TESTOVÁNÍ SOFTWARU A ZAJIŠTĚNÍ KVALITY

Softwarové systémy jsou rozšiřující se součástí života, od obchodních aplikací (např. bankovnictví) až po spotřebitelské produkty (např. automobily). Bohužel, ne vždy tyto systémy pracují správně a většina lidí má tak zkušenosti se softwarem, který nepracoval,jak se očekávalo. Software, který nepracuje správně, může způsobit mnoho problémů, včetně ztráty peněz, času nebo obchodní reputace, může dokonce způsobit zranění nebo smrt.

Důsledné testování softwaru a dokumentace mohou pomoci snížit riziko těchto problémů. Tím přispívají ke kvalitě softwarového systému (jestliže jsou zjištěné defekty opraveny dřív, než je systém uvolněn do provozu).

Pomocí testování je možné měřit kvalitu softwaru ve smyslu zjištěných defektů, pro funkcionální a také pro nefunkcionální softwarové požadavky a charakteristiky (např. spolehlivost, použitelnost, účinnost, udržovatelnost a přenositelnost).

Samozřejmě, že testování SW má i svá pravidla a ověřené postupy. Pro pochopení celé problematiky jsou v dalších kapitolách definovány základní pojmy .

Definice chyby

Případová studie – Problém roku 2000 (Y2K)

V sedmdesátých letech pracoval pro svoji firmu jistý počítačový programátor. Jeho úkolem bylo vyvinout mzdový systém, který by usnadnil práci administrativnímu oddělení. V této době však počítače disponovaly relativně malou operační pamětí, a proto bylo nutné psát co nejúspornější programy. Zmíněný programátor tento problém vyřešil tak, že údaje o datech zkrátil ze čtyř cifer (např. „1973“) na pouhé cifry dvě (tedy „73“). Díky této úpravě bylo ušetřeno velké množství místa v paměti. To vše v domnění, že za 25let bude program jistě nahrazen novým, rychlejším. V roce 1995 byl však program i nadále používán. Jeho autor mezitím odešel do důchodu a nik nevěděl, zda program zvládne přechod na rok 2000. Odhaduje se, že v rámci náhrady a aktualizace počítačových programů jako byl tento a v rámci potenciálního selhání z důvodu problému roku 2000 (Y2K) bylo nutné investovat několik stovek miliard dolarů po celém světě.

Případová studie – Webové stránky Ministerstva průmyslu a obchodu ČR

Jedna nejmenovaná společnost, která se zabývá mj. i zajišťováním kvality, realizovala bezpečnostní testování modernizovaných webových stránek Ministerstva průmyslu a obchodu ČR. Díky svým zkušenostem a vhodně zvoleným postupům byla schopna provést svou práci během několika dní. Výstupem tohoto testování bylo odhalení několika chyb, které tak mohly být odstraněny ještě před nasazením stránek do ostrého provozu. Odhaduje se, že při neopravení nalezených chyb by vzniklé škody mohly dosáhnout až několika milionové částky.

Tento případ naopak dokazuje, že kvalitní testování pomáhá předejít případným budoucím problémům (např. Útok hackera) a s tím spojené např. nemalé finanční náklady.

Co je to chyba?

Z předchozích kapitol je jasné, co všechno se může stát při selhání softwaru a jak může kvalitní testování tomuto selhání předcházet. Ve všech těchto případech je totiž zřejmé, že software nepracoval v souladu s původním záměrem. Většina selhání je docela malých a některé z nich dokonce tak bezvýznamné, že není úplně vždy jasné jestli se jedná o skutečné selhání – chybu. Důsledkem selhání softwaru může být tedy určité nepohodlí = chyba. Tímto nepohodlím může být cokoli od znechucení uživatele až po milionové ztráty či dokonce katastrofu, která může zapříčinit ztrátu na životech.

Co je cílem testera?

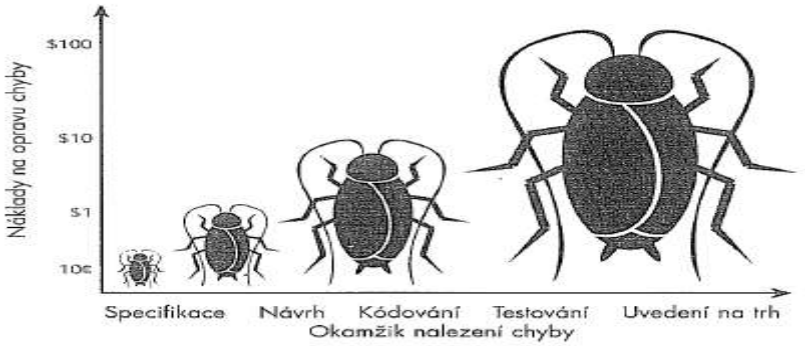
Cíl (nejen) softwarového testera je definován poměrně jednoduše: *„vyhledávat chyby, a to vyhledávat je co nejdříve a zajistit jejich nápravu“.* Jednotlivé části této definice budou rozebrány v následujících kapitolách.

Cíl softwarového testera: „...vyhledávat chyby...“

Pro softwarového testera je prioritou vyhledávat chyby. Pokud chybu nenajde, pak existují jen dva důvody. Buď je systém bezchybný (v praxi méně obvyklé až nemožné) nebo tester neprovedl svou práci dobře (bohuže nejčastější). Je proto nutné vytvářet testy, které se snaží chybu skutečně odhalit a tím snížit možné budouc prodražení celého projektu a s tím spojené zvýšení nákladů na odstranění chyby.

Cíl softwarového testera: „...vyhledávat chyby co nejdříve...“

Kvalitní testování je spojeno nejen se schopností vyhledávat chyby, ale i s rychlostí jejich vyhledání. S rostoucím časem totiž rostou i náklady na opravu chyby (viz Obrázek 1). Pokud například chybu objevíme již během vývoje, může oprava zabrat vývojáři 30 minut jeho času. Pokud však na chybu narazí až zákazník, budou náklady patrně vyšší, a to včetně možné ztráty zákazníkovy důvěry.



Cíl softwarového testera: „...zajistit opravu chyb...“

Nalezením chyby testerova práce nekončí. Chybu je totiž nutné oznámit např. vývojáři (ve formě zdokumentování chyby do reportovacího systému týmu) a jakmile tento vývojář chybu opraví, je nutné opět zkontrolovat, že vše bylo opraveno správně a oprava nezasáhla ostatní části systému.

Zajišťování kvality (QA)

Následující text se zabývá vysvětlením pojmu „QA“ a popisem některých systému řízení jakosti.

Co je QA?

QA neboli „Quality Assurance“ se dá do českého jazyka volně přeložit jako zajišťování kvality. Hlavním úkolem osoby odpovědné za zajišťování kvality softwaru je zkoumání a měření současného procesu vývoje softwaru a nalezení způsobů jeho zdokonalení, jejichž cílem je zabránit vzniku chyb. Skupina zajišťování kvality softwaru má mnohem širší záběr a odpovědnost než skupiny určené pro testování softwaru – anebo by podle své pracovní náplně aspoň měla mít. Kromě toho, že provádí testování softwaru nebo jeho část, má také za úkol předcházení vzniku chyb a zajišťovat určitou, samozřejmě vysokou, kvalitu a spolehlivost

softwaru. To znamená, že neprovádí jen testování a oznamování chyb – její povinnosti jdou mnohem dále – například zavádění a udržování různých systémů řízení kvality.

Příklady systémů řízení kvality (QMS) v softwarových společnostech

S častějším využíváním testování a sledování kvality jako součásti vývoje, vznikla potřeba ucelených pravidel pro tuto činnost. Proto vznikly různé typy systémů řízení kvality (Quality Management Systems, QMS), které umožňují v organizaci rozpoznávání, měření a zlepšování různorodých procesů tak, že vedou ke zlepšení výkonu společnosti. V následujících kapitolách budou uvedeny některé z takových systémů.

EN ISO 9000

ISO 9000 popisuje základní principy systémů managementu kvality a specifikuje terminologii systémů managementu kvality.

EN ISO 9001

ISO 9001 specifikuje požadavky na systém managementu kvality pro případ, že organizace musí prokázat svoji schopnost poskytovat produkty, které splňují požadavky zákazníka a aplikovatelné požadavky předpisů a že má v úmyslu zvýšit spokojenost zákazníků.

EN ISO 9003

Tato mezinárodní norma specifikuje, jaké požadavky na systém jakosti mají být použity v případech, kdy je třeba prokázat způsobilost dodavatele zjišťovat a řídit vypořádání každé neshody výrobku v průběhu výstupní kontroly a zkoušení. Tato norma je v současné době neplatná. [19]

EN ISO 9004

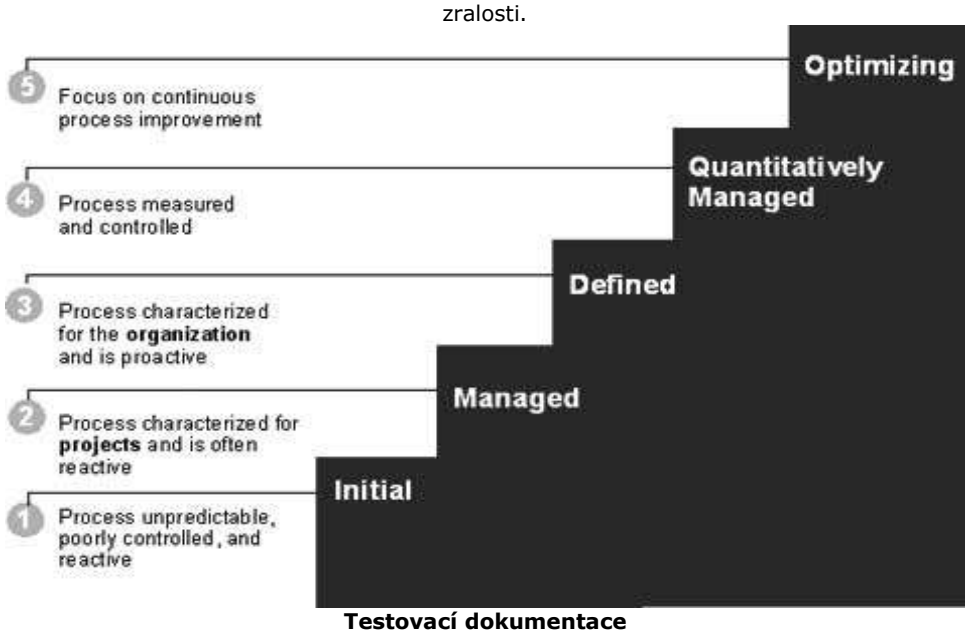
ISO 9004 poskytuje směrnice, které berou v úvahu jak efektivnost, tak účinnost systémů managementu kvality. Cílem této normy je zlepšování výkonnosti organizace, spokojenosti zákazníků a jiných zainteresovaných stran.

EN ISO 20000

Norma ISO/IEC 20000:2005 je první celosvětový standard, který se speciálně vztahuje k managementu služeb IT a zaměřuje se na zlepšování kvality, zvyšování efektivity a snížení nákladů u IT procesů. ISO 20000, které vzešlo ze standardu BS 15000, popisuje integrovanou sadu procesů řízení pro poskytování služeb IT [20]. Jedná se také o normu, která reaguje na potřebu zavedení českého ekvivalentu normy ITIL.

CMMI

Model zralosti a schopnosti (Capability Maturity Model Integration) se zabývá definicí, standardizací a postupným zlepšováním organizace, která se zabývá obecně vývojem, nejčastěji vývojem softwaru. Tento model je obecný a lze jej uplatnit v softwarové společnosti jakékoliv velikosti. Jeho pět úrovní (viz. Obrázek 2) představuje jednoduchý prostředek ohodnocení zralosti procesu vývoje softwaru v dané společnosti a určuje klíčové postupy, které je třeba přijmout pro přechod na další úroveň zralosti.



Testovací dokumentace

Existují dva hlavní dokumenty, které spravuje test analytik popřípadě tester:

- *System Test Specification (STS)* – obsahuje kompletní dokumentaci k testování (jednotlivá pravidla, definice, na čem se má testovat, jednotlivé testovací scénáře apod.). Zjednodušeně říká „co a jak testovat".
- *System Test Report (STR)* – slouží jako závěrečná zpráva po testování – (obsahuje tedy informace, co a jak bylo otestováno a především s jakými výsledky).

Navíc existuje ještě jeden dokument, ke kterému by měl mít test analytik i tester neustál přístup a na jehož tvorbě (testování) by se měl zčásti i podílet:

- *System Requirements Specification (SRS)* – skládá se ze specifikace požadavků na vyvíjený systém (tedy jak má konečná aplikace vypadat)

ZPŮSOBY TESTOVÁNÍ

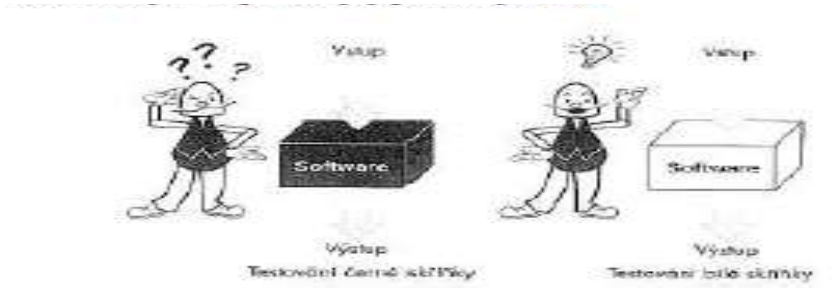
Aby mohl tester softwaru správně provádět svou práci, musí znát určitá pravidla. V následujících kapitolách jsou uvedena základní rozdělení přístupů k testování a s nimi spojené aktivity a nástroje.

Rozdělení přístupů k testování

Testy můžeme rozdělit do několika kategorií. Dále jsou rozebrány ty nejpoužívanější.

White box a Black box testování

Celkový postup při testování softwaru se charakterizuje jedním z následujících dvou pojmů: *testování černé skříňky* a *testování bílé skříňky*. Rozdíl mezi oběma pojetími ukazuje Obrázek 3. Při testování černé skříňky ví tester jen to, co má předložený software dělat – dovnitř skříňky se podívat nemůže a neví tedy, jak pracuje uvnitř. Jestliže napíše nějaký údaj na vstupu, dostane určitý odpovídající výsledek na výstupu. Neví, proč a jak se zrovna tento výsledek objevil, pouze jej pozoruje.



U testování bílé skříňky má oproti tomu softwarový tester přístup ke zdrojovému kódu programu a jeho zkoumání mu může pomoci při testování – vidí tedy jakoby „dovnitř" skříňky. Z tohoto pohledu pak tester může odhadnout, jestli určitá kombinace vstupů způsobí chybu a podle těchto informací může lépe přizpůsobit další testování.

Statické a dynamické testování

Při statickém testování se testuje něco, co neběží – to znamená, že zkoumaný objekt pouze prohlížíme a kontrolujeme (revidujeme). Statické testy (revize) jsou způsobem testování softwarových pracovních produkt (včetně kódu) a mohou se vykonat mnohem dříve než dynamické testy. Defekty nalezené během revizí v časných fázích životního cyklu jsou často odstranitelné mnohem levněji než ty, které jsou nalezeny až během vykonání testů (např. defekty nalezené v požadavcích). Dynamické testování je nejčastějším typem testování – software je spuštěn a tester s ním tedy pracuje v dynamickém stavu.

Funkční a nefunkční testování

Funkce, které systém, subsystém nebo komponenta má vykonávat, mohou být popsány v pracovních produktech, jako jsou specifikace požadavků, případy užití nebo funkční specifikace nebo mohou být nedokumentované. Funkce představují to, *co* systém dělá. Funkční testy jsou založeny na funkcích a vlastnostech (jak jsou popsány v dokumentech nebo chápány testery) a jejich spolupůsobení se specifickými systémy a mohou být
vykonávány na všech úrovních testování (např. testy komponent mohou být založeny na specifikaci komponent).
Nefunkční testování zahrnuje testování výkonu, zátěžové testování, stres testování, testování použitelnosti, testování udržovatelnosti, testování spolehlivosti a testování přenositelnosti, neomezuje se ale jen na ně. Je testováním toho, *jak* systém pracuje. Nefunkční testování může být vykonáváno na všech úrovních testování. Termín nefunkční testování popisuje testy vyžadované pro měření charakteristik systému a softwaru, které mohou být kvantifikovány vůči různým stupnicím měření, jako například doby odezvy pro testování výkonu.

Regresní a konfirmační testování

Poté, co je defekt nalezen a opraven, by měl být software retestován za účelem potvrzení, že původní defekt byl úspěšně odstraněn. Takovéto testování se nazývá konfirmační testování. Regresní testování je opakované testování již testovaného programu po modifikaci s cílem najít všechny defekty, které mohly být zaneseny nebo objeveny jako důsledek jiné změny (změn). Tyto defekty se mohou nacházet v testovaném softwaru nebo v jiné související nebo nesouvisející softwarové komponentě. Regresní testování se provádí, když je změněn software nebo jeho prostředí. Rozsah regresního testování je odvozen od rizika nenalezení defektů v softwaru, který předtím fungoval. Testy by měly být opakovatelné, pokud se mají používat v konfirmačním testovaní a pomáhat regresnímu testování. Regresní testování může být vykonáváno na všech úrovních testování a využívá se na funkcionální a nefunkcionální testy produktu. Sady regresních testů jsou spuštěny mnohokrát a všeobecně se vyvíjejí pomalu, proto je regresní testování silným kandidátem na automatizaci.

Automatické testy a testovací nástroje

postupem času byly a jsou vyvíjeny stále rozsáhlejší systémy a to má samozřejmě za následek i růst objemu testů. Při testování takových rozsáhlých systémů již není možno vystačit s obyčejným poznámkovým blokem nebo jednoduchou tabulkou - vznikají tak SW nástroje (např. Microsoft Test Manager či Testlink) a technologie (Selenium), které se testerovi snaží jeho práci jak ulehčit, tak zvýšit její efektivitu. Pro účely testování
mobilních aplikací pro Android je možné využít např. Robotium (http://code.google.com/p/robotium), jenž se velmi podobá technologii Selenium, ale je určeno pro Android.

Vlastnosti automatických testovacích nástrojů

Nejdůležitější vlastnosti a výhody testovacích nástrojů a automatizace jsou:

Rychlost

Zřejmě nejdůležitější vlastnost automatických nástrojů. Všechny úkoly jsou zpracovány strojově a bez prodlev. Je prokázáno, že při správně napsaných testech dokáže automatický nástroj urychlit testování až 15x.

Efektivita

Tato výhoda spočívá v tom, že testování přebere jakoby další tester (stroj). Fyzický tester se tak může věnova úpravě stávajících testů či testováním testovacích případů, které nemohly být zautomatizovány.

Správnost a přesnost

Člověk je mnohdy ovlivněn stresem či únavou, které se mohou negativně projevit na jeho pozornosti při testování. Výhodou stroje oproti člověku je jeho neúnavnost. Všechny úkoly vykonává tak, jak jsou nadefinovány a ať
jsou spuštěny kdykoliv, vždy je výsledek zpracován naprosto stejně a správně.

Výhody a nevýhody automatických testů

Výhody automatizovaného testování jsou zejména tyto:

+Eliminuje vyšší náklady spojené s dodatečnou údržbou testů
- Zvýšení produktivity (až 15x rychleší než manuální testování)
- Zvýšení kvality (větší pokrytí, vyloučení chyb lidského faktoru)

Naopak nevýhody lze definovat následovně:

-Komplikace při rozpoznání objektů (stroj není schopen „vidět" to, co člověk např. díky nemožnosti algoritmizace)
- Komplikace při synchronizaci (je stroj schopný vyčkat do vykreslení
obrazovky?)
- Vhodné pouze pro regresní testování
- Vyšší počáteční časové náklady pro psaní automatických skriptů

Srovnání softwarového testování a testování mobilních aplikací

Ačkoli je testování aplikací pro mobilní zařízení velmi podobné klasickému testování počítačového software, tak v žádném případě není totožné. Existuje tedy několik, občas dokonce zásadních, rozdílů. První z následujících tabulek uvádí rozdíly mezi počítačovým software a aplikacemi pro mobilní zařízení. Druhá tabulka pak uvádí rozdíly v samotných testech.

