástrojů, a navzdory častému omylu tak nejde o testování v pravém slova smyslu. Tato technika umožňuje například detailně analyzovat správu paměti a výkon systému a zjistit tak problémy, které by jinak byly velmi obtížně odhalitelné. Typickým příkladem využití dynamické analýzy je profilování aplikací v rámci zajišťování jejich optimalizace.



Obrázek 2.3 Profilování aplikace (herní server v jazyce C#) – zobrazení volaných metod a jejich trvání (nástroj EQATEC)

A konečně nejtypičtější dynamickou technikou užívanou ke zjištění kvality produktu je samotné testování v jeho nejrůznějších podobách. To je autory, kteří používají pro techniky statické analýzy souhrnné označení „statické testování", někdy explicitně označováno jako testování dynamické.

Z výše uvedeného by mělo být patrné, že testování v dnešním pojetí se zdaleka netýká pouze zkoumání spustitelného produktu a využívá navíc i jiné, úzce související a komplementární aktivity.

KAPITOLA 3

Defekt softwaru a související pojmy

V této kapitole:
- Chyba, defekt, selhání a pár slov o dependabilitě
- Příklady známých selhání
- Co je to „bug"?
- Závažnost a priorita defektu
- Analýza defektů

Software jako produkt je výsledkem spolupráce různých profesionálů – analytiků, systémových architektů a samozřejmě programátorů, kteří jako všichni lidé mohou a budou ve své práci chybovat. Následkem jejich pochybení se do kódu či ostatních artefaktů procesu softwarového vývoje zavlékají defekty způsobující chyby a selhání, o kterých si blíže povíme v této kapitole.

Chyba, defekt, selhání a pár slov o dependabilitě

Jednou ze základních vlastností charakterizujících výpočetní systémy je tzv. dependabilita neboli schopnost výpočetního systému dodávat službu, které lze oprávněně důvěřovat. Jinak ji lze také definovat jako schopnost systému vyhnout se selháním služby, která jsou častější a závažnější, než je pro uživatele přijatelné. [Avižienis; 2004]

Přestože s požadavky na vysokou míru dependability se dnes setkáváme především u kritických systémů, naše stále rostoucí závislost na výpočetních systémech v nejrůznějších oblastech života zvyšuje i potřebu jejich spolehlivosti, bezpečnosti, dostupnosti a udržovatelnosti. To vše lze zahrnout právě pod pojem dependabilita. Z tohoto důvodu se nám zdálo vhodné nastínit tuto problematiku v souvislosti s chápáním některých základních pojmů a principů.

Koncept dependability v podstatě sestává ze tří částí, kterými jsou: prostředky k jejímu dosažení, atributy a hrozby, na které se – s jistou mírou zjednodušení – nyní podíváme blíže.

Jako hrozby dependability se označují:

- *Závady* (*faults*)
- *Chyby* (*errors*)
- *Selhání* (*failures*)

Závada softwarového systému je statickým *defektem*, a tak budeme tento termín v tomto smyslu používat i my. Nutno podotknout, že pojmy defekt/závada, chyba a selhání jsou si velmi blízké, avšak v tomto pojetí nejsou vzájemně zaměnitelné, jak si mnozí nesprávně myslí.

Upozornění: Nepřesné používání těchto pojmů má často na svědomí i literatura, která pro zjednodušení používá ten či onen pojem. Ve skutečnosti jsou však uvedené pojmy defekt, chyba a selhání rozlišovány nejen v kontextu dependability, ale samozřejmě také v související oblasti testování (přestože je to mnohými ignorováno či redukováno na defekt-selhání). O dělení ve výše uvedené podobě se lze přesvědčit například v těchto knihách (včetně používání *fault* a *defect* jako synonym): *Introduction to Software Testing (Ammann,Offutt;2008), Software Testing and Quality Assurance: Theory and Practice (Naik,Tripathi; 2011), Design for Reliability: Information and Computer-Based Systems (Bauer, 2010), Testing Techniques in Software Engineering (Borba et al;2007).*

Jaký je ale rozdíl mezi chybou a defektem, v jakém vzájemném vztahu jsou a jak souvisí se selháním, to se dozvíme v následujících odstavcích.