Tabulka 1 – Srovnání SW testování a testování aplikací pro mobilní zařízení - činnosti

Testy	Software pro PC	Mobilní aplikace reálné zařízení	Mobilní aplikace virtuální emulátor
Ovládací prvky	Klávesnice + Myš	Dotykový displej	Myš
Tvorba screenshots	Ano	Ne	Ano
Podporované multimediální formáty	Závislost na SW	Závislost na SW i OS	
Rychlost připojení k síti	Jen ve specifických případech, např. v kritických provozech	Proměnlivá dle síly signálu	Stabilní
Spotřeba při spuštění aplikace	Nesleduje se	Nutné sledovat	Nesleduje se
Ovládací gesta	Nepoužívají se	Možné testovat	
Místo instalace	Většinou pevný disk	Vnitřní paměť či paměťová karta	
Nároky na výkon HW	Výkon většinou přesahuje nároky aplikace	Nutné brát ohled na pomalejší procesory	Výkon většinou přesahuje nároky aplikace
Rozdíly v OS	OS dle verze	OS dle verze a výrobce zařízení	
Externí zařízení	Připojené k počítači	Součásti přístroje	

Tabulka 2 - Srovnání SW testování a testování aplikací pro mobilní zařízení – typy testů

Testy	Software pro PC	Mobilní aplikace reálné zařízení	Mobilní aplikace virtuální emulátor
Funkční testy	Ano	Ano	
Nefunkční testy	Ano	Ano	
„White-box" testy	Ano	Ne	
„Black-box" testy	Ano	Ano	
Statické testy	Ano	Ne	
Dynamické testy	Ano	Ano	
Regresní testy	Ano	Ano	
Performance testy	Ano	Ne	
„Monkey" testy	Ne	Ne	Ano
Manuální testy	Ano	Ano	
Automatické testy	Ano	Ne	

PROBLÉMY PŘI TESTOVÁNÍ

Následující text definuje a analyzuje nejčastější problémy a překážky, se kterými se tester může v praxi setkat.

Testovací data

Testovací data, převážně jejich vhodný výběr a kvalitní příprava, jsou jednou z nejdůležitějších částí tvorby testů. Vhodný výběr může přispět nejen ke zrychlení práce s testy (využití stejných dat pro kontrolu více funkcionalit testovaného programu), ale také zkvalitnit finální výsledky testů. Naopak jejich špatný výběr může způsobit řadu problémů, od neshody s produkčními daty až po narůstání objemu testů a tudíž i jejich následnou
problematickou údržbu. Testovací data samotná vyjadřují data, se kterými vyvíjená aplikace pracuje. Tato data se mohou vyskytovat na vstupu (nejčastější) nebo na výstupu.

Způsob získávání testovacích dat

V praxi se využívají převážně tři způsoby získávání dat.

Data od zákazníka

Zákazník nejlépe ví, kam přesně bude vyvíjený systém umístěn a jaká data bude zpracovávat – díky tomu má jasný přehled o tom, jaká data budou do systému vstupovat a jaké výsledky (výstupní data) budou ze systému vystupovat. Pokud je to tedy možné, je nejsnazší zákazníka požádat o reálný vzorek těchto dat a ty si případně dál upravit dle vytvořených testovacích případů (TCs - test cases). Takový přístup ušetří spoustu času, který bylo nutné vynaložit na manuální tvorbu dat a zároveň zajistí, že data se velmi podobají reálným datům, se kterými bude systém pracovat po jeho uvedení do provozu. Je tedy z uvedených možností nejvhodnější.

Generátor testovacích dat

Pokud není možnost získat data od zákazníka, je možné si vytvořit generátor testovacích dat, který vytváří umělá testovací data např. na základě specifikace. Data je nutné generovat tak, aby kvalitativně pokryla celé spektrum dat reálných (např. pomocí využití vzorkování, viz [28]). Tento způsob se může jevit pro testera jako ideální, není však úplně ideální pro projekt - na tvorbu

generátoru je totiž nutné alokovat nějakou pracovní sílu a spotřebovaný čas pak může chybět jinde, hrozí pak nedodržení termínů, vyšší náklady apod. Tento způsob je tedy vhodný pro dlouhodobější projekty či pro větší balíky testovacích dat.

Vlastní tvorba

Časově velmi náročný a chybově velmi náchylný způsob tvorby testovacích dat. Výhodu má snad jen v tom, že tester se při jejich tvorbě dokonalé seznámí s typem (formátem) těchto dat a tudíž s nimi dokáže v budoucnu rychleji a lépe pracovat.

Verifikace a validace testovacích dat

Při verifikaci se provádí kontrola, že postupujeme správně, tzn. zda používáme pro testování správná data. Toto je nutné provést ideálně ještě před započetím tvorby dalších testů či testováním samotným. Někdy bývá pojem verifikace nahrazen pojmem „kontrola správnosti". U validace se naopak snažíme potvrdit, že výstup testovacích dat odpovídá požadavkům, případně se jim maximálně přibližují a to v případě, že je tester testovací data nucen vytvářet uměle. Validaci lze provést matematickými postupy, typicky se také provádí konzultace se zákazníkem. Ideálním případem je tedy při zasílání náhledu testovacích scénářů poslat zákazníkovi i vzorek testovacích dat pro jejich validaci.

Typy a formáty

Testovací data můžou být nejrůznějších typů a formátů. Můžou existovat v některém ze známých formátu jako BMP či JPG (například pro obrázkové editory), TXT (textové prohlížeče), AVI či MP3 (audiovizuální nástroje) či XML (univerzální použití). Případně může být využit formát přesně na míru dané aplikace s kódováním například do hexadecimálního formátu.

Statická data

Tato data se nacházejí ve statické (časově neměnné) formě a jsou fyzicky přítomna v testovaném zařízení (na pevném disku počítače či paměťové kartě mobilního telefonu) nebo můžou být umístěny na vzdáleném serveru, odkud jsou čteny pomocí připojené sítě. Výhoda těchto dat je, že je lze připravit dopředu a využívat ve zvolený okamžik. Další výhodou je, že je lze dle libosti upravovat.

Dynamická data

Dynamická data jsou měněna dynamicky v čase (většinou produkována systémy třetích stran) a jsou tedy velmi těžko predikovatelná. To velmi ztěžuje práci testera, protože ten pak není schopen přesně určit předpokládaný výstup programu ve zvolený okamžik a tím není také někdy schopen chybu zreprodukovat. Tomuto se dá předcházet tvorbou tzv. klonů (generátorů dat). Tyto klony pak produkují totožná data jako systémy třetích stran, ale jsou plně konfigurovatelné a tester tak může ovlivňovat jejich výstupy. Takový způsob ale stejně jako při ruční tvorbě testovacích dat zabere určitý čas, a proto je vhodný pouze při déletrvajících projektech, kde je taková snaha odůvodněná návratem takové investice (ROI – return of investments).

Nemožnost pokrytí veškerých dat

Jeden z velkých problémů u tzv. mass systémů je problém nemožnosti pokrytí kompletních dat. Jedná se o systémy, které zpracovávají obrovské množství nejrůznějších dat (např. burzovní nebo telekomunikační systémy). Zde je nutné volit optimální vzorek dat tak, abychom například v již zmíněném burzovním systému „zasáhli" všechny obchodní skupiny, v případě jejich vysokého počtu pak ty nejpoužívanější. Je tedy důležité si zachovat jistou úroveň abstrakce, tedy udržet velikost testovacích dat na minimální možné úrovni při získání maximální efektivity. Proto je pozice test analytiků a lidí, kteří test data a testy samotné připravují, možná nejnáročnější vůbec.

Testovací prostředí

Stejně jako je důležité vhodně volit testovací data, je stejně důležité volit vhodné testovací prostředí a to v nejvyšší možné míře shodné s prostředím produkčním. S tímto je nutné počítat i v rámci finančních nákladů v rámci projektů (nákup dalšího HW či modelů mobilních zařízení, licence apod.). Důraz je nutné klást jak na kvalitu HW, tak i na jeho firmware (malá změna ve verzi může způsobit vyřazení instrukce z instrukčního setu a tím vyřadit z provozu celý systém). Na přesnou specifikaci cílového prostředí je nutno se zaměřit již v procesu definice požadavků anebo dokonce již při podepisování kontraktu.

Tester by měl mít pro svou práci opět k dispozici konfigurovatelné testovací prostředí. Díky konfigurovatelnosti je pak schopen simulovat například selhání či odpojení některého HW prvku či např. změnu firmware klíčového zařízení.

Způsob definování testovacího prostředí

Opět existuje několik způsobů, jak získat či vytvořit testovací prostředí.

Zařízení zapůjčené zákazníkem

Pokud zákazník dodá/zapůjčí vlastní zařízení, či k nim povolí přístup, pak má tester jistotu reálných zařízení s předem definovaným firmware a konfigurací. Určitá nevýhoda může být ta, že například oproti virtualizaci se hůře simulují defekty a také se zařízení obvykle pomaleji dostává do původního nastavení/stavu. Jedná se však o nejvhodnější řešení.

Zakoupení vlastních zařízení

Obvykle nejnákladnější možnost. Jednoduše spočívá v tom, že se pro projekt nakoupí zákazníkem předem definovaný hardware a software (včetně licencí) a ten se pak po celou dobu života projektu využívá dle potřeb týmu. Jedinou výhodou je to, že po skončení kontraktu se všechna tato zařízení dají využít v projektech dalších. Je zřejmé, že tento způsob je tedy vhodný pro projektové týmy, jež se specializují na určitý obor (například vývoj aplikací pro mobilní telefony běžící na OS Android).

Virtualizace

V rámci úspor je pak možné vytvořit či zakoupit virtuální přístroje, které umožňují plnou konfiguraci prostředí na programové úrovni a jsou tak schopny simulovat různé podmínky a chyby v prostředí. Jejich velkou výhodou je také to, že se dají velmi rychle uvést do původního, tzv. čistého stavu a tím tak urychlit přechod na další iteraci testů.

Nemožnost pokrytí veškerého HW

Tester ani nikdo jiný nikdy nebude schopen zákazníkovi zaručit, že vytvořený software poběží na jakémkoli stroji za jakýchkoli podmínek. Kdykoliv se totiž v budoucnu může stát, že nastane porucha starší komponenty, která už se nebude vyrábět, a proto bude nutné systém migrovat na úplně jiný HW, který se sice podobá původnímu, ale díky odlišnému výrobci může postrádat možnost některého nastavení. V případě volby testovacího prostředí je tedy nutné (stejně jako u testovacích dat), aby zvolené zařízení nijak výrazně nepřesahovalo náklady projektu a zároveň splňovalo všechny potřebné požadavky pro kvalitní otestování finálního produktu.

Požadavky od zákazníka

Požadavky od zákazníka (tzv. URD – user requirements definition) musí být zejména při vývoji na zakázku základní kámen celého projektu. Je velmi důležité v něm definovat tyto informace:

- **Přesné verze podporovaných prostředí (OS)** – v případě mobilních zařízení tedy například Android 2.3 či iOS 4.2.1
- **Typ použitých zařízení** – zde se jedná především o výrobce, model, verze firmware (jednotlivé modely se totiž mohou lišit jak v klávesách tak třeba

v rozlišení a velikosti displeje a dalším HW)

- **Jaké externí zdroje budou využity** – Internet, GPS, senzor gyroskopu a další připojitelné zařízení
- **Specifikace objemu přenášených dat** – umožní optimalizaci aplikace pro vysokorychlostní sítě WiFi (jaké pásma), či pomalejší 3G nebo GPRS
- **Cílová země a jazyk** – týká se dvou problémů: jedním jsou rozdíly v HW a SW zařízení pro různé trhy a druhým jsou podporované jazykové balíčky a tím rozdílné uživatelské prostředí (UI – user interface) vyvíjené aplikace
- **Jakými způsoby bude aplikace distribuována** – FTP, web, USB, Bluetooth či jiné
- **Přesná specifikace požadavků** – nejlépe podložená grafickými návrhy a v jazyce, kterému je schopen porozumět každý člen týmu.

Na co si zákazník často stěžuje

Následující text je zaměřen zejména na fázi získávání prvních požadavků od zákazníka a předprojektového plánování. Upozorňuje především na časté problémy, ke kterým je v této fázi potřeba velmi pozorně přihlížet.

- Mobilní telefony jsou silně závislé na signálu, a proto se internetové aplikace mohou často potýkat se **zamrzáváním či přerušením stahování**
- Problémy s **časovými limity u ověřování kreditních karet** se vyskytují u bankovních mobilních aplikací
- **Rychlost aplikace během slabého signálu** připojení k internetu či mobilní síti
- **Nekonzistentní tlačítka, fonty, nadměrná „barevnost"**
- **Chybějící nebo poškozené linky** – mobilní provedení funkce copy+paste je na rozdíl od PC obtížněji proveditelná
- **Aplikace není podporována zvoleným zařízením**
- **Zastaralé verze** – a tudíž staré chyby v aplikaci
- **Nemožnost blíže upřesnit nalezené problémy** – zákazník obvykle nemá možnost zasílat jakékoli screenshots či logy nalezených chyb v mobilním zařízení

Nevyslovené požadavky

Největším problémem při definici požadavků na produkt jsou tzv. zažitá pravidla, tedy něco, o čem se předpokládá, že to každý zná. Příkladem mohou být potvrzovací dialogová okna v MS Windows - v drtivé většině dokumentací projektů se dočtete, že po vykonání určité operace se má zobrazit potvrzovací okno s možnostmi „OK" a „Cancel", případně dokumentace obsahuje i funkcionalitu jednotlivých tlačítek. Absolutní většina uživatelů je zvyklá na to, že tlačítko „OK" je vždy vlevo, zatímco tlačítko „Cancel" vždy vpravo. Je to standard, který ale není nikde definován, tedy nevyslovený požadavek. Je nezbytně nutné, aby tyto standardy byly známé také vývojovému týmu. Proto je doporučeno dokumentaci doplňovat ideálně o grafické náhledy všech dostupných oken, případně uvádět i jejich rozměry či barevný odstín. Zároveň je ideální uvádět průměrnou požadovanou dobu reakce aplikace při specifických úkonech Vývojový tým se díky tomu pak vyvaruje budoucí dodatečné úpravě kódu a dokumentace. Nevyslovené požadavk bývají velkým problémem převážně u společností/zákazníků, kteří s vývojem nemají příliš zkušeností. Takové společnosti se typicky vyznačují tím, že jsou chaotické, nemají žádný systém řízení kvality, jsou příliš malé, špatně organizované či mají nejasné kompetence jednotlivých zaměstanců. Jejich požadavky pak mají tendenci se často měnit a mají nepřesnou formu.

Uchování testů

Všechny pracně vytvořené testy je samozřejmě nutné někde uchovávat. V dnešní době, kdy se vyvíjejí rozsáhlé informační systémy, již není možné uchovávat testy v papírové formě. Ke slovu pak přicházejí různé programy pro podporu ukládání testovací dokumentace. Tyto nástroje jsou většinou připojeny k databázi testů a umožňují tak přes uživatelské rozhraní jejich správu. Ty nejzákladnější nástroje (většinou také jako freeware) umožňují právě takové uchování a editaci. Ty robustnější umožní testy nejen spravovat, ale také spouštět, reportovat chyby či dokonce tvořit plně automatizované testy. Takových nástrojů je velké množství, v následujících kapitolách se proto zmíním o těch nejfrekventovanějších

TestLink

TestLink je jeden z webových nástrojů pro uchování, správu a spouštění testů. Jako jeden z mála je bezplatně dostupný a jeho vývoj je financován výhradně z dobrovolnýchpříspěvků uživatelů. Existuje ve formě webové aplikace, a tudíž nabízí i možnost tvorby a správy přístupových účtu jednotlivých testerů, jejich alokace n projektu a správu projektů samotných. Každý uživatel pak může dle svých práv v rámci přiřazeného projektu spravovat požadavky od zákazníka, vytvářet test cases a test suites a z nich následně tvořit kompletní test plány – samozřejmě včetně verzování. Pro připravené testy pak TestLink umožňuje vyplnění výsledků spuštěných testů, které ukládá a následně připraví nejrůznější metriky a test reporty.

Microsoft Test Manager

Test Manager (někdy též Test Professional) je distribuován jako doplněk k Microsoft Visual Studio. Nástroj spolupracuje jak s balíkem TFS (Team Foundation Server), tak s Visual Studiem. Kromě klasického managementu testů umí Test Manager i testy spouštět a to jak manuálně, tak dokonce i automatizovaně. Automatizace spočívá v tom, že při prvním spuštění manuálních testů nástroj nabídne možnost nahrání testerovy práce. Při každém dalším spuštění pak Test Manager „nakliká" většinu test kroků sám. Jednou z nevýhod tohoto nástroje (například oproti TestLinku) je jeho vysoká HW náročnost, a proto se nehodí pro firmy se starším HW. Další nevýhodou je velmi špatná dostupnost informací a nepřehledná nápověda. Uplatnění však nalezne ve větších společnostech, kde projektové týmy využívají různých SW balíků firmy Microsoft.

HP QuickTest Professional (QTP)

Tento nástroj poskytuje možnost automatizace funkčních a regresních testů pro nejrůznější aplikace a prostředí. Je součástí balíku HP Quality Center, který je určen pro zajišťování kvality ve společnostech. [29] QTP podporuje klíčová slova a tvorbu skriptů pro testování SW na uživatelské i programové úrovni. Ke specifikaci testovacího procesu a k manipulaci s jednotlivými objekty a kontrolními prvky v aplikaci využívá skriptovací edici Visual Basic (VBScript).

Borland SilkTest

SilkTest je dalším z nástrojů pro automatizaci funkčního a regresního testování. Je určen také pro business analytiky, test analytiky i vývojáře. Kromě testování klasického uživatelského rozhraní jeho prostředí podporuje tvorbu skriptů, které mohou být využity pro opakované testování aplikací ve většině z dostupných webových prohlížečů. Umožňuje také rychlé spouštění testů.

IBM Rational Robot

Rational Robot je automatizační tool pro funkční testování klient/server aplikací. Je určen kompletním QA týmům, kterým umožňuje detekování chyb, správu testovacích případů i celého testovacího procesu. Jeho výhodou je tak podpora více UI (user interface) technologií.

Ostatní nástroje

Z dalších nástrojů pak například:

- o TestComplete (SmartBear Software)
- o Parasoft SOAtest (Parasoft)
- o Ranorex (Ranorex GmbH)
- o Selenium (Open source)
- o WATIR (Open source)

Problémy při testování mobilních aplikací

V následujících několika kapitolách budou uvedeny kritické body a procesy, kterých j nutné si všímat při navrhování, vývoji a testování mobilních aplikací. Tyto údaje jsou získány převážně z praktických zkušeností.

Problémy při testování funkčních požadavků

Zde je uvedeno několik bodů, ke kterým je třeba přihlížet obzvláště během testování aplikací pro mobilní zařízení.

- **Ovládání klávesnicí, prstem či pointerem** – nutno vyzkoušet všechny možnosti
- **Gesta**1 – především u aplikací pro iOS a Android
- **Využití simulátorů** – simulátory jsou vhodné pro rané testování či při nedostupnosti fyzického zařízení. Zároveň jsou schopny provádět stress testy či monkey testy (viz. Níže.
  - o **Stress a „monkey" testy** – monkey test spočívá v chaotickém „klikání" do různých oblastí obrazovky – může odhalit skryté slabiny UI
  - o **Benchmarking** – aneb testování výkonu a měření množství paměti využívané aplikací (včetně doby načítání jednotlivých obrazovek)
- **Propojení mezi aplikacemi** – test komunikace a předávání dat mezi aplikacemi (nutno otestovat, co se stane, pokud druhá aplikace chybí, je poškozena či existuje v jiné verzi – např. rozmanitost multimediálních přehrávačů v mobilních zařízeních)
- **Změna rozlišení a orientace** – mnohá zařízení dnes během změny jejich polohy podporují dynamické otáčení displeje – je nutné zjistit, zda je obrazovka překreslena správně
- **Dynamické změny konfigurace** – změny v dostupnosti klávesnice, změna času či jazyka
- **Závislost na externích zdrojích** – pokud je aplikace závislá na přístupu do sítě, SMS, gyroskopu či datech z GPS, je nutné otestovat, zda se dokáže vypořádat s nedostupností těchto dat (tedy například pokud je jejich externí zdroj vypnut)
- **Závislost na internetovém připojení a jeho rychlosti** – tedy například změna připojení z rychlého na pomalé (WiFi na GPRS)
- **Podporované formáty** – pokud se aplikace zabývá např. multimediálními formáty, je třeba ověřit dostupné formáty v rámci platformy/OS
- **Crowdsourced aneb davové testování** – jestliže je aplikace určena pro široké masy, je vhodné zajistit patřičný vzorek testerů pro daný projekt (přínosné především pro testování UI)
- **Zjistit možnosti pro reportování chyb** – jakým způsobem budou tvořeny logy či screenshots z aplikace
- **Možnosti instalace/odinstalace aplikace a její ukončování/spouštění** – test, zda je aplikace vždy správně ukončena či smazána
- **Verze firmware** – testy pro všechny požadovaná zařízení a jejich konfigurace
- **Přerušení běhu aplikace** – například přesunutím na pozadí pomocí uživatelovy aktivity či během příchozího hovoru, SMS, slabá baterie, připojení/vypojení baterie
- **Využití skrytých údajů o telefonu** – může se hodit např. při performance testování (u OS Android je takové menu dostupné pomocí vytočení „*#*#4636#*#*")
- **Look and Feel** – závisí především na zkušenostech testera a jeho znalosti dané platformy – je nutné zajistit, aby aplikace fungovala (a dala se ovládat) pomocí uživatelsky zažitých tradic (tedy např. gestikulace prstů, velikost klikatelných tlačítek, čitelnost údajů, způsoby prohlížení galerií či vzhled menu)

Problémy při testování nefunkčních požadavků

V této kapitole je uvedeno několik praktických rad určených pro vývojáře (nejen však pro ně), které jim můžou být užitečné a především jim mohou pomoci předejít negativní kritice jejich práce.