Nejprve si definujeme několik pojmů, se kterými budeme pracovat. Začneme *systémem*, který budeme chápat jako entitu sestávající z jednotlivých vzájemně propojených komponent, jež jsou samy systémy, a komunikující s jinými entitami. *Funkce* tohoto systému – neboli k čemu je určen a co má vykonávat – popisuje funkční specifikace. *Chováním* systému pak rozumíme to, jak systém implementuje svou funkci, tedy jakým způsobem vykonává to, co se od něj očekává. Chování systému je reprezentováno řadou stavů.

Uživatel systému, ať je to jiný systém nebo člověk, vnímá chování systému jako *službu* poskytovanou na *rozhraní služby* (*service interface*), což lze chápat jako společnou hranici mezi systémem a ostatními systémy. Z celkového stavu systému ovšem na rozhraní služby vstupuje jen jeho část, která se označuje jako externí čili vnější. Služba je tedy řadou externích stavů systému.

Poznámka: Například systém může provádět složité výpočty a poskytovat výstup ve formě jednoho celého čísla, které bude jediným externím stavem pozorovatelným uživatelem, a tedy službou, kterou ten očekává. Zbylou část systému, jež nevstupuju na rozhraní služby, nazýváme interní.

Služba poskytovaná systémem je správná, pokud implementuje jeho funkci. Pokud se však služba odchýlí (respektive jeden či více externích stavů, ze kterých sestává) od stavu správného, mluvíme o selhání. V takovém případě služba není v souladu se specifikací nebo specifikace nepopsala správně funkce systému.

Nyní se dostáváme k samotným příčinám selhání – defektům a chybám. Zajímat nás však nebudou defekty hardwarové, které mohou pochopitelně vést také k chybám a selháním.

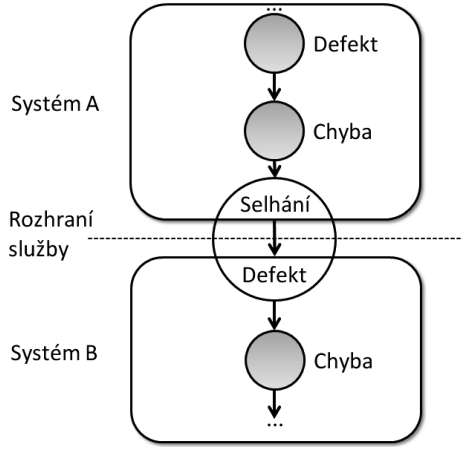
Defektem v softwarovém systému rozumíme vadu v kódu či datech, která je způsobena pochybením (nejčastěji) programátora různého původu – chybou v kódu, špatným návrhem, nedostatečně či nesprávně pochopenou specifikací, nebo dokonce záměrnou sabotáží. V rámci dependability lze defekty klasifikovat podle řady kritérií, to však již přesahuje rámec tohoto stručného úvodu.

Obsahuje-li program defektní kód či data, označujeme takový defekt jako neaktivní (dormant) do té doby, než dojde k jeho aktivaci provedením daného kódu či zpracováním vadných dat.

Připomeňme si, že selhání nastává tehdy, když se nejméně jeden z externích stavů systémů odchýlí od správného stavu služby, přičemž právě toto odchýlení, za jehož příčinu považujeme defekt, nazýváme *chybou*. Defekt většinou způsobí chybu nejprve ve stavu komponentu, který je součástí interního stavu systému, a jeho externí stav proto neovlivní okamžitě.

Chyba je tedy součástí celkového stavu systému a může následně vést k selhání. To nastane v případě, kdy se chyba stane součástí externího stavu daného systému (dosáhne rozhraní služby) a bude tak „viditelná" pro uživatele jako odchýlení od správné služby.