- **Operační a úložná paměť** – mobilní zařízení mají limitovanou velikost operační a úložné paměti, při testování aplikace je tedy nutné sledovat i tuto veličinu
- **Výkon** – starší telefony mají pomalejší procesory, pokud se vyvíjí aplikace i pro ně, mělo by s tím být počítáno při optimalizaci (např. Android 3.0 podporuje více procesorů, starší verze nikoli), tester tedy musí sledovat, jak moc se aplikace promítá do aktuálního výkonu daného zařízení
- **Kompatibilita** – je nutné zjistit informace o všech zařízeních, pro které je aplikace určena (pro OS Android jsou tyto informace specifikovány v manifest-file aplikace)
- **Spotřeba** – potřeba testů pro měření spotřeby baterie
- **Podpora zařízení s rozdílným rozlišením či více displeji** – jedná se o nový trend na poli mobilních zařízení, se kterým je nezbytné počítat
- **Neinstalovat aplikaci na externí úložiště** – taková instalace může způsobit problémy při přepnutí do režimu „USB mass storage" během běžící aplikace
- **UnitTesty**2 – specializace pro každý OS (např. JUnit3 pro Android)

Srovnávací tabulka výskytů a závažností

Následující tabulka uvádí pro jednotlivé body z předcházejících kapitol jejich závažnost (tj. jak moc mohou ovlivnit úspěch projektu) a výskyt (tj. jak často se problém vyskytuje). Na konci kapitoly je pak uvedena tabulka s vysvětlením hodnot pro tyto pojmy. Poslední

sloupec tabulky pak uvádí součet hodnot přecházejících obou hodnot (čím vyšší je hodnota, tím větší riziko daný problém představuje).

Tabulka 3 – Závažnost a výskyt problémů při testování funkčních požadavků

Testování funkčních požadavků			
Problém	**Závažnost**	**Výskyt**	**SOUČIN**
„Look and Feel" problémy	3	3	**9**
Aplikace je pomalá	2	3	**6**
Pomalá komunikace po síti	2	3	**6**
Chyby v komunikaci s HW zařízení (např. GPS)	3	2	**6**
Chyby při instalaci/odinstalaci	3	2	**6**
Problémy s ovládáním (prsty, klávesnice, pointer)	3	1	**3**
Problémy při přerušení běhu aplikace	2	3	**5**
Špatné zobrazování na displeji s odlišnými parametry, špatné překreslení při otočení zařízení	2	2	**4**
Chyby v komunikaci mezi aplikacemi	2	2	**4**
Chyby při změně konfigurace zařízení	2	2	**4**
Nepodporované formáty	2	1	**2**
Špatné rozpoznání gest	1	1	**1**
Selhání v „monkey" testu	1	1	**1**

Tabulka 4 – Závažnost a výskyt problémů při testování nefunkčních požadavků

Testování nefunkčních požadavků			
Problém	**Závažnost**	**Výskyt**	**SOUČIN**
Nedostatek volné paměti RAM	3	3	**9**
Problémy s kompatibilitou (různí výrobci zařízení)	2	3	**6**
Vysoká spotřeba při běhu aplikace	2	2	**4**
Nedostatečný výkon procesoru zařízení	2	2	**4**
Nedostatek místa ve vnitřní paměti zařízení	2	1	**2**
Problémy s vykreslování na více displejů zároveň	2	1	**2**
Špatné umístění instalace (paměťová karta)	1	1	**1**

Následující tabulka uvádí vysvětlení obou použitých sloupců.

Tabulka 5 – Vysvětlení značení

Závažnost		
Hodnota	**Definice**	**Příklad dopadu**
3	VYSOKÁ	Pád aplikace, zamrznutí telefonu, neakceptace zákazníka
2	STŘEDNÍ	Problém s funkčností, který však lze obejít alternativním postupem
1	NÍZKÁ	Drobný problém, kosmetická chyba

Výskyt	
Hodnota	**Definice**
3	Velmi často
2	Často
1	Zřídka

OPERAČNÍ SYSTÉMY MOBILNÍCH TELEFONŮ

Android od firmy Google

Google Android je jedním z nejmladších operačních systémů pro mobilní zařízení („chytré" telefony, PDA, navigace). I přesto je však v současné době jedním z nejpoužívanějších OS pro mobilní zařízení a dá se říct, že jediným větším konkurentem je pro něj iOS (systém běžící na mobilních zařízeních od firmy Apple).

Historie a základní informace

Jak již jeho úplný název napovídá, jedná se o systém od firmy Google. Jde o OS na linuxovém jádře, má však kořeny sahající mimo Google – pochází totiž z dílny firmy Android Inc, kterou v roce 2005 převzal právě Google. Ten pak veškeré zdrojové kódy předal sdružení firem Open Handset Alliance (uskupení původně 34 výrobců mobilních telefonů, telekomunikačních operátorů a technologických firem, které prosazují používání otevřených standardů na poli mobilních zařízení). Za vznikem tohoto sdružení stojí též sám Google, který také financoval odměny pro autory prvních aplikací pro tento OS (např. v rámci soutěže Android Developer Challenge).

Vývoj aplikací se provádí v jazyce Java a za pomoci speciálních knihoven od Google (to vše v rámci vývojářského balíku Android SDK, které je zároveň plně podporováno vývojovým prostředím Eclipse včetně Android simulátoru). Android však Javu nativně nepodporuje. Oficiální ohlášení platformy proběhlo 5. listopadu 2007 a od počátku roku 2008 byla k dispozici první veřejně dostupná verze platformy a to jak pod licencí Apache, tak pod licencí GPLv2 (jedná se tedy o open-source software). Android 1.0 včetně vývojového

prostředí pak spatřil světlo světa 28. září 2008.

Distribuce - Android Market

Jedná se o aplikaci v každém telefonu s OS Android, která skrze internetové připojení zařízení slouží k distribuci aplikací pro tento operační systém. Uživatel si tak může skrze něj stahovat do mobilu nejrůznější aplikace, které jsou pro přehlednost rozděleny do patřičných kategorií. V současné době je na něm k dispozici přes 150 000 aplikací a to jak placených, tak bezplatných. Díky open-source licenci je však možné instalovat aplikace i pomocí zkopírování na paměťovou kartu a následné instalace pomocí některého za správců souborů telefonu.

Upgrade a downgrade systému

Upgrade je možný, pokud je zařízení kompatibilní s novým systémem. Záleží tedy pouze na výrobci, zda a kdy nový systém pro své zařízení zpřístupní. S downgradem je to bohužel horší, protože jsou potřeba pokročilé znalosti systému a v neposlední řadě záleží i na možnostech daného mobilního zařízení. Seznam podporovaných zařízení lze nalézt zde .

iOS od firmy Apple

Operační systém iOS pochází od firmy Apple, která jej vyvíjí primárně pro svá mobilní zařízení typu iPhone, iPad či iPod.

Historie a základní informace

Historie tohoto operačního systému se začala psát 9. ledna 2007, kdy byl vydán první iPhone. Tehdy ještě známý pod názvem iPhone OS. Název iOS se začal používat až od čtvrté verze tohoto operačního systému . iOS je odnoží klasického MacOS X a je tedy postaven na UNIXovém systému, ke kterému přidává multi-touch podporu (podpora
ovládání více prsty). Původně tento systém nepodporoval žádné z aplikací třetích stran, a proto vůbec první oznámení o vývoji SDK přišlo až ke konci roku 2007. Vývoj aplikací mimo Apple začal až 6. března 2008,což je datum vydání SDK (software development kit) pro iOS. Vývojáři si tak mohli vytvářet a ladit aplikace na iPhone simulátoru, který je součástí SDK. iOS však zůstal i pak poněkud uzavřeným systémem a je možné do něj instalovat jen „ověřené" aplikace. Pro získání této „ověřovací" licence je nutné se přidat do iPhone Developers Programu a samozřejmě zaplatit poplatek. Apple se tím údajně snaží bránit psaní neoptimalizovaných aplikací.

Vývoj probíhá v jazyce Objective-C ve vývojovém prostředí Xcode od Apple. Zároveň toto IDE běží pouze na počítačích Mac s procesory Intel. Jak již bylo zmíněno, přejmenování na iOS přišlo v červnu roku 2010. Tato zkratka zároveň pobouřila společnost Cisco Systems, jenž název IOS používala u SW, který byl instalován na jejích zařízeních.

Distribuce - App Store

Každému vývojáři z iPhone Developers Programu je umožněno zveřejnit svou aplikaci skrze App Store (podobný Android Marketu) a také si nastavit cenu, za stažení této aplikace. Z každé prodané licence pak získá 70% podíl (zbytek připadá společnosti Apple). V současné době je v App Store k dispozici něco přes 300 000 aplikací (což je zhruba dvakrát tolik než u Androidu). Na iOS nelze oficiálně instalovat jiné než ověřené aplikace. Neoficiální cesta samozřejmě existuje a to pomocí tzv. jailbreaku. Jedná se o instalaci speciálního exploitu, který daný přístroj odemkne a uživatel pak může instalovat veškeré aplikace. Tyto exploity jsou pak většinou zveřejňovány různými hackerskými skupinami většinou několik týdnů až měsíců po vydání nové verze iOS. Apple se však brání vydáváním nových systémů a zároveňneuznáváním reklamací takto odemknutých přístrojů.

Upgrade a downgrade systému

Upgrade systému se děje většinou automaticky skrze připojení k počítači s iTunes (multimediální program pro správu zařízení firmy Apple) a Internetem. Ten při zjištění nové verze iOS na ni sám upozorní, případně stáhne a nainstaluje do připojeného zařízení. Pokud jde o downgrade, tak nastává problém. Je nutné totiž mít IPSW (jedná se o jakýsi popis předchozích iOS). Problémem je, že je nutné mít tato data přesně pro konkrétní model (IPSW je tedy vázáno na sériové číslo telefonu). A i pokud tato záložní data jsou k dispozici, tak je downgrade ne zrovna snadnou záležitostí (některé společnosti pak mají pro účely vývoje a testování hned několik iPhonů s různými verzemi iOS a přísným zákazemjejich upgradu). Neoficiální seznam podporovaných zařízení (samozřejmě pouze od firmy Apple) je zveřejněn na stránkách Wikipedie

Ostatní

Kromě již zmíněných nejrozšířenějších operačních systému se samozřejmě používají i další, které přímo či nepřímo podporují někteří výrobci mobilních zařízení.

Windows Phone (Windows Mobile)

Windows Mobile byl jeden z prvních systému právě pro „chytré telefony", ale v současnosti (a ve verzi Windows Phone 7) však tento mobilní systém stojí spíše ve stínu ostatních. Používá se stále především v USA. nicméně slibnější budoucnost se mu nabízí ve formě smlouvy s firmou Nokia - pokud by totiž k této dohodě skutečně došlo, tak by tato finská společnost mohla tento operační systém začít prosazovat u většiny svých zařízení (což by znamenalo zřejmě také zánik Symbianu, viz dále).

BlackBerry

O tomto systému se v České republice příliš neví, důvodem je zřejmě malé rozšíření těchto smartphonů. Doménou těchto zařízení je především severoamerický trh, kde zaujímá přední místa v popularitě spolu se zařízeními pro Windows Mobile. Autorem tohoto systému je kanadská společnost RIM (Research in Motion). Mobilní zařízení BlackBerry si vybírají především uživatelé z oblasti managementu a to z důvodu velmi dobrého zabezpečení, kterým toto zařízení disponuje. Aby byl totiž BlackBerry využit na maximum, je nutný dodatečný SW na některém ze vzdálených serverů, který umožňuje propojení jednotlivých zařízení například v rámci společnosti (funguje jako intranet).

Symbian

Posledním jmenovaným OS pro mobilní zařízení je Symbian. Jeden z nejstarších a možná i nejznámějších operačních systémů, který však po obrovském boomu Google Android upadl téměř v zapomnění (ačkoli je, stejně jako Android, postaven na svobodné licenci). I nadále však probíhá jeho vývoj - v současné době je k dispozici Symbian OS 9.4 a to především na mobilních telefonech značky Nokia. Nevýhoda tohoto systému spočívá v časté nekompatibilitě se staršími verzemi nebo také v omezení převážně na procesory ARM.

PRAKTICKÁ ČÁST

TESTING SKELETON

Pojem Testing skeleton by se dal do češtiny volně přeložit jako „kostra testování". Jedná se tedy o jakýsi manuál či popis toho, jak by měl tester/test analytik postupovat při své práci (tedy zejména přípravě, spouštění a vyhodnocení testů) na projektu. Následující kapitoly by měly sloužit právě jako takový testing skeleton (v tomto případě má však svá specifika, která jsou zaměřena primárně na testování mobilních aplikací). Postupy a myšlenky z tohoto manuálu však může využít například i tester webových či desktopových aplikací. Následující podkapitoly slouží jako jednotlivé procesy testing skeletonu. Součástí každé podkapitoly/procesu bude vysvětlení kroků, důvod jejich zavedení a rizika, která s sebou přináší při nedodržení. V každé podkapitole je zároveň uveden výstup neboli výsledek po ukončení daného procesu. Náročnost je odhadována jako procento z celkové alokace
testera na daném projektu. Jednotlivé kroky procesu jsou pro lepší orientaci číslovány (sekundární kroky, tedy nepovinné, jsou na začátku označeny hvězdičkou).

Projektová příprava

Popis: Na počátku každého projektu by se měl tester seznámit s projektem. Mělo by jej zajímat pro koho projekt, použité technologie, složení týmu a také jazyk pro komunikaci s klientem.

Výstup: Tester by měl získat základní pojem o připravovaném projektu a především se rozhodnout (případně přispět k rozhodování), zda je pro něj vhodným kandidátem (dostatečné znalosti).

Náročnost: cca 1-3%

Tabulka 6 – Popis procesu Projektové přípravy

Primární cíle procesu			
Krok	**Popis činnosti**	**Důvody zavedení**	**Rizika při nedodržení**
1.1	Seznámit se s cílem projektu (obor, pojmy)	Získání povědomí o projektu	Možné nepochopení pojmů z projektu, špatně definované testy
1.2	Pochopit použité technologie	Nutnost alespoň pasivní znalosti použitých technologií	Nepochopení základních principů a možností použitých technologií
1.3	Zjistit jazyk pro komunikaci s klientem	Důležité pro správné pochopení vstupů od klienta	Obtížná, pomalá a nepřesná komunikace s klientem
Sekundární cíle procesu			
Krok	**Popis činnosti**	**Důvody zavedení**	
*1.1	Zjistit bližší informace o klientovi	Být proaktivní (snaha identifikovat klientovy požadavky, a to dříve než jsou skutečně vyřčeny)	
*1.2	Zvážit crowdsourced testování	Vhodné pokud je aplikace určena pro široké masy (především pro testování UI)	

Seznámení se s týmem

Popis: Cílem tohoto procesu je základní seznámení s týmem (tedy vývojáři, designery a vedoucím týmu nebo projektovým manažerem). To zahrnuje definici způsobu komunikace a specifikaci co vyžaduje tým od testera a naopak.

Výstup: V tomto případě by měl tester získat přehled o kompetencích jednotlivých členů týmu a sjednotit se s týmem jak v komunikaci, tak v nastavení jednotlivých projektových nástrojů.

Náročnost: cca 5-7%

Tabulka 7 – Popis procesu Seznámení se s týmem

Primární cíle procesu			
Krok	**Popis činnosti**	**Důvody zavedení**	**Rizika při nedodržení**
2.1	Definice kompetencí jednotlivých členů týmu	Dotazy jsou směřovány ke správné osobě	Zmatená komunikace napříč celým týmem – zdržení.
2.2	Zpřístupnění projektových adresářů	Přístup k projektové dokumentaci	Dodatečné vyřizování - zdržení
2.3	Začlenění do projektové e-mail skupiny (+ vytvoření pravidel pro užívání e-mailu, např. v MS Outlook)	Pro zrychlení a zpřehlednění komunikace v rámci projektu	Nekonzistentní informovanost týmu
2.4	Přístup do bugsystému (Mantis, Bugzilla, TFS)	Přístup k reportovacímu systému	Dodatečné vyřizování - zdržení
2.5	Centralizace projektových úkolů (office documents, TFS, XPlanner)	Pro zpřehlednění prací na projektu	Chaotické řízení a nepřehlednost v projektových úkolech
2.6	Instalace ostatních projektových prostředí	Přístup ke všem informacím a začlenění do projektu	Pozdější instalace špatných verzí, pomalá práce s neznámými nástroji
Sekundární cíle procesu			
Krok	**Popis činnosti**	**Důvody zavedení**	
*2.1	Žádost o informování o nové verzi a místu jejího uložení (pomocí email/skype)	Zjednodušení komunikace a urychlení práce	
*2.2	Žádost o umístění verzování přímo do aplikace (název, verze, datum)	Zpřehlednění aktuálně nasazených verzí	

*2.3	Žádost o zveřejňování úprav v každé vydané verzi (i interní)	Objektivnější psaní testů a testování
2.4	Při opravách bugů by měli vývojáři definovat, ve které verzi se oprava vyskytne	Retestování jen skutečně opravených bugů
2.5	Zavedení tvorby logu přímo v mobilním zařízení	Kvalitnější údaje při reportování chyby
2.6	Zavedení možnosti vytváření screenshots přímo v aplikaci	Kvalitnější údaje při reportování chyby

Analýza požadavků a SRS

Popis: V tomto procesu má tester za úkol si nastudovat specifikaci systému (SRS), analyzovat zákazníkovy požadavky a případně specifikovat nedostatky, které musejí být v SRS doplněny či opraveny.

Výstup: Tester získá kompletní znalosti o vyvíjeném systému a případně doplní či opraví patřičnou dokumentaci.

Náročnost: cca 5-7%

Tabulka 8 – Popis procesu Analýza požadavků a SRS

Primární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	Rizika při nedodržení
3.1	Nastudování URD a SRS	Získání všech potřebných informací o vyvíjeném systému	Tvorba špatných testů, nepochopení některých pojmů
3.2	Analýza URD a SRS (viz. Příloha III)	Případné doplnění všech potřebných údajů do SRS	Vážné odlišnosti mezi produktem a zákaznikovými požadavky
3.4	Založení STS (tvorba šablony dokumentu)	Slouží k dokumentaci testerovy práce	Chaotická nebo nulová testerská dokumentace
Sekundární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	
*3.1	Rezervace/zapůjčení cílových mobilních zařízení	Seznámení se s jejich funkcemi a prací s nimi ještě před začátkem vývoje	
*3.2	Brainstorming – nechybí něco?	Možné doplnění a následná lepší efektivita práce	

Definice testovacího prostředí

Popis: Tento proces se zabývá instalací a nastavením testovacího prostředí. Jedná se nejen o počítače, ale také o rezervaci a přípravu mobilních zařízení.

Výstup: Po tomto procesu by měl mít tester k dispozici plně funkční a nastavená testovací zařízení i počítač, což mu bude velmi užitečné jak při tvorbě a ladění testů tak při testování samotném.