Jestliže dojde k selhání služby, pak pro její příjemce (uživatele) jde o externí defekt, šířící vstupem chybu do jejich systémů. Tam může opět dojít k selhání a následně šíření do dalších systémů. Toto lze demonstrovat řetězcem: …defekt (*aktivace)* → chyba (*šíření*) → selhání (*příčina*)→ defekt …



Obrázek 3.1 Šíření chyby. Selhání systému A je pro příjemce jeho služby (Systém B) externím defektem, který může vést k chybě a ta poté opět k selhání.

Ne všechny chyby povedou k selhání a šíření do dalších systémů, například díky použití různých korekčních mechanizmů umožňujících systému přechod z chybového stavu zpět do správného bez negativního efektu na poskytovanou službu. K selhání však nemusí dojít také proto, že daný (chybný) stav jen není potřebný pro poskytování služby.

Poznámka: Můžeme sem započítat i defekt v „mrtvém kódu" (kód, který je nadbytečný a měl by být odstraněn) – například funkci obsahující chybný výpočet, jejíž volání však v kódu chybí, a tak daný výpočet nebude nikdy proveden.

Shrneme-li výše uvedené, pak:

- *Defekt* je následek pochybení člověka a je příčinou chyby.
- *Chyba* je stav systému a může vést k selhání.
- *Selhání* je nesoulad mezi aktuálním a specifikovaným chováním systému.

Na tomto místě uveďme ještě jinou, často používanou definici: chyba je akce člověka vedoucí k nesprávnému výsledku, přičemž její manifestací je defekt softwaru. Ačkoli se to zdá zcela odlišné, jde jen o *jiný pohled na výše zmíněný řetězec* defekt→ chyba→ selhání…: chyba programátora vede k jeho selhání napsat korektní kód. To se projeví defektem v kódu, jehož aktivace vede k chybě a případnému selhání systému.

Podívejme se na velmi jednoduchý příklad: Níže uvedený kód je součástí metody komponentu a má za úkol vypočítat vzdálenost mezi dvěma body na horizontální ose X, přičemž výsledkem má být přirozeně číslo větší nebo rovno 0:

```
static int VzdalenostX(int X1, int X2)
{
  return X1 - X2;
}
```

Metoda přijímá dva celočíselné parametry, z nichž každý představuje X souřadnici jednoho z bodů. Pokud je například souřadnice prvního 200 a souřadnice druhého 150, pak funkce správně vypočítá, že vzdálenost mezi oběma souřadnicemi je 50.

V kódu je však defekt: jestliže bude hodnota prvního argumentu nižší než hodnota druhého (například X1=150 a X2=200), pak bude výsledek vrácený metodou záporný (-50). Programátor totiž u výpočtu zapomněl použít absolutní hodnotu. Defekt bude tedy neaktivní do té doby, než bude funkce zavolána s takovými argumenty, že vrátí záporné číslo a způsobí tak chybu.

Pokud by tato komponenta poskytovala službu, jejíž součástí by byl i výsledek uvedeného výpočtu, pak by chyba jako součást externího stavu komponenty dosáhla rozhraní služby a došlo by k selhání. Příjemcům služby, např. dalším komponentám, by se její selhání jevilo jako externí defekt, kterým by došlo k dalšímu šíření chyby. Stačí si představit, že by se příjemce očekávající kladné číslo kupříkladu snažil z výsledku vypočítat druhou odmocninu.

Jestliže se chyba komponenty stane součástí jejího externího stavu, dochází k selhání služby komponenty, ovšem pro samotný systém jako celek zůstává chyba interní. Selhání jedné z komponenty je pro systém defektem – k selhání služby systému tak nedojde do té doby, dokud se externí stav komponenty v chybovém stavu nestane součástí externího stavu systému a chyba tak nedosáhne rozhraní služby systému, kde způsobí její odchýlení od služby správné.

Pro lepší pochopení si představme tento příklad: Systém pro správu zákaznických požadavků sestává z komponent A a B. Komponenta A sbírá data ze všech poboček společnosti, ukládá je a zpracovává. Tato data jsou poskytována komponentě B, která v závislosti na jejich obsahu umožňuje provádět různé akce – například pro požadavky hodnocené jako „Závažné" a zaslané po 15:00 zobrazí operátorovi tlačítko „Ihned přiřadit pracovníka" a před 15:00 naopak tlačítko „Zařadit do fronty", u ostatních typů závažnosti čas nehraje roli. V komponentě A je však defekt – u požadavků se totiž ukládá pouze datum (čas je pak automaticky 0:00). Pokud si komponenta B vyžádá záznam hodnocený jako „Závažný" a zaslaný v 10:00, dostane se sice defekt do systému B, avšak nezpůsobí selhání, protože náhodou čas 0:00 (stejně jako 10:00) vyhovuje podmínce – chyba je maskována.

Problém nastane ve chvíli, kdy je otevírán požadavek zaslaný v 16:30. Komponenta A selže a nedodává komponentě B čas. Nulový čas je pro komponentu B defektem, avšak k jeho aktivaci dojde jen tehdy, pokud je typ defektu „Závažný". V takovém případě se komponenta dostává do chybového stavu, neboť porovnává čas 0:00 namísto 16:30. Ve výsledku selhává, neboť zobrazuje nesprávné tlačítko.

Je třeba si ale uvědomit, že chyba může dosáhnout rozhraní služby komponenty, a přesto nemusí způsobit selhání systému jako celku – vše závisí na funkci, kterou má tento systém vykonávat. Představme si například komponenty A a B, jejichž externí stavy představuje jedno celé číslo. U komponenty A považujeme za selhání, pokud je toto číslo rovno nule. Obě komponenty jsou pak součástí systému, jehož funkce vyžaduje pouze rovnost těchto čísel. Jestliže by komponenta B vrátila nulu a komponenta A by selhala (vrátila tedy také nulu), pak jsou obě čísla rovna, a k selhání tak nedojde (přestože chybný externí stav komponenty je součástí externího stavu systému).

O selhání dependability mluvíme tehdy, jestliže systém vykazuje vyšší míru a závažnost selhání, než je pro uživatele přijatelné.

Příklady známých selhání

V závislosti na účelu systému mohou mít selhání způsobená softwarovými defekty různé následky, včetně těch nejzávažnějších, tedy škodách na lidském zdraví a životech. Známým se stal především tragický případ jednotky Therac-25, zařízení určeného k ozařování pacientů s rakovinou. Mezi lety 1985 a 1987 vinou několika defektů v řídicím systému (a přílišné důvěře v samotný software, neboť oproti jeho předchozí verzi byla odstraněna hardwarová pojistka) zemřelo či bylo vážně zraněno šest lidí následkem mnohonásobně vyšších dávek záření.

Jindy může být defekt „jen" velice nákladný, jako v případě nosné rakety Ariane 5 v roce 1996. Ta musela být zničena necelou minutu po startu, neboť se odchýlila od svého kurzu. Příčina se ukázala být v letovém počítači, jehož systém se osvědčil v raketě Ariane 4. Horizontální rychlost, měřená jako 64bitové číslo s plovoucí čárkou, byla konvertována do celočíselné 16bitové hodnoty, jejíž maximální hodnota je 32 767 a pro raketu Ariane 4 byla dostačující. Modernější Ariane 5 byla při odlišné trajektorii rychlejší a během konverze došlo bohužel k situaci, kdy již 16bitové číslo nestačilo. Tato klasická chyba přetečení (*overflow*) poté způsobila selhání dalších systémů, které vyústilo v již uvedený konec.

Abychom se však nepohybovali jen v zahraničí, připomeňme situaci z druhé poloviny roku 2012, kdy byl v České republice spuštěn nový Centrální registr vozidel (CRV). Kolabující systém, se kterým úředníci mohli jen stěží (pokud vůbec) pracovat, měl tak na svědomí i několik případů, kdy rozhněvaný dav lidí čekajících na odbavení musela uklidňovat policie. Navíc závažnější než samotný problém s rychlostí byl fakt, že v registru se nacházela a stále zřejmě nachází (v době vydání této knihy) chybná data. Následky této situace Ministerstvo dopravy nakonec muselo kompenzovat. Na základě auditu bylo později identifikováno několik příčin, přičemž u jedné z nich bylo uvedeno následující: *V oblasti přechodu na CRV bylo nejdůležitějším krokem komplexní testování vytvářeného systému a proškolení uživatelů. Podle předložených dokumentů však nebyla především z časových důvodů testování a důkladnému proškolení uživatelů věnována dostatečná pozornost*. [Rezort dopravy přiznal příčiny dlouhotrvajících problémů registru vozidel – eGov.cz, 3. 4. 2013]

Co je to „bug"?