Náročnost: cca 3-5%

Tabulka 9 – Popis procesu Definice testovacího prostředí

Primární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	Rizika při nedodržení
4.1	Identifikace potřebného HW a SW (včetně licencí) z SRS	Získání seznamu potřebných prostředků	Nedostatek financí na dodatečné získání opomenutých prostředků
4.2	Zajištění těchto prostředků – Odblokované telefony, SIM karty (přesné konfigurace)	Tester musí mít veškeré prostředky okamžitě k dispozici	Při nedostatku zařízení se může práce prodlužovat
4.3	Obstarání uživatelských manuálů ke všem zařízením	Pro snadnější řešení problémů s nastavením a ovládáním	Dodatečné vyhledávání může zabrat čas navíc
4.4	Instalace a nastavení SW (virtuální simulátor)	Snaha o co nejvěrnější napodobení cílového prostředí	Nemožnost sjíždět benchmark testy
4.5	Nastavení HW (dle SRS)	Snaha o co nejvěrnější napodobení cílového prostředí	Nedokonalé otestování
4.6	Zavedení okolního prostředí	Některé aplikace mohou být využívány potmě, při hluku či otřesech	Nedokonalé otestování
4.7	Doplnění údajů do STS	Slouží k dokumentaci testerovy práce	Chaotická nebo nulová testerská dokumentace
Sekundární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	
*4.1	Vyhodnocení nutnosti tzv. field testu	Testování v reálných podmínkách, které jsou laboratorně nedosažitelné	
*4.2	Tvorba/doplnění manuálů	Mohou pomoci nově příchozím testerům	

Definice testovacích dat

Popis: Je nutné před samotným testováním získat specifikaci testovacích dat. Tento proces se zabývá jejich specifikací, získáním (zákazník či vlastní generátor) a vybráním kvalitního vzorku pro testy.

Výstup: Tester získá výchozí vzorek testovacích dat, ze kterého pak bude vycházet při tvorbě testů i při testech samotných.

Náročnost: cca 3-5%

Tabulka 10 – Popis procesu Definice testovacích dat

4.6	Zavedení okolního prostředí	Některé aplikace mohou být využívany potmě, při hluku či otřesech	Nedokonalé otestování
4.7	Doplnění údajů do STS	Slouží k dokumentaci testerovy práce	Chaotická nebo nulová testerská dokumentace
Sekundární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	
*4.1	Vyhodnocení nutnosti tzv. field testu	Testování v reálných podmínkách, které jsou laboratorně nedosažitelné	
*4.2	Tvorba/doplnění manuálů	Mohou pomoci nově příchozím testerům	

Definice testovacích dat

Popis: Je nutné před samotným testováním získat specifikaci testovacích dat. Tento proces se zabývá jejich specifikací, získáním (zákazník či vlastní generátor) a vybráním kvalitního vzorku pro testy.

Výstup: Tester získá výchozí vzorek testovacích dat, ze kterého pak bude vycházet při tvorbě testů i při testech samotných.

Náročnost: cca 3-5%

Tabulka 10 – Popis procesu Definice testovacích dat

Primární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	Rizika při nedodržení
5.1	Určení typu dat – dynamická/statická data (SRS/zákazník)	V případě dynamických dat nutné konfigurovat servery a přístupy	Chybná simulace vstupních dat
5.2	Identifikace potřebného vzorku testovacích dat (SRS/zákazník)	Získání seznamu potřebných prostředků	Nedostatek financí na dodatečné získání opomenutých prostředků
5.3	Zajištění testovacích dat (zákazník)	Tester musí mít veškeré prostředky okamžitě k dispozici	Zdržení při zajišťování až ve fázi příprav testů

5.4	Doplnění údajů do STS	Slouží k dokumentaci testerovy práce	Chaotická nebo nulová testerská dokumentace
Sekundární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	
*5.1	Tvorba vlastního generátoru dat	Urychlení přípravy dat pro jednotlivé testy	
*5.2	Tvorba/doplnění manuálů	Mohou pomoci nově příchozím testerům	

Specifikace testovacích činností

Popis: Cílem je zjistit, co vlastně bude náplní testerovy práce – definice typů testů na základě výstupu předchozích kroků, případně na základě další domluvy. Zahájení práce na STS.

Výstup: Výstupem tohoto procesu je základní orientace jaké typy a rozsahy testů budou použity a jaké další činnosti bude tester provádět.

Náročnost: cca 3-5%

Tabulka 11 – Popis procesu Specifikace testovacích činností

5.4	Doplnění údajů do STS	Slouží k dokumentaci testerovy práce	Chaotická nebo nulová testerská dokumentace
Sekundární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	
*5.1	Tvorba vlastního generátoru dat	Urychlení přípravy dat pro jednotlivé testy	
*5.2	Tvorba/doplnění manuálů	Mohou pomoci nově příchozím testerům	

Specifikace testovacích činností

Popis: Cílem je zjistit, co vlastně bude náplní testerovy práce – definice typů testů na základě výstupu předchozích kroků, případně na základě další domluvy. Zahájení práce na STS.

Výstup: Výstupem tohoto procesu je základní orientace jaké typy a rozsahy testů budou použity a jaké další činnosti bude tester provádět.

Náročnost: cca 3-5%

Tabulka 11 – Popis procesu Specifikace testovacích činností

Primární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	Rizika při nedodržení
6.1	Definice typů testů	Získání základního přehledu o náročnosti a rozsahu testování	Špatné odhady náročnosti testů či jejich neúplnost
6.2	Doplnění údajů do STS	Slouží k dokumentaci testerovy práce	Chaotická nebo nulová testerská dokumentace
Sekundární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	
*6.1	Tvorba/doplnění manuálů	Mohou pomoci nově příchozím testerům	

Plánování testů

Popis: Na základě předcházejících kroků je tester povinen vytvořit odhady na časovou náročnost své činnosti. Jedná se o jednu z nejtěžších činností a je třeba ji vyhradit dostatek času, aby byla provedena správně.

Výstup: Časové odhady jsou užitečné především pro vedoucí týmu, kteří mají na starost plánování a chod celého projektu.

Náročnost: cca 3-5%

Tabulka 12 – Popis procesu Plánování testů

Primární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	Rizika při nedodržení
7.1	Sepsání všech plánovaných testů a činností (odhady, psaní všech typů testů, revize testů, spouštění testů)	Zpřesnění a zpřehlednění plánovaných činností	Tvorba nepřesných odhadů
7.2	Ohodnocení těchto činností časovou náročností	Zpřesnění a zpřehlednění plánovaných činností	Tvorba nepřesných odhadů
7.3	Tvorba časových odhadů	Pro vhodné naplánování testerovy práce	Špatné rozvržení práce
Sekundární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	
*7.1	Pokud je to možné, je vhodné vytvářet odhady paralelně ve více lidech	Porovnání výstupů může zkvalitnit výsledný odhad	
*7.2	Tvorba/doplnění manuálů	Mohou pomoci nově příchozím testerům	

Implementace manuálních testů

Popis: Implementace manuálních testů je zřejmě nejobtížnější fází přípravy testování. Jedná se o sepsání TS a TC všech typů testů z kapitoly 5.6. Testy (především funkční a nefunkční) jsou psané dle SRS.

Výstup: Sepsané testy jsou po revizi určené k otestování kvality vyvíjeného SW.

Náročnost: cca 12-25%

Tabulka 13 – Popis procesu Implementace manuálních testů

Primární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	Rizika při nedodržení
8.1	Vytváření testů (ukázka viz. Příloha IV)	Vyšší efektivita tvorby testů	Nebezpečí přehlédnutí některých chyb
8.2	Testy sepisovat do definovaného prostředí/nástroje	Přehlednost	Špatná dodatečná údržba
8.3	Mapovat UseCases či URD na jednotlivé TS/TC	Zajištění otestování požadavků zákazníka	Netestování některých požadavků zákazníka
8.4	Připravit testovací data pro jednotlivé testy	Úplnost testů	Zdržení při přípravě dat během testování
8.5	Identifikace, které testy jsou určeny pro virtuální a které pro reálné zařízení	Specifikace, kde spouštět jednotlivé testy	Nemožnost otestování některých typů testů
Sekundární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	
*8.1	Psát objektivní názvy TS/TC a jejich popisy	Údaje, které čte zákazník	
*8.2	Doplňovat prioritu jednotlivých testů	Usnadnění výběru TS/TC pro regresní testování	

Implementace automatických testů

Popis: Automatické testy na poli mobilních zařízení jsou stále ve fázi vývoje a veškeré nástroje pro tento druh testování mají v držení externí firmy, které nabízejí pouze své služby testování – nikoli však nástroje.

Výstup: Pokud jsou automatické testy využity, mohou velmi usnadnit testerovu práci (hlavně v oblasti regresního testování).

Náročnost: cca 25-37%

Pozn.: *Postup v této kapitole lze aplikovat pouze ve specifických případech a její obsah je nad rámec původního zadání.*

Tabulka 14 – Popis procesu Implementace automatických testů

Primární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	Rizika při nedodržení
9.1	Zjištění, zda je možné využít automatických nástrojů	Vyšší efektivita práce	Pomalejší manuální testování, případně zbytečná investice do autom. testů
9.2	Instalace a nastavení nástrojů pro automatické testy	Vyšší efektivita práce	Pomalejší manuální testování
9.3	Psaní automatických testů (dle specifikace)	Vyšší efektivita práce	Pomalejší manuální testování
9.4	Identifikace, které testy jsou určeny pro virtuální a které pro reálné zařízení	Specifikace, kde spouštět jednotlivé testy	Nemožnost otestování některých typů testů
Sekundární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	
*9.1	-	-	

Revize testů

Popis: Revize testů má za úkol zjistit, zda připravené testy pokrývají všechny funkcionality systému, které mají být implementovány v testované verzi. Tuto revizi by měla provádět jiná osoba než autor testů. Nalezené nedostatky je třeba opravit před spuštěním testů.

Výstup: Výstupem jsou revidované testy, které jsou připraveny pro kompletní otestování aktuálně vydané verze.

Náročnost: cca 7-12%

Tabulka 15 – Popis procesu Revize testů

Primární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	Rizika při nedodržení
10.1	Nezávisle identifikovat všechny zákaznikovy požadavky na testovanou verzi	Nezávislost může pomoci odhalit přehlédnuté požadavky	Nedokonalá identifikace požadavků
10.2	Kontrola pokrytí testů na verifikaci chyb z minulých verzí	Nutnost verifikovat dříve nalezené chyby.	Možnost nepřetestování starších chyb
10.3	Kontrola pokrytí těchto požadavků	Revize testů	Zbytečnost revize
10.4	Úprava/oprava připravených testů	Revize testů	Zbytečnost revize
Sekundární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	
*10.1	Kontrola a případná úprava časových odhadů na testování	Možnost odhalení přehlédnutých časových nákladů	
*10.2	Tvorba/doplnění manuálů	Mohou pomoci nově příchozím testerům	

Provedení testů – virtuální emulátor

Popis: Tento proces se zabývá samotným spouštěním testů na virtuálním zařízení. Toto testování by však mělo být pokud možno vždy doplněno i o testování na zařízení reálném.

Výstup: Výstupem jsou výsledky spuštění jednotlivých testů na reálném zařízení.

Náročnost: cca 7–12%

Tabulka 16 – Popis procesu Provedení testů – virtuální simulátor

Primární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	Rizika při nedodržení
11.1	Instalace verze určené pro testování	Pro otestování správné verze	Otestování špatné či nedokončené verze
11.2	Příprava testovacího prostředí dle STS	Pro otestování dle správných vstupních požadavků	Nepřesné výsledky testování
11.3	Příprava testovacích dat dle STS	Pro otestování dle správných vstupních požadavků	Nepřesné výsledky testování
11.4	Spuštění a vyhodnocení jednotlivých testů	Samotné testování	-
11.5	Spuštění tzv. performance testů	Možné pouze u virtuálních zařízení	Nemožnost pozdějšího otestování na reálném zařízení
Sekundární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	
*11.1	Zaznamenávat poznatky z testování (úpravy TS/TC, návrhy na vylepšení postupů apod.)	Vhodné pro údržbu testů před další iterací	

Provedení testů – reálné zařízení

Popis: Tento proces se zabývá samotným spouštěním testů na reálném zařízení.

Výstup: Výstupem jsou výsledky spuštění jednotlivých testů na reálném zařízení.

Náročnost: cca 7–12%

Tabulka 17 – Popis procesu Provedení testů – reálné zařízení

Primární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	Rizika při nedodržení
12.1	Instalace verze určené pro testování	Pro otestování správné verze	Otestování špatné či nedokončené verze
12.2	Příprava testovacího prostředí dle STS	Pro otestování dle správných vstupních požadavků	Nepřesné výsledky testování
12.3	Příprava testovacích dat dle STS	Pro otestování dle správných vstupních požadavků	Nepřesné výsledky testování
12.4	Spuštění a vyhodnocení jednotlivých testů	Samotné testování	-
Sekundární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	
*12.1	Zaznamenávat poznatky z testování (úpravy TS/TC, návrhy na vylepšení postupů apod.)	Vhodné pro údržbu testů před další iterací	

Report a verifikace chyb

Popis: Po testování (pokud nebylo realizováno během něj) přichází na řadu reportování a verifikace chyb. Tento proces se zabývá popisem činností, na které je třeba brát ohled. Tester by měl chyby reportovat tak, aby chyb byla již z reportu jasně a přesně pochopitelná.

Výstup: Tester bude schopen reportovat chyby přesně a jasně, aby je vývojáři byli schopni identifikovat a opravit bez dodatečné komunikace s testerem.

Náročnost: cca 5–7%

Tabulka 18 – Popis procesu Report a verifikace chyb

Primární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	Rizika při nedodržení
13.1	Verifikace opravených chyb (opraveno/neopraveno)	Nutnost zaznamenání, které chyby jsou skutečně opraveny a které nikoliv	Nepřehlednost v kontrolách již opravených chyb
13.2	Identifikovat co chyba je a co není (proti SRS)	Objektivní report chyb, možnost odhalení chyby v samotných testech	Reportování tzv. falešných chyb
13.3	Název chyby - uvádět místo chyby i v prefixu názvu chyby	Odpověď na otázku: „Kde se chyba vyskytuje?“	Nepřehlednost v reportovaných chybách
13.4	Stručný popis chyby	Odpověď na otázku: „Jak se to chová?“	Nejasný popis chyby = dodatečná komunikace s vývojářem
13.5	Očekávaný výsledek	Odpověď na otázku: „Jak by se to mělo chovat?“	Nejasný popis chyby = dodatečná komunikace s vývojářem
13.6	Frekvence výskytu, reprodukovatelnost	Odpověď na otázku: „Vyskytuje se chyba pravidelně či náhodně?“	Nejasný popis chyby = dodatečná komunikace s vývojářem
13.7	Kroky k reprodukci	Odpověď na otázku: „Jak chybu zopakuji?“	Nejasný popis chyby = dodatečná komunikace s vývojářem
13.8	Priorita	Odpověď na otázku: „Jak rychle je nutné chybu opravit?“	Špatná posloupnost oprav chyb
13.9	Verze testovaného produktu	Odpověď na otázku: „Ve které verzi se chyba vyskytuje?“	Nejasný popis chyby = dodatečná komunikace s vývojářem
13.10	Verze a nastavení testovacího prostředí	Odpověď na otázku: „Na jakém HW/SW se chyba vyskytuje“	Nejasný popis chyby = dodatečná komunikace s vývojářem
13.11	Použitá testovací data (např. příloha k reportu)	Odpověď na otázku: „Při jakých datech se chyba vyskytuje?“	Nejasný popis chyby = dodatečná komunikace s vývojářem
13.12	Dodatečné informace (např. přístupový účet, použitý server)	Vhodné pro bližší specifikaci chyby	Nejasný popis chyby = dodatečná komunikace s vývojářem
13.13	Přílohy (screenshot, log, atd.)	Urychlení identifikace a opravy chyby	Nejasný popis chyby = dodatečná komunikace s vývojářem

Sekundární cíle procesu		
Krok	Popis činnosti	Důvody zavedení
*13.1	Pokud není automaticky pak: Unikátní ID chyby	Lepší přehlednost v reportech chyb
*13.2	Pokud není automaticky pak: Jméno testera	Lepší přehlednost v reportech chyb
*13.3	Pokud není automaticky pak: Datum nalezení	Lepší přehlednost v reportech chyb
*13.4	Snaha přiřazovat chyby k TS/TC, ve kterých byla nalezena	Usnadnění verifikace chyb
*13.5	Tvorba/doplnění manuálů	Mohou pomoci nově příchozím testerům

Vyhodnocení výsledků testů

Popis: Po dokončení všech testů je nutné vypracovat tzv. System Test Report (STR). Tento dokument slouží jako doklad o provedené práci jak pro zákazníka, tak pro vedoucího týmu a vývojářský tým.

Výstup: Výstupem tohoto procesu bude STR, který bude obsahovat veškeré údaje o dosavadním testování.

Náročnost: cca 3–5%

Tabulka 19 – Popis procesu Vyhodnocení výsledků testů

Primární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	Rizika při nedodržení
14.1	Tvorba základní struktury STR (obvykle z STS)	Zachování formátu STS versus STR (a urychlení práce)	Nepřehlednost výsledného reportu
14.2	Kontrola/doplnění testované verze	Nutnost specifikace testované verze	Neúplnost výsledného reportu
14.3	Kontrola/doplnění použitého testovacího prostředí	Pro sledování objektivnosti testování	Neúplnost výsledného reportu
14.4	Kontrola/doplnění použitých testovacích dat	Pro sledování objektivnosti testování	Neúplnost výsledného reportu
14.5	Kontrola/doplnění plánovaných testů	Revize	Neúplnost výsledného reportu
14.6	Kontrola/doplnění výsledků testů (Pass/Failed)	Základní údaje celého STR	Neúplnost výsledného reportu
14.7	Kontrola/doplnění výsledků testů (ID + krátký popis nalezených chyb)	Základní údaje celého STR	Neúplnost výsledného reportu
14.8	Kontrola/doplnění jmen jednotlivých testerů	Přehlednost	Neúplnost výsledného reportu
14.9	Kontrola/doplnění dat spuštění testů	Přehlednost	Neúplnost výsledného reportu

Sekundární cíle procesu		
Krok	Popis činnosti	Důvody zavedení
*14.1	Možnost doplnění různých statistik/grafů z testování	Ocení především zákazník či project manager
*14.2	Možnost doplnění reálné časové náročnosti jednotlivých činností	Užitečné pro project managera a tvorbu příštích časových odhadů

Údržba testů

Popis: Pokud bylo testování jen pro určitý vývojový cyklus (např. tzv. sprint v metodice SCRUM), či pokud byly v průběhu testů nalezeny chyby, bude nutné testy (nebo jejich část) spustit znova. Před tímto spuštěním by však měla proběhnout určitá údržba (aktualizace/oprava) testů.

Výstup: Tento proces se zabývá identifikací kroků, které je nutné provést před další testovací iterací.

Náročnost: cca 5–7%

Tabulka 20 – Popis procesu Údržba testů

Primární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	Rizika při nedodržení
15.1	Vyhodnocení a zpracování vstupů od zákazníka, tzv. customer feedback	Zpracování požadavků zákazníka	Nespokojenost zákazníka
15.2	Vyhodnocení a zpracování vstupů od vedoucího týmu, tzv. PM feedback	Zpracování požadavků zákazníka	Nespokojenost zákazníka
15.3	Případné opravy/doplnění TS/TC	Vyšší kvalita testů pro příští testovací iteraci	Nepřesné testy
15.4	Případné opravy/doplnění regresních testů	Vyšší kvalita testů pro příští testovací iteraci	Nepřesné testy
15.5	Doplnění kontroly nalezených/neopravených chyb do příští testovací iterace	Umožnění např. zohlednění vyšších časových nároků na spouštění testů	Přehlednutí kontroly opravených chyb v příští testovací iteraci
15.6	Vrácení zařízení (a případně i generátorů dat) do výchozího nastavení	Nastavení stejných podmínek pro další testování	Nebezpečí odlišností ve výsledcích příštích testů
Sekundární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	
*15.1	Vyhodnocení možnosti použití automatizace některých procesů testování	Urychlení práce v příští iteraci	

Závěr testování

Popis: Tento proces následuje po skončení testovacích prací na projektu a slouží spíš jako doporučení pro testera, kterých kroků by se měl v takovém případě držet.

Výstup: Výstupem je tzv. čisté ukončení působnosti testera na daném projektu.