Znalost rozdílu mezi termíny defekt, chyba a selhání je jistě přínosná, ovšem v praxi se až na výjimky dané firemní kulturou mnohem častěji setkáváme spíše s neformálním a obecnějším označením, kterým je tzv. *bug*, jenž v angličtině znamená „brouk" či „otravný hmyz".

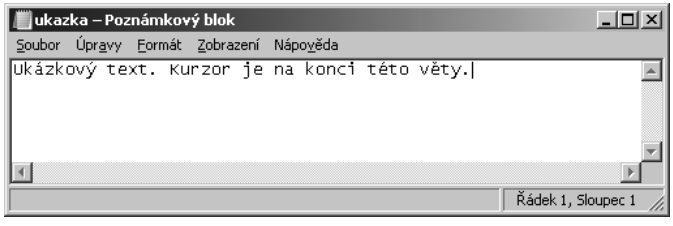
Již Thomas Alva Edison v textu z roku 1878 použil toto označení pro závady a potíže [E. J.Marinissen, 2008], se kterými se potýkal při práci na svých vynálezech. Jak poté termín „bug" pronikl také do softwarového průmyslu, je velmi dobře známo, neboť jde o vcelku

zábavnou historku: Ke konci čtyřicátých let tým pracující na elektromechanickém počítači MARK II na Harvardově univerzitě řešil podivný problém, který se nedařilo identifikovat, natož odstranit. Po pečlivém přezkoumání byla nakonec příčina nalezena – šlo o mrtvou můru zachycenou na kontaktech jednoho z relé. Po jejím vyproštění byl incident zaznamenán a označen jako první výskyt skutečného bugu (brouka).

Poznámka: Nakonec i v českém jazyce používáme výraz „mít své mouchy“, přestože to nemá nic společného se skutečným hmyzem.

Typicky se jako „bug“ označuje jakýkoli defekt softwaru (následek pochybení člověka), a proto budeme v dalším textu již nadále používat tento termín. Zde platí to, co jsme si o defektech, jejich příčinách a následcích řekli výše. Ať už se (aktivní) defekt a jím zapříčiněný chybový stav projeví jakkoli (od neošetřené výjimky zobrazené v uživatelském rozhraní přes nefunkční tlačítko až po náhodnou havárii systému), z pohledu uživatele se nakonec jedná o selhání. V obecnějším pojetí se tak „bug“ běžně používá i k označení svých následných projevů v systému – tedy chybových stavů a selhání v nejrůznějších podobách, kdy chování systému neodpovídá požadavkům. Pokud bychom však trvali na přesném používání výše uvedených termínů, pak při provozu systému nacházíme selhání, která indikují přítomnost defektu.

Pro zajímavost se podívejme na program Poznámkový blok ve Windows 7 (SP1): Po spuštění programu si zobrazíme **Stavový řádek**, otevřeme libovolný textový soubor, přesuneme kurzor někam do textu a uložíme. Pokud se nyní podíváme na stavový řádek udávající pozici kurzoru, uvidíme hodnoty **Řádek 1**, **Sloupec 1**, přestože kurzor se na této pozici nenachází. Jde tak očividně o selhání způsobené defektem, který aktivuje uložení otevřeného dokumentu. Po opětovném přesunutí kurzoru již bude pozice zobrazena korektně.



Obrázek 3.2 Poznámkový blok Windows 7 – kurzor se nenachází na pozici, kterou udává stavový řádek

Defekt se však nemusí objevit pouze ve zdrojovém kódu, ale i v dokumentaci popisující vyvíjený systém, jako jsou specifikace, případy užití či diagramy. V takovém případě jde o pochybení osob zodpovědných za dané dokumenty. Bude-li systém implementován podle chybné dokumentace, pak za provozu způsobí chybějící či nesprávně popsaná funkčnost (v okamžiku jejího použití) chybový stav, kdy poskytovaná služba bude odlišná od služby očekávané. Uživatel bude tento stav vnímat jako selhání, neboť specifikace v takovém případě nepopsala správně funkce systému.

Z hlediska specifikovaných požadavků můžeme podle Pattona mluvit o výskytu defektu, jestliže platí alespoň jeden z následujících bodů:

- Systém vykonává něco, co by podle specifikace vykonávat neměl.
- Systém nevykonává to, co by podle specifikace vykonávat měl.
- Systém vykonává něco, co není specifikováno.
- Systém nevykonává něco, co specifikováno není, ale mělo být.

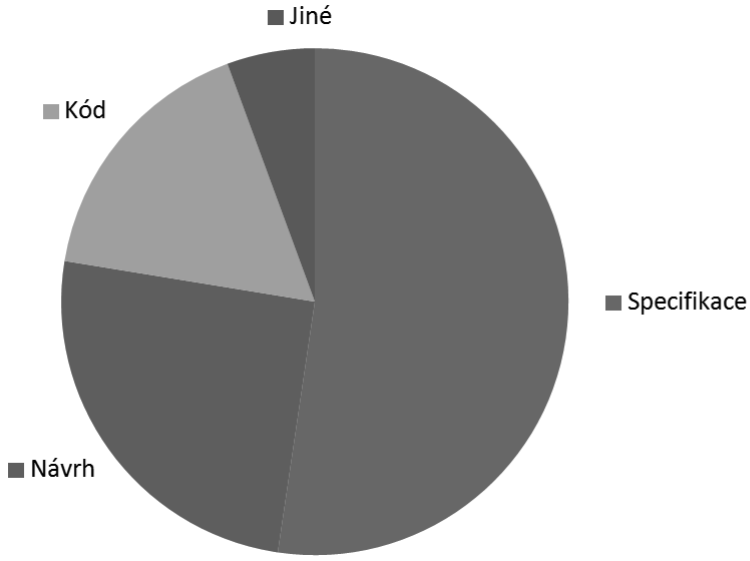
Uveďme si opět jednoduchý příklad: Systém by měl umožnit uživateli zvolit a odeslat na server obrázek ve formátu JPG či PNG, jehož velikost nepřesahuje 2 Mb. Během testování však zjistíme, že můžeme odeslat soubor s velikostí přesahující 2 Mb (první bod). Navíc zjistíme, že soubor ve formátu PNG odeslat nelze (druhý bod) a že při volbě souboru je jich možné označit více zároveň (třetí bod). Když poté zkusíme odpojit klienta od sítě aplikace zhavaruje s chybovým hlášením (čtvrtý bod) – chování pro případ ztráty spojení nebylo specifikováno, ale mělo být, protože předpoklad trvalého připojení k síti není správný.

Defekt se může také vyskytnout v oblasti mimofunkčních požadavků, například pokud není splněn požadavek na rychlost systému, robustnost, bezpečnost či přívětivé a snadno použitelné uživatelské rozhraní (*User Interface - UI*). Právě poslední zmíněné je však poněkud problematickým bodem, protože jde o subjektivní vnímání ovlivněné preferencemi, zkušenostmi i schopnostmi uživatele. Pokud nejsou požadavky na zpracování UI blíže specifikovány, je třeba řídit se vlastním úsudkem s přiměřenou mírou kritiky a celkovým dojmem, jakým na nás práce s daným softwarem působí. V úvahu musíme také brát, kdo a k čemu bude systém využívat.