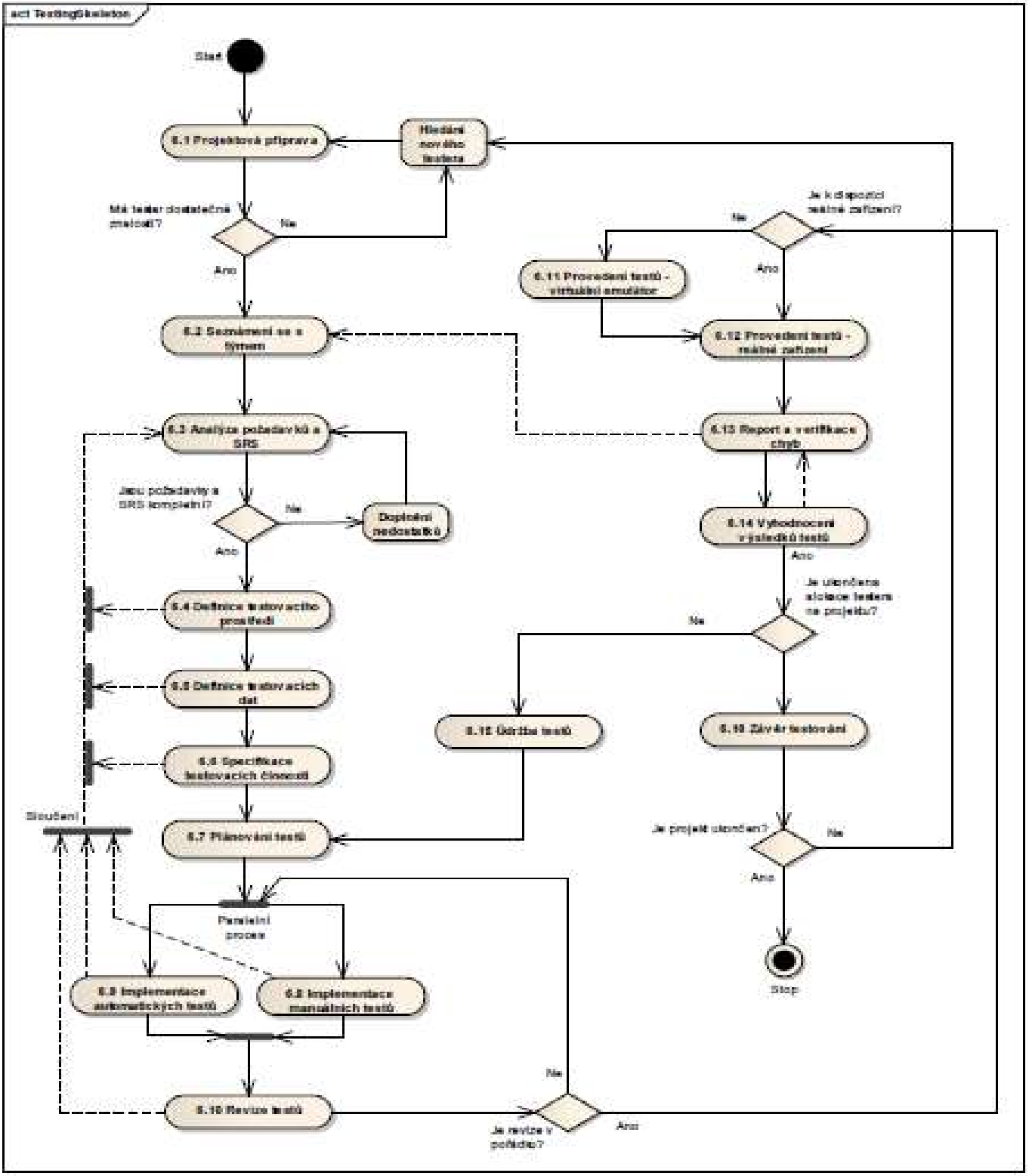
Náročnost: cca 7–12%

Tabulka 21 – Popis procesu Závěr testování

Primární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	Rizika při nedodržení
16.1	Uzavření/předání všech vlastních úkolů na projektu	Dokončení všech alokovaných úkolů	Nebezpečí dodatečné uzavírání úkolů nezpůsobilými lidmi
16.2	Doplnění veškeré dokumentace (např. manuály)	Uzavření práce na dokumentech	Nebezpečí dodatečné dokončování dokumentů nezpůsobilými lidmi
16.3	Sepsání zkušeností s projektem (tzv. Lessons Learnt)	Může pomoci k lepším výsledkům při příštím projektu	Opakování stejných chyb i při příštím projektu
16.4	Nastavení zapůjčených zařízení do původního nastavení	Důležité pro postoupení zařízení na další projekty	Chybně nastavená zařízení, práce navíc na dalších projektech
16.5	Vrácení zapůjčených zařízení (včetně příslušenství)	Důležité pro postoupení zařízení na další projekty	Nedostatek zařízení pro další projekty
16.6	Úklid svěřených projektových složek na serverech	Přehlednost projektových složek	Nekonzistence projektových složek
16.7	Úklid dat na lokálním počítači	Uvolnění místa	Postupné vyčerpání kapacity pevného disku
16.8	Případné předání funkce a školení nového testera	Plynulé začlenění nového člověka do týmu	Zdlouhavé začlenění nového testera do týmu
Sekundární cíle procesu			
Krok	Popis činnosti	Důvody zavedení	
*16.1	Tvorba/doplnění manuálů	Mohou pomoci nově příchozím testerům	

Vývojový diagram

Vývojový diagram testing skeletonu skládající se z jednotlivých kroků z předchozí kapitoly a jejich posloupnosti.



Obrázek 6 – Diagram průchodu testing skeletonem

TESTOVÁNÍ APLIKACE PRO ANDROID

Tato kapitola se zabývá simulací praktického nasazení předcházejícího testing skeletonu při testování aplikace pro zařízení s operačním systémem Android.

Popis aplikace

Pro účely otestování byla zvolena aplikace ThinkFree Office. Jedná se o mobilní provedení kancelářského balíku, který umožňuje otevírání, editaci a tvorbu dokumentů typu *.ppt, *.doc, *.xls.

Obrázek 7 – Náhledy aplikace ThinkFree Office pro Android [32]

Vstupy

V tomto testu bude testována jeho verze ThinkFree Mobile Viewer, která umožňuje pouze čtení dokumentů a je často k dispozici na nových mobilních zařízeních (čili bez nutnosti dodatečné instalace).

Vstupní požadavky

N/A – nejedná se o zákaznický projekt, v tomto případě budou tedy uvažovány pouze dva zdroje požadavků:

1. Help (pro funkční testování)
2. Obecná funkcionalita (viz kapitola Nevyslovené požadavky)

Testovací prostředí

Aplikace je určena pro nový mobilní telefon na trhu – LG Optimus One, na kterém bude distribuována již při koupi. Z tohoto důvodu je nutné aplikaci otestovat právě na tomto zařízení a při jeho továrním nastavení.

Testovací data

Testovací data nejsou dodána. Je nutné si je vytvořit manuálně. Dle zadání mají být otestovány formáty *.doc, *.docx, *.ppt, *.pptx, *.xls, *.xlsx, *.pdf.

Výstup podle testing skeletonu

Tato kapitola uvádí samotný postup činnosti testera dle navrhnutého testing skeletonu. Každá podkapitola obsahuje tabulku procesů a určitý výstup. Z hlediska objemu výstupních dat a dokumentace je výstup omezen na slovní vyjádření či odkaz na přílohu této práce.

Projektová příprava

Tabulka 22 – Projektová příprava - Výstup

Krok	Popis činnosti	Výstup
1.1	Seznámit se s cílem projektu	Otestování kancelářského balíku v jeho Lite verzi pro telefon LG Optimus One = vytvoření seznamu případných chyb
1.2	Pochopit použité technologie	Využito pouze OS Android a ovládacích prvků telefonu (funkční black-box testování)
1.3.	Zjistit jazyk pro komunikaci s klientem	Dokumentace v angličtině, STS a STR pro české zastoupení v češtině
*1.1	Zjistit bližší informace o klientovi	Společnost zabývající se vývojem kancelářského SW pro Internet a mobilní zařízení. Další informace na stránkách společnosti. (Možnost nabídnutí služeb dalšího testování)
*1.2	Zvážit crowdsourced testování	Z důvodu velikosti a funkcionalitě Lite verze není nutné

Seznámení se s týmem

Tabulka 23 – Seznámení se s týmem - Výstup

Krok	Popis činnosti	Výstup
2.1	**Definice kompetencí jednotlivých členů týmu**	V tomto případě se jedná pouze o jednoho externího testera, který komunikuje přímo s project managerem (zástupce zadavatele)
2.2	**Zpřístupnění projektových adresářů**	Aplikace zpřístupněna na vnitřním FTP společnosti
2.3	**Začlenění do projektové e-mail skupiny (+ vytvoření pravidel v Outlooku)**	Projektová skupina neexistuje – komunikace probíhá jen mezi testerem a project managerem
2.4	**Přístup do bugsystému (Mantis, Bugzilla, TFS)**	Chyby budou značeny přímo k jednotlivým TS (viz. ukázka v Příloze V)
2.5	**Centralizace projektových úkolů (office documents, TFS, XPlanner)**	N/A
2.6	**Instalace ostatních projektových prostředí**	E-mail, FTP-klient, Skype
*2.1	**Žádost o informování o nové verzi a místu jejího uložení (pomocí email/skype)**	Jedná se pouze o jednu iteraci testování
*2.2	**Žádost o umístění verzování přímo do aplikace (název, verze, datum)**	Společnost má vlastní způsob verzování
*2.3	**Žádost o zveřejňování úprav v každé vydané verzi (i interní)**	Verze bude testována jako celek dle kompletní dokumentace
2.4	**Při opravách bugů by měli vývojáři definovat, ve které verzi se oprava vyskytne**	N/A
2.5	**Zavedení tvorby logu přímo v mobilním zařízení**	N/A
2.6	**Zavedení možnosti vytváření screenshots přímo v aplikaci**	N/A

Analýza požadavků a SRS

Tabulka 24 – Analýza požadavků a SRS - Výstup

Krok	Popis činnosti	Výstup
3.1	**Nastudování SRS**	Nejedná se o zákaznický projekt. Jediná dokumentace je k dispozici na [32]
3.2	**Analýza URD a SRS (viz. Příloha III)**	Nejedná se o zákaznický projekt. Dodatečné informace budou vykomunikovány s odpovědným project managerem.
3.4	**Založení STS (tvorba šablony dokumentu)**	Přebrána STS šablona zadavatele
3.1	**Nastudování SRS**	Přečtena projektová dokumentace – User Guide
*3.1	**Rezervace/zapůjčení cílových mobilních zařízení**	Vyhrazen jeden LG Optimus One pro testovací účely
*3.2	**Brainstorming – nechybí něco?**	N/A

6.3.4 Definice testovacího prostředí

Tabulka 25 – Definice testovacího prostředí - Výstup

Krok	Popis činnosti	Výstup
4.1	**Identifikace potřebného HW a SW (včetně licencí) z SRS**	Zadání hovoří pouze o jednom zařízení – LG Optimus One (Android 2.2 Froyo), tovární nastavení, je nutné vytvořit google účet
4.2	**Zajištění těchto prostředků – Odblokované telefony, SIM karty (přesné konfigurace)**	Telefon zapůjčen včetně kompletního příslušenství (SD karta, nabíječka, USB kabel, manuál)
4.3	**Obstarání uživatelských manuálů ke všem zařízením**	Viz. předchozí bod
4.4	**Instalace a nastavení SW (virtuální simulátor)**	Využit AVD emulátor od Google. V něm vytvořeno zařízení, které je přesnou kopií LG Optimus One. Bohužel emulátor je i na 2-jádrovém procesoru velmi pomalý
4.5	**Nastavení HW (dle SRS)**	Testování bude probíhat při továrním nastavení zařízení
4.6	**Zavedení okolního prostředí**	Specifická okolní prostředí (hluk, tma, …) nejsou známy
4.7	**Doplnění údajů do STS**	Údaje o HW a SW zaneseny do připraveného STS
*4.1	**Vyhodnocení nutnosti tzv. field testu**	Není nutno

*4.2	Tvorba/doplnění manuálů	Z důvodu jednoduchosti přípravy testovacího prostředí není pro toto zaveden žádný manuál

6.3.5 Definice testovacích dat

Tabulka 26 – Definice testovacích dat - Výstup

Krok	Popis činnosti	Výstup
5.1	Určení typu dat – dynamická/statická data (SRS/Zákazník)	Data jsou statická – office dokumenty. Tato data jsou čtena z paměťové karty telefonu
5.2	Identifikace potřebného vzorku testovacích dat (SRS/Zákazník)	Vytvořeny 2 vzorky (prázdný soubor a soubor s údaji) pro každý typ kompatibilního dokumentu
5.3	Zajištění testovacích dat (Zákazník)	Tvorba testovacích dokumentů na PC (MS Office 2003 a 2007)
5.4	Doplnění údajů do STS	Údaje o použitých testovacích datech zaneseny do STS
*5.1	Tvorba vlastního generátoru dat	Není potřeba
*5.2	Tvorba/doplnění manuálů	Z důvodu jednoduchosti přípravy testovacího prostředí není pro toto zaveden žádný manuál

6.3.6 Specifikace testovacích činností

Tabulka 27 – Specifikace testovacích činností - Výstup

Krok	Popis činnosti	Výstup
6.1	Definice typů testů	Zákazník požaduje pouze funkční black-box testování a testy použitelnosti. Online sekce nebude testována.
6.2	Doplnění údajů do STS	Typ zvolených testů doplněn do STS
*6.1	Tvorba/doplnění manuálů	N/A

6.3.7 Plánování testů

Tabulka 28 – Plánování testů - Výstup

Krok	Popis činnosti	Výstup
7.1	Sepsání všech plánovaných testů a činností (odhady, psaní všech typů testů, revize testů, spouštění testů)	Pouze funkční black-box a usability testy
7.2	Ohodnocení těchto činností časovou náročností	Shodná náročnost
7.3	Tvorba časových odhadů	Cca 10MD (1MD = 8h)
*7.1	Pokud je to možné, je vhodné vytvářet odhady paralelně ve více lidech	N/A
*7.2	Tvorba/doplnění manuálů	N/A

6.3.8 Implementace manuálních testů

Tabulka 29 – Implementace manuálních testů - Výstup

Krok	Popis činnosti	Výstup
8.1	Vytváření testů (ukázka viz. Příloha IV)	Ukázka použitých TS viz. Příloha V
8.2	Testy sepisovat do definovaného prostředí/nástroje	Testy sepisovány do šablony klienta
8.3	Mapovat UseCases či URD na jednotlivé TS/TC	N/A
8.4	Připravit testovací data pro jednotlivé testy	Data přiložena k testům
8.5	Identifikace, které testy jsou určeny pro virtuální a které pro reálné zařízení	Oba typy testů jsou vhodné jak pro virtuální, tak pro reálné zařízení
*8.1	Psát objektivní názvy TS/TC a jejich popisy	Viz. Příloha V
*8.2	Doplňovat prioritu jednotlivých testů	V tomto případě není nutné

6.3.9 Implementace automatických testů

Jelikož se jedná o jedno-iterační testování, není nutné vytvářet automatické testy.

6.3.10 Revize testů

Tabulka 30 – Revize testů - Výstup

Krok	Popis činnosti	Výstup
10.1	Nezávisle identifikovat všechny zákaznikovy požadavky na testovanou verzi	Z důvodu přítomnosti pouze jednoho test analytika je jim revize provedena vůči uživatelské příručce k programu a vůči doporučením v této práci
10.2	Kontrola pokrytí testů na verifikaci chyb z minulých verzí	N/A
10.3	Kontrola pokrytí těchto požadavků	Nalezeny drobné nedostatky a překlepy

10.4	Úprava/oprava připravených testů	Úprava aplikována
*10.1	Kontrola a případná úprava časových odhadů na testování	N/A
*10.2	Tvorba/doplnění manuálů	N/A

6.3.11 Provedení testů

Tabulka 31 – Provedení testů - Výstup

Krok	Popis činnosti	Virtuální emulátor	Reálné zařízení
11.1	Instalace verze určené pro testování	Tvorba virtuálního zařízení – kopie reálného	Zařízení je připraveno z výroby
11.2	Příprava testovacího prostředí dle STS	Spuštění virtuálního zařízení	Zařízení je připraveno z výroby
11.3	Příprava testovacích dat dle STS	Nakopírování dat na virtuální SD kartu skrze IDE Eclipse	Nakopírování dat na SD kartu
11.4	Spuštění a vyhodnocení jednotlivých testů	Viz. Příloha V	Viz. Příloha V
*11.1	Zaznamenávat poznatky z testování (úpravy TS/TC, návrhy na vylepšení postupů apod.)	N/A	N/A

6.3.12 Report a verifikace chyb

Tabulka 32 – Report a verifikace chyb - Výstup

Krok	Popis činnosti	Výstup
13.1	Verifikace opravených chyb (opraveno/neopraveno)	N/A
13.2	Identifikovat co chyba je a co není (proti SRS)	Nalezené chyby verifikovány vůči uživatelské příručce
13.3	Název chyby - uvádět místo chyby i v prefixu názvu chyby	Doplněno do STR
13.4	Stručný popis chyby	Doplněno do STR
13.5	Očekávaný výsledek	Doplněno do STR
13.6	Frekvence výskytu, reprodukovatelnost	Doplněno do STR
13.7	Kroky k reprodukci	Doplněno do STR
13.8	Priorita	Doplněno do STR
13.9	Verze testovaného produktu	Doplněno do STR
13.10	Verze a nastavení testovacího prostředí	Doplněno do STR
13.11	Použitá testovací data (např. příloha k reportu)	Doplněno do STR
13.12	Dodatečné informace (např. přístupový účet, použitý server)	N/A
13.13	Přílohy (screenshot, log, atd.)	Tvorba fotek při testování na reálném zařízení, tvorba screenshots na virtuálním zařízení
*13.1	Pokud není automaticky pak: Unikátní ID chyby	Řešeno autoinkrementací ID chyby
*13.2	Pokud není automaticky pak: Jméno testera	Není nutno – pouze jeden tester
*13.3	Pokud není automaticky pak: Datum nalezení	Doplněno do STR
*13.4	Snaha přiřazovat chyby k TS/TC, ve kterých byla nalezena	Doplněno do STR
*13.5	Tvorba/doplnění manuálů	N/A

6.3.13 Vyhodnocení výsledků testů

Tabulka 33 – Vyhodnocení výsledků - Výstup

Krok	Popis činnosti	Výstup
14.1	Tvorba základní struktury STR (obvykle z STS)	Chyby přepsány do šablony STR (vznik úpravou STS)
14.2	Kontrola/doplnění testované verze	Doplněno
14.3	Kontrola/doplnění použitého testovacího prostředí	Převzato z STS
14.4	Kontrola/doplnění použitých testovacích dat	Převzato z STS
14.5	Kontrola/doplnění plánovaných testů	Převzato z STS
14.6	Kontrola/doplnění výsledků testů (Pass/Failed)	Doplněno
14.7	Kontrola/doplnění výsledků testů (ID + krátký popis nalezených chyb)	Doplněno (viz. Příloha V)

14.8	Kontrola/doplnění jmen jednotlivých testerů	Převzato z STS
14.9	Kontrola/doplnění dat spuštění testů	Doplněno
*14.1	Možnost doplnění různých statistik/grafů z testování	N/A
*14.2	Možnost doplnění reálné časové náročnosti jednotlivých činnosti	N/A

6.3.14 Údržba testů

V tomto případě se nevykonává. Testy jsou vykonány pouze jednou a následně zálohovány a předány zákazníkovi. Při dalším testování se testy upraví dle nové specifikace.

6.3.15 Závěr testování

Tabulka 34 – Závěr testování - Výstup

Krok	Popis činnosti	Výstup
16.1	Uzavření/předání všech vlastních úkolů na projektu	Kontrola splnění všech testovacích úkolů
16.2	Doplnění veškeré dokumentace (např. manuály)	STS i STR kompletní – předáno zákazníkovi
16.3	Sepsání zkušeností s projektem (tzv. Lessons Learnt)	Zkušenosti zaznamenány do poznámek k testování mobilních aplikací
16.4	Nastavení zapůjčených zařízení do původního nastavení	Pomocí volby v menu, u virtuálního zařízení reset nastavení
16.5	Vrácení zapůjčených zařízení (včetně příslušenství)	Reálné zařízení vráceno
16.6	Úklid svěřených projektových složek na serverech	N/A
16.7	Úklid dat na lokálním počítači	Zálohovány testy i test data
16.8	Případné předání funkce a školení nového testera	N/A
*16.1	Tvorba/doplnění manuálů	N/A

Testujme s rozumem: Seriál

Jak rozumně začít testovat malý nebo středně velký softwarový projekt? Jako specialista na oblast testování a kvality se čas od času dostanu do rozjetého projektu, kde mám pomoci při testování aplikací. Povětšinou se nejedná bohužel o výpomoc, ale o záchranu projektu. Proč k tomu dochází?

Málokdy jsem se setkal s rozumnou přípravou práce pro testery. Povětšinou neexistovaly ani definované cíle testování, natož popsané jednotlivé testované případy apod. Ale ve většině případů testeři netestovali to, co bylo skutečně zapotřebí. Pokud začínáte projekt a potřebujete začít nějakým rozumným způsobem řídit testování a aktivity testerů, právě pro vás je určena tato série článků. Provede vás postupně celým procesem testování.

Jak začít?

Máme projekt, vyhranou zakázku nebo prostě potřebujeme začít testovat. Jak začít? Co dát testerům?

Potřebujete klíčové vlastnosti

Začněte tím, že si definujte (pokud ještě nemáte) podmínky úspěchu. Odpovězte si na otázku: Co musí software (produkt) splňovat, aby byl úspěšný? Co musí umět?

Příklad:

Představme si implementaci systému správy výpůjček do knihovny. Co náš systém musí splňovat?

- Systém umožní vyhledat konkrétního klienta knihovny.
- Systém umožní vyhledat konkrétní knihu k vypůjčení.
- Systém umožní zobrazit historii výpůjček daného klienta.
- Systém poskytne statistiku o vypůjčovaných knihách.

Co se může stát

Nyní si projděte seznam těchto klíčových vlastností a definujte, co všechno se může zkazit, přerušit, přestat fungovat a tím ohrozit splnění požadavků na produkt. Jinými slovy, identifikujte produktová rizika.

Příklad:

Co se může přihodit v našem knihovním systému?

- Seznam uživatelů nemusí být dostupný.
- Seznam knih nemusí být dostupný.
- Systém neuloží výpůjčku, knihy nepůjde půjčit.
- Systém neuloží vrácení knihy, uživatelé budou následně pokutováni.

Ohodnoťte rizika

Dopad:

Jednotky uživatelů	1
Skupiny uživatelů	2
Všichni uživatelé	3

Pravděpodobnost:	
Pravděpodobně nenastane	1
Může nastat	2
Velmi pravděpodobně nastane	3

Nyní provedeme kvalifikaci rizik. Není to nic těžkého. Pouze ohodnotíme, jak jsou daná rizika pro nás podstatná. Jinými slovy, jak moc si jich musíme všímat. Ohodnotíme si rizika například od 1 do 3, a to dle dopadu a pravděpodobnosti výskytu. S tím, že 1 bude nejméně vážný stav a 3 nejhorší možnost. Můžete si zvolit svoje rozsahy a zvolit dokonce vlastní typ rizika. Například místo dopadu, můžete v určitých aplikacích kalkulovat s finanční ztrátou.

Příklad:

Riziko	**Dopad**	**Pravděpodobnost**
Systém uživatelů nemusí být dostupný.	3	1
Seznam knih nemusí být dostupný.	2	1
Systém neuloží výpůjčku, knihy nepůjde půjčit.	3	2
Systém neuloží vrácení knihy, uživatelé budou následně pokutováni	3	3

Definujte si tabulku závažností

Zde doporučuji použít metodu MoSCoW. Metoda dělí jednotlivá rizika na Must test, Should test, Could test a Won't test. Hodnoty v tabulce jsou součinem jednotlivých vah vynesených na jednotlivých osách tabulky.

Závažnost = Dopad * Pravděpodobnost

Bodové hodnocení je čistě na vás. Zde pro názornost jsme je zvolili následovně:

Příklad:

		Pravděpodobnost		
		1	2	3
Dopad	1	1	2	3
	2	2	4	6
	3	3	6	9

Závažnost	**Zvolená kategorie**
1	Won't test (nemusíme testovat)
2-3	Could test (můžeme testovat)
4-5	Should test (měli bychom testovat)
6-9	Must test (musíme testovat)

K čemu to bylo dobré

Nyní máme identifikovaná produktová rizika projektu, máme definovanou jejich závažnost a víme, které jsou pro nás z hlediska úspěšnosti projektu důležité a kterým je zapotřebí věnovat nejvíce síly a energie. Můžeme rovněž prohlásit, že ověření těchto rizik jsou naše cíle testování (Test target).

Příklad:

Riziko	**Závažnost**	**Kategorie**
Systém uživatelů nemusí být dostupný.	3	Should test
Seznam knih nemusí být dostupný.	2	Could test
Systém neuloží výpůjčku, knihy nepůjde půjčit.	6	Must test
Systém neuloží vrácení knihy, uživatelé budou následně pokutováni.	9	Must test

Co dále

Pokud máte k dispozici test analytika, ten by měl být schopen s tímto vstupem efektivně pracovat. Pokud ho nemáte a není k dispozici senior tester, musíme ještě naši práci doplnit testovanými scénáři (test cases). Získání „Test case" z „Use case" a další metody získávání potřebných scénářů pro testování budou náplní

Testujeme s rozumem (2.) – Jak z UC získat TC

Při přípravě na testování projektu je zapotřebí vytvořit scénáře (postupy), které budou testeři procházet při testování. Scénáře by v optimálním případě měly pokrývat případy užití, pro které je aplikace vyvíjena. Máme-li rozumně zpracovanou analýzu, může být získání těchto scénářů relativně snadné.

Tyto scénáře testů nazýváme „testovanými scénáři", anglicky test case, zkratkou TC. Co je to TC? Co by měl obsahovat? TC je soubor vstupů, kroků i podmínek a očekávaných výstupů. Ale kde ho vezmeme? Každá analýza vyvíjeného systému by měla obsahovpat případy užití (use cases), dále UC. Tyto scénáře užití lze s výhodou použít pro vytvoření seznamu TC. Jak na to? Ukážeme si na následujícím příkladu.

Představte si jednoduchý systémový UC na evidenci vydané faktury v nějakém systému. Daný uživatel má právo zadávat faktury pouze do 100 tisíc.

Basic Flow

- 1. Uživatel zadá číslo faktury.
- 2. Systém provede kontrolu čísla faktury.
- 3. Uživatel vyplní fakturovanou částku.
- 4. Systém provede kontrolu výše částky.
- 5. Uživatel vyplní datum splatnosti.
- 6. Systém provede kontrolu data splatnosti.
- 7. Systém vytvoří fakturu a podá uživateli zprávu o jejím úspěšném zaevidování.

Nyní máme zpracovaný zcela jednoduchý UC, kde máme základní průchod aplikací. Bohužel svět není jednoduchý a ani náš příklad nemůže být takto jednoduchý. Je zapotřebí se zamyslet a definovat další toky, kterými může tento scénář být přerušen, či pokračovat zcela jinou cestu. Jedná se o tzv. Alternate flows.

Alternate flow 1

Faktura v systému již existuje, nelze přidat.

- Basic Flow v bodě 2. Systém podá uživateli zprávu o existenci faktury tohoto čísla a UC se tímto ukončí (faktura se nevytvoří).

Alternate flow 2

Co když částka na faktuře přesáhne hodnotu 100 tis Kč?

- Basic Flow v bodě 4. Systém podá uživateli zprávu o převýšení maximální hodnoty faktury a UC se tímto ukončí (faktura se nevytvoří).

Alternate flow 3

Co když zadáme datum splatnosti v minulosti?

- Basic Flow v bodě 6. Systém podá uživateli zprávu o špatné hodnotě data splatnosti a UC se tímto ukončí (faktura se nevytvoří).

Nyní si sepíšeme do tabulky pod sebe všechny možné scénáře, a pojmenujeme si je. Pojmenování volte rozumně, neboť by se mělo používat skrze celé testování.

Sestavení a pojmenování scénářů

Scénář 1 – založení faktury	Basic Flow
Scénář 2 – faktura tohoto čísla již v systému existuje	Basic FlowAlternate flow 1
Scénář 3 – částka vyšší než 100 tisíc	Basic FlowAlternate flow 2
Scénář 4 – datum splatnosti v minulosti	Basic FlowAlternate flow 3

Teď již máme identifikovány hlavní scénáře, které budeme potřebovat otestovat. Pro vlastní testování však toto ještě nestačí. Musíme definovat vstupy a očekávané výstupy jednotlivých TC. Vstupy lze identifikovat dle rozhodovacích bloků v UC. Každé alternative flow by mělo být způsobeno nějakou příčinou, vstupem. V našem případě máme situaci poněkud zjednodušenou. V reálných systémech jsou i desítky vstupů. Zde lze s výhodou použít excel, či vhodný TC management nástroj. V našem případě dochází k startu alternate flow 3 v scénáři č.4: Datum splatnosti v minulosti. Je tedy evidentní, že v tomto scénáři se rozhodne o provedení alternativního scénáře v bodě 6. Tudíž je zapotřebí zadat číslo faktury, částku a datum splatnosti. Číslo faktury a částka je validní a datum splatnosti je neplatné (Invalid). Další vstupy jsou již pro tento scénář irelevantní a v tabulce se neobjeví. Nyní vyplňme tabulku. „V" představuje potřebný validní vstup a „I" invalidní vstup. Pokud na vstupech nezáleží nebo jsou irelevantní, zanechte příslušné buňku tabulky prázdnou. Názvy jednotlivých TC volíme ve sloupci TC ID# tak, aby odpovídaly testovanému scénáři. V našem případě jde o evidenci faktury, zvolil jsem tedy zkratku IE (Invoice Evidence). V případě velkých systémů nám toto pojmenování pomůže v orientaci.

TC ID#	Scénář	číslo faktury	částka	datum splatnosti	Očekávaný výsledek
IE 1	Scénář 1 – založení faktury	V	V	V	Faktura úspěšně založena
IE 2	Scénář 2 – faktura tohoto čísla již v systému existuje	I			Faktura nezaložena, uživatel informován.
IE 3	Scénář 3 – částka vyšší než 100 tisíc	V	I		Faktura nezaložena, uživatel informován.
IE 4	Scénář 4 – datum splatnosti v minulosti	V	V	I	Faktura nezaložena, uživatel informován.

Nyní máme tabulku validních a nevalidních vstupů. Víme, na kterých vstupech záleží úspěšnost dané operace. Připravíme si testovací prostředí tak, abychom mohli splnit tyto scénáře. Dost často mají UC tzv. preconditions. Tyto precondititons například říkají, že uživatel, který chce založit jiného uživatele, musí mít administrátorská práva. Musíte tedy do systému před začátkem testovaní založit uživatele s administrátorskými právy. Na základě takto připraveného testovacího prostředí jste již schopni definovat konkrétní tabulku s konkrétními vstupy. V našem případě vypadá například takto. Pokud používáte excel, doplňte si tabulku o sloupce precondition a postconditions. Tam definujte, co je zapotřebí udělat před spuštěním testu a zároveň co má tester udělat po ukončení testu. (například po sobě v aplikaci uklidit).

TC ID#	Scénář	číslo faktury	částka	datum splatnosti	Očekávaný výsledek
IE 1	Scénář 1 – založení faktury	20	45 000	15.1.2010	Faktura úspěšně založena
IE 2	Scénář 2 – faktura tohoto čísla již v systému existuje	20			Faktura nezaložena, uživatel informován.
IE 3	Scénář 3 – částka vyšší než 100 tisíc	21	150 000		Faktura nezaložena, uživatel informován.
IE 4	Scénář 4 – datum splatnosti v minulosti	22	23 000	11.9.2001	Faktura nezaložena, uživatel informován.

Dost často na svých školeních se setkávám s názorem, že tento postup je příliš přebyrokratizovaný a prý zdržuje. Praxe ovšem ukazuje, že problémem není vyplnění tabulky, ale neochota některých (test) analytiků se dopředu zamyslet a definovat tyto scénáře a vstupy. Pokud se ovšem nechcete smířit s intuitivním testováním a postoupit například až k objektivnímu měření kvality, zavedení podobných pravidel vás nejspíše nemine. Pozorný čtenář teď zareaguje: Vždyť v těch TC nemám přeci kroky, jak mám postupovat? Tehdy ale nehledáme artefakt TC ale Test Script. Čili postup, kterým bude tester postupovat při ověřování konkrétního testovaného případu. Test Scripty psát je časově náročné a mnohdy se musí s měnící aplikací znovu přepisovat. Proč je tedy píšeme? Rozdíl mezi TC a Test Script je v době jejich vzniku a v tom, kdo tyto artefakty vytváří. Zatímco k definici TC dochází na začátku projektu a provádí je test analytik (dosti často i analytik projektu), tak Test Scripty si píší sami testeři. Oblast okolo názvů, tvorby TC a organizace práce je natolik rozsáhlá, že by vydala na několik článků. Pokud máme menší projekty a sdílené role, potřebujeme relativně malý aparát. Pokud ovšem projekt přerůstá určitou hranici, je zapotřebí zvolit adekvátní metody přístupu k testování.

Testujeme s rozumem (2.) – Jak z UC získat TC

Při přípravě na testování projektu je zapotřebí vytvořit scénáře (postupy), které budou testeři procházet při testování. Scénáře by v optimálním případě měly pokrývat případy užití, pro které je aplikace vyvíjena. Máme-li rozumně zpracovanou analýzu, může být získání těchto scénářů relativně snadné.

Tyto scénáře testů nazýváme „testovanými scénáři", anglicky test case, zkratkou TC. Co je to TC? Co by měl obsahovat? TC je soubor vstupů, kroků i podmínek a očekávaných výstupů. Ale kde ho vezmeme? Každá analýza vyvíjeného systému by měla obsahovpat případy užití (use cases), dále UC. Tyto scénáře užití lze s výhodou použít pro vytvoření seznamu TC. Jak na to? Ukážeme si na následujícím příkladu.

Představte si jednoduchý systémový UC na evidenci vydané faktury v nějakém systému. Daný uživatel má právo zadávat faktury pouze do 100 tisíc.

Basic Flow

- 1. Uživatel zadá číslo faktury.
- 2. Systém provede kontrolu čísla faktury.
- 3. Uživatel vyplní fakturovanou částku.
- 4. Systém provede kontrolu výše částky.
- 5. Uživatel vyplní datum splatnosti.
- 6. Systém provede kontrolu data splatnosti.
- 7. Systém vytvoří fakturu a podá uživateli zprávu o jejím úspěšném zaevidování.

Nyní máme zpracovaný zcela jednoduchý UC, kde máme základní průchod aplikací. Bohužel svět není jednoduchý a ani náš příklad nemůže být takto jednoduchý. Je zapotřebí se zamyslet a definovat další toky, kterými může tento scénář být přerušen, či pokračovat zcela jinou cestu. Jedná se o tzv. Alternate flows.

Alternate flow 1

Faktura v systému již existuje, nelze přidat.

- Basic Flow v bodě 2. Systém podá uživateli zprávu o existenci faktury tohoto čísla a UC se tímto ukončí (faktura se nevytvoří).

Alternate flow 2

Co když částka na faktuře přesáhne hodnotu 100 tis Kč?

- Basic Flow v bodě 4. Systém podá uživateli zprávu o převýšení maximální hodnoty faktury a UC se tímto ukončí (faktura se nevytvoří).

Alternate flow 3

Co když zadáme datum splatnosti v minulosti?

- Basic Flow v bodě 6. Systém podá uživateli zprávu o špatné hodnotě data splatnosti a UC se tímto ukončí (faktura se nevytvoří).

Nyní si sepíšeme do tabulky pod sebe všechny možné scénáře, a pojmenujeme si je. Pojmenování volte rozumně, neboť by se mělo používat skrze celé testování.

Sestavení a pojmenování scénářů

Scénář 1 – založení faktury	Basic Flow
Scénář 2 – faktura tohoto čísla již v systému existuje	Basic FlowAlternate flow 1
Scénář 3 – částka vyšší než 100 tisíc	Basic FlowAlternate flow 2
Scénář 4 – datum splatnosti v minulosti	Basic FlowAlternate flow 3

Teď již máme identifikovány hlavní scénáře, které budeme potřebovat otestovat. Pro vlastní testování však toto ještě nestačí. Musíme definovat vstupy a očekávané výstupy jednotlivých TC. Vstupy lze identifikovat dle rozhodovacích bloků v UC. Každé alternative flow by mělo být způsobeno nějakou příčinou, vstupem. V našem případě máme situaci poněkud zjednodušenou. V reálných systémech jsou i desítky vstupů. Zde lze s výhodou použít excel, či vhodný TC management nástroj. V našem případě dochází k startu alternate flow 3 v scénáři č.4: Datum splatnosti v minulosti. Je tedy evidentní, že v tomto scénáři se rozhodne o provedení alternativního scénáře v bodě 6. Tudíž je zapotřebí zadat číslo faktury, částku a datum splatnosti. Číslo faktury a částka je validní a datum splatnosti je neplatné (Invalid). Další vstupy jsou již pro tento scénář irelevantní a v tabulce se neobjeví. Nyní vyplňme tabulku. „V" představuje potřebný validní vstup a „I" invalidní vstup. Pokud na vstupech nezáleží nebo jsou irelevantní, zanechte příslušné buňku tabulky prázdnou. Názvy jednotlivých TC volíme ve sloupci TC ID# tak, aby odpovídaly testovanému scénáři. V našem případě jde o evidenci faktury, zvolil jsem tedy zkratku IE (Invoice Evidence). V případě velkých systémů nám toto pojmenování pomůže v orientaci.

TC ID#	Scénář	číslo faktury	částka	datum splatnosti	Očekávaný výsledek
IE 1	Scénář 1 – založení faktury	V	V	V	Faktura úspěšně založena
IE 2	Scénář 2 – faktura tohoto čísla již v systému existuje	I			Faktura nezaložena, uživatel informován.
IE 3	Scénář 3 – částka vyšší než 100 tisíc	V	I		Faktura nezaložena, uživatel informován.
IE 4	Scénář 4 – datum splatnosti v minulosti	V	V	I	Faktura nezaložena, uživatel informován.

Nyní máme tabulku validních a nevalidních vstupů. Víme, na kterých vstupech záleží úspěšnost dané operace. Připravíme si testovací prostředí tak, abychom mohli splnit tyto scénáře. Dost často mají UC tzv. preconditions. Tyto precondititons například říkají, že uživatel, který chce založit jiného uživatele, musí mít administrátorská práva. Musíte tedy do systému před začátkem testovaní založit uživatele s administrátorskými právy. Na základě takto připraveného testovacího prostředí jste již schopni definovat konkrétní tabulku s konkrétními vstupy. V našem případě vypadá například takto. Pokud používáte excel, doplňte si tabulku o sloupce precondition a postconditions. Tam definujte, co je zapotřebí udělat před spuštěním testu a zároveň co má tester udělat po ukončení testu. (například po sobě v aplikaci uklidit).

TC ID#	Scénář	číslo faktury	částka	datum splatnosti	Očekávaný výsledek
IE 1	Scénář 1 – založení faktury	20	45 000	15.1.2010	Faktura úspěšně založena
IE 2	Scénář 2 – faktura tohoto čísla již v systému existuje	20			Faktura nezaložena, uživatel informován.
IE 3	Scénář 3 – částka vyšší než 100 tisíc	21	150 000		Faktura nezaložena, uživatel informován.
IE 4	Scénář 4 – datum splatnosti v minulosti	22	23 000	11.9.2001	Faktura nezaložena, uživatel informován.

Dost často na svých školeních se setkávám s názorem, že tento postup je příliš přebyrokratizovaný a prý zdržuje. Praxe ovšem ukazuje, že problémem není vyplnění tabulky, ale neochota některých (test) analytiků se dopředu zamyslet a definovat tyto scénáře a vstupy. Pokud se ovšem nechcete smířit s intuitivním testováním a postoupit například až k objektivnímu měření kvality, zavedení podobných pravidel vás nejspíše nemine. Pozorný čtenář teď zareaguje: Vždyť v těch TC nemám přeci kroky, jak mám postupovat? Tehdy ale nehledáme artefakt TC ale Test Script. Čili postup, kterým bude tester postupovat při ověřování konkrétního testovaného případu. Test Scripty psát je časově náročné a mnohdy se musí s měnící aplikací znovu přepisovat. Proč je tedy píšeme? Rozdíl mezi TC a Test Script je v době jejich vzniku a v tom, kdo tyto artefakty vytváří. Zatímco k definici TC dochází na začátku projektu a provádí je test analytik (dosti často i analytik projektu), tak Test Scripty si píší sami testeři. Oblast

okolo názvů, tvorby TC a organizace práce je natolik rozsáhlá, že by vydala na několik článků. Pokud máme menší projekty a sdílené role, potřebujeme relativně malý aparát. Pokud ovšem projekt přerůstá určitou hranici, je zapotřebí zvolit adekvátní metody přístupu k testování.