Poznámka: Požadavky na vzhled UI mohou být i konkrétní – setkali jsme se například s takovým, aby v aplikaci nebyla vyskakovací okna a minimalizovalo se tak „proklikávání" zdržující od práce. UI tak tvořilo velké množství ovládacích prvků uspořádaných do skupin. Ačkoli bylo toto provedení zdánlivě nepřehledné a těžko použitelné, splnilo požadavky.

V širším smyslu lze tak o výskytu defektu mluvit v případě, kdy se software neshoduje se specifikovanými požadavky anebo oprávněnými očekáváními ze strany zadavatele, respektive zákazníků.

Z hlediska původu defektů se lze často setkat s tvrzením, že nejčastěji mají defekty svůj původ ve specifikaci nebo návrhu, což je zdůvodňováno neúplností či absencí psané specifikace, jejími změnami, nedostatečně promyšleným návrhem a podobně.



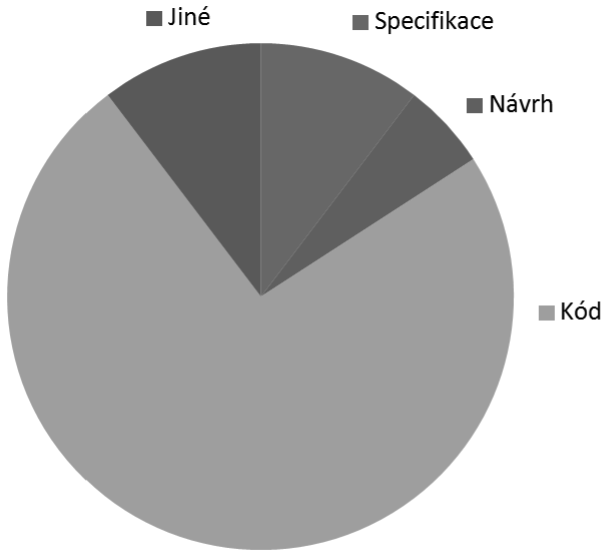
Obrázek 3.3 Rozložení příčin defektů, zdroj z roku 2001 [Patton, 2002]

Je však třeba poukázat na fakt, že tento (viz obrázek 3.3) vcelku často citovaný zdroj vychází z analýzy projektů do roku 2001. Aktuálnější materiály (viz obrázek 3.4) však uvádí čísla znatelně odlišná, kde jednoznačně převládají defekty s původem ve zdrojovém kódu – okolo 70–80%. To odpovídá i naší zkušenosti z řady projektů během posledních několika let. Lze se domnívat, že jde o výsledek používání vhodných metodik pro konkrétní projekty, dodržování norem i využití stále lepších nástrojů během analýzy a návrhu systému.

K obrázku 3.4 doplňujeme, že v původních grafech autoři navíc rozlišovali defekty grafického uživatelského prostředí (špatné rozložení ovládacích prvků apod.) a defekty dokumentace, které jsme sloučili do kategorie *Ostatní*.

Jednu z prvních detailních taxonomií defektů představil ve své knize Boris Beizer v roce 1990. Ten defekty rozděluje do tří hlavních skupin podle fáze, ve které vznikají, a poté až do čtyř podskupin, respektive úrovní. Pro zajímavost se podívejme na první dvě úrovně v jednotlivých fázích:

Poznámka: Níže uvedený výčet je přeložen volněji pro větší srozumitelnost. Nutno také podotknout, že taxonomií existuje řada a není výjimkou ani jejich vytváření v kontextu konkrétního projektu.



Obrázek 3.4 Rozložení příčin defektů, data zdroje z roku 2010 [Sakthi, 2010]

- Fáze návrhu
  - Požadavky
    - Nesprávnost požadavků
    - Nelogické, nedosažitelné apod
    - Neúplnost
    - Neověřitelnost
    - Defekty v dokumentaci nebo prezentaci požadavků
    - Změny požadavků
  - Vlastnosti a funkcionalita
    - Nesprávnost
    - Neúplnost funkcionality
    - Neúplnost případů v rámci funkcionality
    - Nesprávné zpracování určité kombinace vstupů
    - Chybné uživatelské zprávy nebo diagnostika (například nesprávné varování)
    - Neošetřené výjimečné stavy