Testovací případ

V tomto příspěvku se dozvíte, co je to testovací případ, jaké typy testovacích případů máme a jaké informace by měl testovací případ obsahovat.

Testovací případ (test case) definuje postup, jak otestovat daný požadavek. Pokud hovoříme o požadavcích, máme na mysli funkční a nefunkční požadavky uváděné většinou v SRS (Software Requirements Specification) dokumentu. Testovací případ popisuje, jak otestovat požadovanou funkčnost, to znamená, co má být zadáno na vstupu a co lze očekávat na výstupu. Testovací případy si můžeme rozdělit na logické a fyzické.

- Logický testovací případ (logical test case) je abstraktní popis toho, co se má otestovat, definuje obor a množinu hodnot.
- Fyzický testovací případ (physical test case) je konkrétní popis toho, co se má otestovat, definuje jaké hodnoty se mají zadat.

Doporučuji vždy začít návrhem logického testovacího případu a poté k němu vytvořit odpovídající fyzické testovací případy (obvykle k jednomu logickému případu existuje několik fyzických). Každý testovací případ by měl:

- mít přidělen jednoznačný identifikátor,
- mít stanovenu prioritu,
- obsahovat stručný popis předmětu testu,
- uvádět, jaké hodnoty se mají zadat na vstupu,
- uvádět, jaká je očekávaná doba zpracování,
- uvádět, jaké hodnoty je možno očekávat na výstupu,
- mít předepsáno na jaké HW a SW konfiguraci se má test provést,
- uvádět, jaký test má tomuto testu předcházet.

Nyní si výše uvedené požadavky na ideální testovací případ stručně popišme.

Jednoznačná Identifikace testovacího případu

Aby bylo možné jednotlivé testovací případy vyhodnotit a případně zopakovat, musí mít každý testovací případ přidělen jednoznačný identifikátor. Nezapomínejte na to, že stejný test může být prováděn více testery. To je další důvod, proč je jednoznačná identifikace nutná.

Priorita testovacího případu

Pokud nebyla ve fázi requirements development podceněna prioritizace požadavků, stačí již jen přihlédnout k tomu, jaký dopad by měla chyba na business zákazníka a podle toho výslednou prioritu testovacího případu stanovit.

Popis testovacího případu

Doporučuji u každého testovacího případu uvádět kromě stručného popisu i odkaz na dokumentaci, kde je uveden detailní popis vlastnosti, která má být testována. Jedině tak lze zajistit, že předmětné vlastnosti budou pokryty odpovídajícími testovacími případy. Další důvod je ten, že lidé obvykle dosahují lepších výsledků, pokud rozumí tomu, co vlastně dělají.

Vstup

Vstupem je obvykle hodnota zadaná z klávesnice, vybrána myší nebo načtená ze souboru. Doporučuji věnovat zvýšenou pozornost datovému typu, jeho syntaxi, délce a hraničním hodnotám. Testujte také, jak se aplikace zachová a jaký je výsledek operace pokud zadáte číslo mimo rozsah, desetinnou čárku místo tečky a naopak, číslo se zápornou hodnotou, matematický výraz nebo textový řetězec tam, kde aplikace očekávala číslo.

Zpracování

Mezi vstupem a výstupem obvykle dochází k nějakému zpracování. U některých typů testů nás zajímá i doba zpracování, v takovém případě by měla být uvedena konkrétní hodnota, které se má dosáhnout. Pokud je v SRS dokumentu požadována okamžitá odezva, jedná se sice o nefunkční požadavek, ale špatně definovaný, protože každý z nás může mít o okamžité odezvě zcela jinou představu. Takový požadavek by měl být vrácen a upřesněn.

Výstup

Výstup je obvykle zobrazen na obrazovce nebo zapsán do souboru. Pokud máme výstup porovnávat s očekávanou hodnotou, musí být očekávaná hodnota nebo alespoň její rozsah uveden.

HW a SW konfigurace

Pokud má mít testování nějakou vypovídající hodnotu, musí být definováno na jaké HW a SW konfiguraci se mají jednotlivé testovací případy provádět. Prostředí, kde je daná aplikace hostována, je většinou dostatečně popsáno, problém však často bývá s popisem prostředí, ze kterého se testy spouští. Pokud budeme uvažovat webovou aplikaci, ke které se přistupuje prostřednictvím tenkého klienta (browseru), měli bychom popsat nejen typ a verzi OS, ale i typ prohlížeče a na jakých verzích se má testování provádět. Pokud hovořím o SW konfiguraci, mám na mysli například i to, že výsledky testování a tedy i vlastní funkčnost daného SW mohou ovlivnit i zdánlivě takové maličkosti, jako je zapnutí nebo vypnutí java skriptu, ActiveX komponenty či cachování. I tuto skutečnost je nutné vzít při návrhu testů v potaz. Největším problémem samotného testování tak může být rozhodnutí o tom, na jakých HW a SW konfiguracích by se mělo testování provádět. Ale i odpověď na tuto otázku bychom měli nalézt v SRS dokumentu. Je zřejmé, že není možné aplikaci otestovat na všech možných konfiguracích a je nutné se smířit s tím, že na některých konfiguracích daná aplikace prostě fungovat nebude. Z tohoto důvodu by vždy mělo být uvedeno, na jakých HW a SW konfiguracích bylo testování provedeno.

Předcházející test

V některých případech nemůže test začít dříve, než jiný test skončí. Tato informace nabývá na významu obzvlášť v případech, kdy se testování účastní více testerů.

A jak stanovujete prioritu testovacího případu vy?

Testovací scénář

V tomto příspěvku se dozvíte, co je to testovací scénář a jak se testovací scénáře vytváří.

Testovacímu scénáři by měl být přidělen jednoznačný identifikátor a měl by obsahovat odkaz na Plán testování (ten by zase měl jednoznačně identifikovat testovací scénáře). Testovací scénář je tvořen sadou testovacích případů. Může se jednat o testovací případy, které na sebe navazují a musí být vykonány v přesně uvedeném pořadí. Nebo na pořadí, v jakém jsou jednotlivé testovací případy prováděny, nezáleží. Testovací scénář může mít stejnou hierarchickou strukturu jako má specifikace požadavků. Cílem test designera by mělo být pokrytí všech požadavků odpovídajícími testovacími případy. Běžně se postupuje tak, že se provede návrh testovacích scénářů, následně dojde k jejich posouzení a poté k jejich případné opravě. Testovací scénář, pokud má podobu dokumentu, mívá obvykle tuto strukturu:

- **Vlastnosti, které budou testovány** (Features to be tested) – co bude obsahem této kapitoly už asi tušíte. Všimněte si ale, že zde není kapitola Vlastnosti, které nebudou testovány (Features not to be tested). Tak je to správně, protože co se nebude testovat, jsme si již uvedli v Plánu testování a bylo by naprosto zbytečné to tady znovu opakovat.
- **Přístup k testování** (Test approach) – způsoby testování a typy testů jsme si popsali již v samostatných článcích.
- **Testovací případy** – v této kapitole se uvádí jednotlivé testovací případy a případně i testovací skripty, které budou pro dané testovací případy použity.
- **Kritéria** – (Pass/Fail criteria) – zde se obvykle uvádí kritéria, na základě kterých je možno rozhodnout o tom, zda test prošel nebo ne.

Vzhledem k tomu, že některé testovací případy mohou být použity i v jiném testovacím scénáři nebo dokonce i v rámci testování úplně jiné aplikace, pravděpodobně se v praxi s dokumentem tohoto typu nesetkáte. Je to proto, že ten kdo se testováním vážně zabývá, používá obvykle nějaký SW nástroj, který mu umožňuje jednotlivé testovací případy a scénáře efektivně spravovat a využívat. Tímto způsobem je možné dosáhnout vysoké znovupoužitelnosti (reusability) testovacích případů a snížit tak náklady na testování.

Poznámka: V tomto příspěvku se úmyslně dopouštím určité nepřesnosti vůči IEEE 829. Ten totiž pojem testovací scénář vůbec nepoužívá. Místo něj hovoří o návrhu testů. Rozdíl je v tom, že zatímco design může v angličtině zastupovat jak sloveso (navrhnout) tak i podstatné jméno (návrh), tak scenario zastupuje jen podstatné jméno – scénář. Vycházím z jednoduchého předpokladu, že test designer navrhuje, jak by měl test probíhat a vytváří tak scénář nebo chcete-li test design.

Testovací skript

V tomto krátkém příspěvku se dozvíte, co je to testovací skript a co by měl dokument popisující testovací skript obsahovat.

V okamžiku, kdy se všechny nebo některé požadavky testují automatizovaně, obvykle se k tomu používá nějaký speciální SW nebo skript. Každému takovému skriptu by měl být přidělen jednoznačný identifikátor a mělo by být popsáno, k testování jakých požadavků slouží. Dále by mělo být uvedeno:

- co je potřeba provést před vlastním spuštěním skriptu,
- jak se daný skript spouští,
- jaké se mu předávají parametry,
- kam se provádí logování,
- jak je možné běh skriptu přerušit a znovu ho spustit,
- co se má udělat poté, co skript skončí.

U popisu skriptu by měl být též uveden odkaz na testovací scénář nebo testovací případ, který ho využívá.

Poznámka: Někdy se místo pojmu testovací skript (test script) používá pojem testovací procedura (test procedure). S tímto pojmem je možné se setkat např. v dokumentech, které jsou připraveny v souladu s IEEE 829.

Testovací log

V tomto krátkém příspěvku se dozvíte, k čemu slouží Testovací log a jaké informace by měl obsahovat.

Testovací log poskytuje informace o průběhu jednotlivých testů. Testovací log by měl mít jedinečný název, aby nedošlo k záměně s logem z jiných testů, z jiného časového období nebo od jiného testera. Testovací log by měl obsahovat tyto informace:

- jednoznačný identifikátor skriptu, který byl spuštěn,
- jednoznačný identifikátor osoby, která skript spustila,
- jednoznačný identifikátor testovacího případu,
- čas zahájení, ukončení a délky trvání jednotlivých testovacích případů,
- jaká byla zadaná vstupní hodnota,
- jaká byla výstupní hodnota.

Všimněte si, že nikde neuvádím, jakým způsobem testovací log vzniká, neboť testovací log může být vytvářen buď přímo testovacím skriptem nebo samotným testerem, který ručně zaznamenává provedení jednotlivých kroků podle testovacího scénáře.

Poznámka: Ideální testovací log by měl obsahovat i informace o aktuálním stavu systému, tzn. jaké bylo zatížení CPU, využití paměti a množství I/O operací v průběhu jednotlivých testovacích případů. Vzhledem k tomu, že ne každý nástroj toto umožňuje, je možné využít vlastních prostředků operačního systému a produktů třetích stran a po dobu testování tyto parametry auditovat. Po skončení testů potom můžeme vyhodnotit, jak jednotlivé testy zatěžují systém.

Protokol o předání SW I testování

V tomto krátkém příspěvku se dozvíte, k čemu slouží Protokol o předání SW k testování (test item transmittal report) a jaké náležitosti by měl obsahovat.

Důvod existence tohoto dokumentu je prostý. Vlastní testování bude nejspíš provádět někdo zcela jiný, než kdo plán testování, testovací scénáře, testovací případy a testovací skripty navrhoval. Protokol o předání SW k testování by proto měl:

- mít přidělen jednoznačný identifikátor,
- jednoznačně identifikovat SW, který se bude testovat,
- uvádět přesné umístění SW (název serveru, IP adresa)
- obsahovat odkaz na plán testování, podle kterého se má testovat.

Nedílnou součástí tohoto protokolu je datum a podpis dvou klíčových osob a to test managera – osoby odpovědné, za správný průběh testů a development managera – osoby odpovědné za přípravu prostředí (HW a SW konfiguraci) pro testování. Development manager svým podpisem stvrzuje, že prostředí pro testování bylo připraveno a test manager zase stvrzuje, že rozumí, co je předmětem testování. Možná se vám tento způsob zdá příliš byrokratický, ale v okamžiku, kdy odpovídáte za správný průběh několika současně prováděných testů, se jedná o jedinou možnost, jak zajistit, aby se testovalo ve správném prostředí a správná verze SW.

Plán testování

V tomto příspěvku se dozvíte, co je to testovací plán, plán testování nebo chcete-li plán testů, jaká je jeho struktura a jaké informace by měl obsahovat.

Hned v úvodu bych chtěl zdůraznit, že plán testování není nic jiného, než dokument, který slouží k popsání toho, co se bude testovat a to kdy, kde, jak, kým a čím. Vzhledem k tomu, že testovacích plánů může vzniknout několik a dokonce v různých verzích, obzvlášť ve společnostech, které se vývojem a testováním SW zabývají, je nutné zajistit, aby se vždy pracovalo se správným testovacím plánem.

Každému testovacímu plánu by proto měl být přidělen jednoznačný identifikátor, který se nejčastěji sestavuje z názvu aplikace nebo modulu, který je předmětem testování a čísla vyjadřujícího verzi plánu. Jednoznačný identifikátor doporučuji uvádět nejen na hlavní stránce, ale i v záhlaví nebo zápatí každé stránky a stránky číslovat (číslo stránky / celkový počet stran). Na první stránce by měl být též uveden text „Plán testování" a poté by mělo následovat jméno osoby resp. osob, které jsou autory tohoto dokumentu. Dokument má obvykle následující strukturu:

Úvod (Introduction/Management summary)

V této kapitole je vhodné velice stručně uvést, k čemu SW, který se bude testovat, slouží, kdo ho používá a z jakých hlavních částí se skládá. Na jaké HW a SW konfiguraci systém bude běžet a na jaké HW a SW konfiguraci se bude testování provádět. Kdo, kde, kdy a jak bude testování provádět.

Dokumentace (Test items)

Zde by měly být uvedeny dokumenty, ze kterých se bude při návrhu testů vycházet. Obvykle se jedná o specifikaci softwarových požadavků (Software Requirements Specification), uživatelskou příručku (User Guide), příručku operátora (Operator Guide) a instalační příručku (Installation Guide). Pokud píšete testy ještě dřív, než se začne programovat, budete mít k dispozici jen SRS. Doporučuji jednoznačně identifikovat SW, který se má testovat, včetně čísla verze a jeho umístění. Určitě nechcete omylem otestovat starou verzi nebo provádět testy v jiném prostředí.

Vlastnosti, které budou testovány (Features to be tested)

Zde by měly být uvedeny všechny vlastnosti, které se mají testovat. Každá vlastnost by měla být popsána a měla by mít přidělen jednoznačný identifikátor.

Vlastnosti, které nebudou testovány (Features not to be tested)

Zde by měly být uvedeny vlastnosti, které se nemají testovat a proč. I zde doporučuji uvést u jednotlivých vlastností jednoznačný identifikátor. Především proto, že jedině tak lze zjistit, zda se na některé vlastnosti uvedené v SRS nezapomnělo.

Průběh testů (Approach)

Zde by měl být uveden stručný popis toho, jak jednotlivé testy budou probíhat a jaký způsob testování bude u jednotlivých typů testů použit. V této kapitole je třeba také uvést nad jakými daty a v jakém prostředí bude testování probíhat a zda se budou data obsahující např. osobní údaje nějakým způsobem modifikovat.

Přerušení testů (Suspension criteria and resumption requirements)

Zde by měly být uvedeny podmínky, za jakých je možno testy zastavit a co je nutné udělat pro to, aby se mohlo v testování pokračovat.

Výstupy z testování (Test deliverables)

Seznam dokumentů, které by měly v rámci testování vzniknout. Minimálně by se mělo jednat o dokumenty popisující: testovací případy, testovací procedury, testovací logy a závěrečnou zprávu o testování.

Harmonogram testování (Schedule)

Vzhledem k tomu, že testování je vhodné pojmout jako projekt, který se musí naplánovat a určit kdo, kdy a co má dělat, nejedná se tedy v podstatě o nic jiného, než o vytvoření WBS. Teoreticky by se sice dalo testovat do nekonečna, ale vzhledem k tomu, že disponujeme jen omezenými zdroji, není možné otestovat všechno a stejně detailně. V praxi prostě musíme něco otestovat velice důkladně, a něco jen zběžně. Většinou podle toho, které funkčnosti jsou pro zákazníka nejdůležitější. To je ostatně důvod, proč by měl být zákazník do testování zapojen.

Požadavky na zdroje (Environmental needs)

V této kapitole by měly být uvedeny veškeré požadavky na zdroje tzn. na HW a SW vybavení, nároky na prostory a jejich vybavení, přístupy do těchto prostor, přístupy do daného systému a aplikace, dodání dokumentace, testovacích dat.

Odpvědnosti (Responsibilities)

Stejně jako v jakémkoliv jiném projektu i při testování lze Identifikovat různé skupiny nebo jednotlivce, kteří mají odpovědnost za splnění jednotlivých úkolů. Typicky se jedná o osoby odpovědné za řízení celých testů, přípravu podmínek pro testování, vlastní testery, správce systému, vývojáře a v neposlední řadě i vlastníka dat. Odpovědnosti jednotlivých osob a skupin je vhodné uvést právě zde.

Personální zajištění (Staffing and training needs)

Počet členů jednotlivých skupin a požadavky na proškolení a znalosti testerů jsou obsahem této kapitoly. Je zřejmé, že čím vyšší požadavky na znalosti testerů jsou kladeny, tím méně detailní potom musí být popis testovacích případů. A naopak, čím kvalitnější popis testovacích případů máme, tím nižší jsou nároky na znalosti testerů.

Rizika (Risks and contingencies)

Zde bychom měli popsat, jaká jsou rizika. Např. že nebude včas dodána verze SW, který se má testovat, nebude připraveno pracoviště pro testery apod.

Souhlas s testováním (Approvals)

Na poslední straně dokumentu se obvykle uvádí datum a jména osob, které svým podpisem odsouhlasí daný testovací plán.

Poznámka: Názvy jednotlivých kapitol uvádím jak v češtině, tak i v angličtině. Ne vždy se jedná o doslovný překlad, srovnejte např. s IEEE 829 Standard for software and system test documentation.

Vývojové-testovací-produkční prostředí a rizika

Publikováno: 11.03.2009, Autor: Miroslav Čermák, AKTUALIZOVÁNO: 18.03.2009, Zobrazeno: 8 019x

Tento článek přináší odpověď na otázku, jak zajistit bezpečnost dat ve vývojovém, testovacím a produkčním prostředí a úspěšně zvládat rizika, která jsou s provozem těchto prostředí spojena.

Je třeba si uvědomit, že každý SW prochází zpravidla během svého vývoje několika různými prostředími. Běžně se jedná o vývojové, testovací a produkční prostředí. Vzhledem k tomu, že testovací prostředí se může ve velkých společnostech dále dělit na integrační a akceptační, jsme postaveni před poměrně nelehký úkol: zajistit zachování důvěrnosti, integrity a dostupnosti informačních aktiv během celého životního cyklu vývoje softwaru a to ve všech těchto čtyřech prostředích. Podívejme se nyní společně na největší rizika, kterým musíme čelit.

Riziko: narušení důvěrnosti

Hrozba: Únik důvěrných dat z vývojového a testovacího prostředí. Je zajímavé, že zatímco data v produkčním prostředí bývají velice často chráněna a společnosti vydávají nemalé prostředky na jejich ochranu, tak ve vývojovém a testovacím prostředí tomu tak není. Podle průzkumu, který provedla společnosti Compuware a organizace Ponemon Institute, používá většina společností při vývoji a testování aplikací skutečná data.

Opatření: Ve vývojovém a testovacím prostředím nepracovat s reálnými daty. Pokud je to možné, mělo by se provést jejich zneplatnění, např. formou kódování, scrambling, apod. V opačném případě musí být před přenesením skutečných dat do vývojového nebo testovacího prostředí zajištěn souhlas vlastníka. Vlastník by měl být s rizikem úniku důvěrných dat prokazatelně seznámen, toto riziko akceptovat a s testováním nad reálnými daty souhlasit. Podepisování nejrůznějších NDA je sice možné, ale v praxi vás to stejně před únikem důvěrných dat neochrání.

Riziko: narušení integrity

Hrozba: Neupravený konfigurační soubor může způsobit, že server z vývojového nebo testovacího prostředí bude komunikovat přímo se serverem z produkčního prostředí. Může se tak snadno stát, že např. požadavek na smazání vybraných záznamů se místo na DB serveru v testovacím prostředí provede na DB serveru v produkčním prostředí. Pokud se na tuto skutečnost nepřijde včas, může to mít pro společnost zcela katastrofální následky.

Opatření: Jako vhodné řešení se jeví oddělit od sebe vývojové, testovací a produkční prostředí a to tak, aby servery z jednoho prostředí nemohly komunikovat se servery z druhého prostředí.

Hrozba: Osoba s přístupem do produkčního i testovacího prostředí může provést úmyslný nebo úmyslný a z pohledu společnosti nežádoucí zásah do SW vybavení a dat

Opatření: Vývojářský tým by měl mít přístup pouze do vývojového a integračního prostředí, neboť nemá dělat nic jiného než unitní a integrační testy. Testovací tým by měl mít přístup zase jen do akceptačního prostředí.

Hrozba: Osoba s přístupem do více prostředí si může splést prostředí v jakém pracuje a provést nežádoucí změnu v SW vybavení a datech.

Opatření: Pokud chce společnost toto riziko co nejvíce eliminovat, je asi nejlepším řešením zablokovat uživateli po dobu testování přístup do konkrétní aplikace v produkčním prostředí a po skončení testů mu ho zase odblokovat. Vzhledem k tomu, že se každé testování velice pečlivě plánuje (předpokládám, že váš Plán testování obsahuje všechny náležitosti), tak kdo a kdy bude testovat je v dostatečném časovém předstihu známo a tak by pro vás neměl být problém účet testera do aplikace v produkčním prostředí zablokovat. Je na test managerovi, aby si ověřil a zajistil, že přístupy testerů do aplikace v testovacím prostředí jim budou aktivovány až poté, co jim bude zablokován přístup do aplikace v produkčním prostředí. Výhodou tohoto řešení je, že pracovník se nemusí nikam stěhovat a může testování provádět ze svého pracoviště. Tímto způsobem lze výrazně ušetřit, neboť není nutné budovat speciální pracoviště a vybavovat ho dalšími počítači, které by byly navíc mimo testování nevyužity.

Opatření: Přístup do testovacího prostředí umožnit jen z počítačů, které budou umístěny v chráněných a oddělených prostorách společnosti. Na takové pracoviště se bude muset pracovník před započetím testu dostavit. V praxi se bohužel ukazuje, že ani přesun pracovníka na jiný počítač není dostatečnou zárukou. Je možné si to vysvětlit tím, že většina kancelářských prostor vypadá úplně stejně, takže za chvíli pracovník už ani neví, kde že to vlastně sedí.

Opatření: Jednotlivá prostředí od sebe odlišit např. odlišnou barvou okna, změnou promptu, titulku okna, jiným pozadím pracovní plochy apod.. Bohužel, stejně jako v předchozím případě i zde se ukazuje, že toto opatření v praxi selhává, protože v okamžiku, kdy mají testeři současný přístup do více prostředí a mezi jednotlivými prostředími se přepínají, vždy se najde někdo, kdo se splete. Toto opatření lze nasadit snad jen jako doplněk k předchozím opatřením, samo o sobě riziko moc nesnižuje.

Riziko: narušení dostupnosti

Hrozba: Nežádoucí síťová komunikace iniciována servery z vývojového nebo testovacího prostředí může vést až k zahlcení produkčních serverů a síťové infrastruktury.

Opatření: Jako vhodné řešení se jeví oddělit od sebe vývojové, testovací a produkční prostředí. To lze provést např. za použití firewallů a nastavení příslušných pravidel spočívající v povolení komunikaci pouze pro konkrétní IP adresu, protokol a port.

Závěr: Je zajímavé, že ač společností investují nemalé prostředky do vybudování jednotlivých prostředí, tak jejich zabezpečení spočívající v implementaci vhodných opatření věnují minimální nebo vůbec žádnou pozornost. A jak výše uvedená rizika zvládáte vy ve vaší společnosti?

Testování SW

V tomto příspěvku se dozvíte, proč testovat SW, co je cílem testování SW, jaká je závislost mezi kvalitou SW, testováním SW a počtem chyb a kdy začít s testováním SW.

Proč testovat? Vývoj SW se stále zrychluje, požadavky na kvalitu jsou minimálně stejné, ne-li vyšší než tomu bylo před lety. Testování SW tak nabývá stále na větším významu, protože čím dříve se chyby odhalí, tím nižší jsou náklady na jejich odstranění. Testování SW se stává nedílnou součástí životního cyklu vývoje software (SDLC – Software Development Life Cycle).

Co je cílem testování? Cílem testování obvykle bývá ověřit, že SW dělá přesně to, co je uvedeno ve specifikaci a dále jak je schopen se vyrovnat s nestandardními stavy, jak reaguje na chybu uživatele nebo chybu v datech, selhání jiné SW nebo HW komponenty, jak se vypořádá se zátěží a nedostatkem systémových zdrojů, zda se dokáže zotavit po havárii, zda je odolný vůči útokům, jak funguje na různých HW a SW konfiguracích, atd.

Je třeba mít neustále na paměti, že to, že se během testování neobjevila žádná chyba a všechny testy byly úspěšné, neznamená, že SW žádné chyby neobsahuje. S velkou pravděpodobností nějaké obsahuje, ale jen se na ně nepřišlo. To je dáno tím, že SW stejně jako testy dělají lidé a lidé jak známo chybují. Odhalené množství chyb je tak značně závislé na kvalitě vývoje a kvalitě testování. V praxi tak může dojít k tomu, že:

- kvalita SW i testů je vysoká, lze předpokládat, že bude odhaleno jen malé množství chyb, ale pár jich produkt přesto může obsahovat;
- kvalita SW je vysoká, naproti tomu kvalita testů je nízká, lze předpokládat, že bude odhaleno méně chyb než v předchozím případě;
- kvalita SW i testů je nízká, lze předpokládat, že bude odhaleno jen malé množství chyb, přestože produkt jich bude obsahovat velké množství;
- kvalita SW je nízká, naproti tomu kvalita testů je vysoká, lze předpokládat, že bude odhaleno velké množství chyb.

To, že je SW nebo testování kvalitní, je často založeno jen na naší víře a přesvědčení. Můžeme se tak mylně domnívat, že nízký počet odhalených chyb je dán vysokou kvalitou SW a testování. Opak však může být pravdou, SW obsahuje plno chyb, ale z důvodu nízké kvality testování jich bylo odhaleno jen pár.

Kdy začít psát testy? Psát testy můžeme začít ještě dřív, než začneme programovat a to v okamžiku, kdy máme k dispozici SRS (Software Requirements Specification) dokument. Testy tak můžeme založit na požadavcích uvedených SRS. Vycházíme z jednoduchého předpokladu, že když se vyvíjí nějaký SW, vždy existuje nějaká specifikace toho, co by měl SW dělat. K dispozici tak většinou máme již na samém počátku vývoje seznam funkčních a nefunkčních požadavků. Na jejich základě můžeme vytvořit testovací případy a scénáře pro manuální i automatizované testy. Ostatní často uváděné dokumenty pro psaní testů jako je instalační příručka (Installation guide), příručka operátora (operator guide) a uživatelská příručka (user guide), vznikají až během vývoje nebo v jeho poslední závěrečné fázi, ale to už můžeme mít většinu testů připravených a naplánovaných.

V článku nazvaném Způsoby testování jsme si testy rozdělili podle přístupu k informacím nutným k provedení testů na white box, black box a grey box. V článku Typy testů jsme si testy rozdělili pro změnu podle V-modelu. Testy však můžeme rozdělit i podle subjektu, který je provádí. V takovém případě můžeme testy rozdělit na testy prováděné:

- dodavatelem, který obvykle provádí testy unitní, integrační a systémové a
- odběratelem, který obvykle provádí testy akceptační.

Další dělení může být podle způsobu provedení, můžeme tedy mít testy:

- automatizované – jejich výhodou je, že jsou rychlé, lze otestovat větší množství vstupních dat, lze je snadno a s minimálními náklady zopakovat, výsledky testu se zapisují do logu, některé testy se v podstatě ani jinak než automatizovaně dělat nedají (např. zátěžové testy). Jejich nevýhodu je, že je nutné je naprogramovat a to něco stojí.
- manuální testy – jejich nevýhodou je, že jsou pomalé, nelze jimi otestovat velké množství vstupních dat, každé opakování je prodražuje, vše je třeba ručně zapisovat. Na druhou stranu se nemusí nic programovat, a některé testy ani jinak než manuálně provést nelze (např. test použitelnosti).

Aby naše dělení bylo úplné, nesmíme zapomenout na:

- statické testování – prohlíží se zdrojový kód SW a hledají se v něm chyby,
- dynamické testování – chyby se hledají za běhu SW.

K testování SW je vhodné přistoupit jako k jakémukoliv jinému projektu. Celý projekt lze rozdělit do 4 hlavních částí:

- Příprava na testování – v této fázi vznikají obvykle tyto dokumenty: testovací plán, testovací scénáře a testovací případy.
- Provedení testů – v souladu s testovacím plánem pak probíhají vlastní testy podle testovacích scénářů.
- Vyhodnocení testů – pokud by se vyhodnocování neprovádělo, tak by testování ani nemělo smysl.
- Rozhodnutí o dalším postupu – zopakování některých testů apod.

V rámci testování SW lze identifikovat tyto role:

- Test designer – zodpovědný za přípravu testů, vytváří testovací scénáře, logické a fyzické testovací případy a rozhoduje o tom, jaká budou testovací data.
- Test manager – zodpovědný za plánování, organizovaní, koordinování a reportování stavu testování.
- Test executor – zodpovědný za vlastní provedení testů, dokumentuje výsledky testů, kontroluje logy a hlásí defekty.
- Defect solver – nejčastěji to bývá vývojář, kdo řeší defekty.

Dále je nutné definovat závazná pravidla pro tvorbu názvů dokumentů, které by měly v průběhu testování vzniknout. V některém z příštích článků si povíme, co je to plán testování, testovací scénář, testovací případ, testovací skript, protokol o předání SW k testování, testovací log, test incident report a test status report.

Usability test

Usability test, nebo chcete-li testování použitelnosti, patří do kategorie testů typu black box. Bohužel jen málokdo jej umí zrealizovat.

Stále se najde dost firem a manažerů, kteří jsou přesvědčeni, že provádět usability testy je naprostá ztráta času a vyhozené peníze, protože nikoliv uživatel, ale oni a jejich vývojový tým nejlépe ví, jak má uživatelské rozhraní vypadat.

V tomto příspěvku nebudu psát o tom, že i usability test musíte naplánovat, připravit, provést a vyhodnotit, to už byste měli vědět. Zaměřím se spíš na to, co je pro usability test charakteristické. **Usability test obecně by se měl zaměřit především na to, jak snadné je produkt zprovoznit a používat.** Je zřejmé, že když budete testovat webovou aplikaci, tak nemůže být o nějaké instalaci nebo konfiguraci řeč. Nezapomínejte ale, že produktem může být jakýkoliv HW a SW, a tam pak má smysl testovat i to, zda uživatel dokáže daný produkt vůbec zprovoznit. To znamená nainstalovat, zkonfigurovat a případně ho i aktualizovat, pokud produkt tuto možnost nabízí. Dost často se v případě hodnocení použitelnosti uvádí následující kritéria:

- **snadnost** – jak rychle se uživatel naučí daný produkt používat;
- **efektivita** – jak rychle dokáže s daným produktem poté, co se ho naučil používat, splnit zadané úkoly;
- **přesnost** – kolik chyb uživatel udělá, jak jsou závažné, a zda má možnost je napravit
- **zapamatovatelnost** – jak rychle si dokáže způsob ovládání vybavit, když produkt dlouho nepoužíval;
- **spokojenost** – zda uživatel s produktem spokojen, líbí se mu a bude ho rád používat.

Všechna výše uvedená kritéria mají svůj význam, a kromě posledního se dají i snadno změřit. Bohužel, zrovna to poslední kritérium je to, co nás obvykle nejvíce zajímá.

Co chcete zjistit?

Hned na úvod byste si měli stanovit, co od nové verze produktu očekáváte? Má vám pomoci udržet stávající zákazníky? Přetáhnout zákazníky konkurence? Získat úplně nové zákazníky? Jak se bude nová verze produktu líbit stávajícím klientům, anebo jak se s ní bude pracovat klientovi, který ji uvidí poprvé v životě, a který používá konkurenční produkt, ba dokonce který žádný podobný produkt doposud nepoužívá? Vidíte, hned zde máme tři kategorie uživatelů a od uživatele každé kategorie můžete očekávat naprosto rozdílné reakce.

Výběr vhodných respondentů

Vybrat vhodné testery je pro správné provedení usability testů naprosto klíčové. Je třeba si uvědomit, že osoby různého věku, pohlaví a vzdělání budou disponovat rozdílnými zkušenostmi a k řešení zadané úlohy budou přistupovat jinak. Vzhledem k tomu, že provedení usability testů je poměrně finančně a časově náročné, je vhodné vyzvat k testování typické zástupce daných skupin uživatelů.

Výběr vhodného prostředí

V materiálech, které se věnují usability testům se obvykle dočtete, že byste měli testerům vytvořit příjemné prostředí, aby se cítili během testování dobře. Toto doporučení je zavádějící. Měli byste usilovat o vytvoření takového prostředí, v jakém se bude váš produkt opravdu používat. To znamená, že pokud se předpokládá využití vašeho produktu např. v rušném prostředí nebo v přírodě, je nesmysl provádět testování v místnosti.

Sestavení úlohy

Hlavní pravidlo zní, že úloha by neměla testera v žádném případě navádět k řešení. Na řešení by měl přijít sám. Zde nevytváříme detailní testovací případy a scénáře, jako tomu je u ostatních testů, kde je to naopak nutnost. Pokud testujeme např. e-shop, tak úloha může znít velice jednoduše. Např.: „Představte s, že jedete se svou 4členou rodinou na dovolenou k vodě a chcete si koupit nafukovací člun, do kterého byste se všichni vešli. Použijte k tomu e-shop, který se nachází na adrese…" A to je vše přátelé, nic víc není třeba uvádět.

Průběh testování

Vlastní usability test spočívá v tom, že sledujeme testera, jak požadovanou úlohu řeší. Zaznamenáváme nejen to, na co uživatel v aplikaci kliká a jak rychle se dostává k cíli, ale především jeho chování a emoce. Za tímto účelem se používají kamery s mikrofony a jednosměrné zrcadlo, aby bylo možné celé uživatelovo sezení nahrát a zpětně vyhodnotit. Tester též může být vyzván, aby přemýšlel nahlas, komentoval, co dělá, co se mu líbí a co se mu naopak nelíbí, že byl překvapen, že se stalo něco úplně jiného, než očekával, že aplikace je nepřehledná, že hledá jak změnit adresu příjemce a nemůže to najít, že se mu nelíbí design, co by udělal jinak apod. Netřeba snad dodávat, že v takovém případě by již neměl být testování přítomen další tester, aby se vzájemně neovlivňovali.

Poznámka: Je nesmysl používat k usability testům vlastní zaměstnance. Jsou sice možná zainteresování na zisku, takže by mělo být v jejich zájmu, aby produkt byl co nejlepší, ale mnozí z nich se také na vývoji produktu podíleli, velice dobře ho znají,

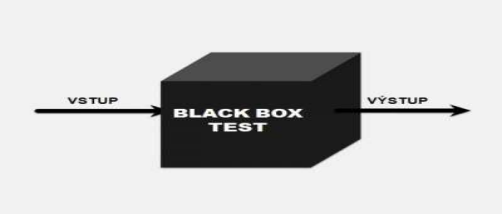
mají k němu osobní vztah a vědí naprosto přesně, jak nejrychleji danou úlohu vyřešit. Od nich nemůžete očekávat upřímnou zpětnou vazbu.

Závěr: Dost často se uvádí, že usability testy jsou finančně velice náročné, a že není možné do usability testů zapojit větší počet testerů. Je to možné, a nemusí to být ani moc drahé. Musíte však počítat s tím, že nikdo nic zadarmo testovat nebude. Testovat lze i přes internet, můžete oslovit testery, které mají web kameru s mikrofonem a požádat je, aby ji během testu aktivovali, a pak už je celkem jedno, zda tester sedí u sebe doma nebo na vašem pracovišti. Množství testerů je pak čistě dáno vaší schopností analyzovat došlé nahrávky a logy ze serveru a výší finančních prostředků, které jste na toto testování ochotni vyčlenit.

Black box test

Black box testování, známé také jako opaque box, closed box, behavioral nebo funkční je realizováno bez znalosti vnitřní datové a programové struktury.

To znamená, že tester nemá k dispozici žádnou dokumentaci, binární ani zdrojové kódy. Tento způsob testování vyžaduje testovací scénáře, které jsou buď poskytnuty testerovi nebo si je tester u některých typů testů sám vytváří. Vzhledem k tomu, že jsou obvykle definovány typy a rozsahy hodnot přípustných a nepřípustných pro danou aplikaci a tester ví, jaký zadal vstup, tak ví i jaký výstup nebo chování může od aplikace očekávat. Black box testy mohou stejně jako white box testy probíhat ručně nebo automatizovaně za použití nejrůznějších nástrojů. I v tomto případě se s oblibou využívá obou přístupů. Black box se jeví jako ideální tam, kde jsou přesně definované vstupy a rozsahy možných hodnot.



Využití

Black box testu lze využít např. na zjištění problémů typu odepření služby (DoS) nebo aktuálních zranitelností již běžícího systému nebo aplikace. V případě použití tohoto způsobu testování je třeba vždy počítat s určitým rizikem pádu daného systému. Black box testem je také možné demonstrovat chybu, která byla objevena během white box testu. K provedení testů stačí mít k dispozici jen daný program nebo znát jeho umístění, v takovém případě může být proveden test i vzdáleně po síti.

Výhody

- Snadnost – test může být proveden bez znalosti programovacích jazyků, operačních systémů.
- Rychlost – lze rychle v krátkém období otestovat i rozsáhlé systémy.
- Transparentnost – test je pro zákazníka srozumitelný – chápe, co se bude a jak testovat, může a často to bývá i on, kdo testovací scénáře vytváří a testování pak sám provádí.
- Testovací scénáře mohou být napsány v okamžiku, kdy je kompletní specifikace (některé prameny uvádí, že je možné je psát už v průběhu vzniku specifikace.)
- Testování není založeno na aktuální implementaci. I když se změní programovací jazyk, operační systém a HW, testování bude probíhat pořád stejně. Testovací scénář není nutno měnit.
- Testerovi není nutné zpřístupňovat zdrojový kód.

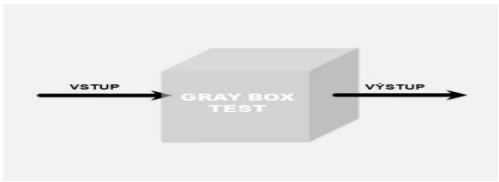
Nevýhody

- Nižší kvalita kódu – to, že se na výstupu objeví očekávaná hodnota, neznamená, že aplikace je správně napsaná. Kód může být značně neefektivní.
- Nežádoucí chování aplikace – kromě požadované funkcionality může produkt provádět i jiné akce, které nejsou ve specifikaci a jejichž výstup se na standardním výstupu neobjeví a test je proto neodhalí.

Grey box test

Grey box testování, známé též jako translucent box předpokládá omezenou znalost interních datových a programových struktur za účelem navrhnutí vhodných testovacích scénářů, které se realizují na úrovni black box.

Způsob testování je tak kombinací black box a white box testování. Nejedná se o black box, protože tester zná některé vnitřní struktury, ale zároveň se nejedná ani o white box, protože znalosti vnitřních struktur nejdou do hloubky. Koncept grey box testování je velice jednoduchý. Jestliže tester ví, jak produkt funguje uvnitř, potom ho může lépe otestovat zvenku. Gray box test, stejně jako black box test je tedy prováděn zvenku, ale tester je lépe informován, jak jednotlivé komponenty fungují a spolupracují.



Využití

Typickým příkladem je, že white box test je použit k nalezení zranitelností a black box test je následně použit ke zjištění, zda je tyto zranitelnosti možné použít k úspěšnému provedení útoku. Se znalostí vnitřních struktur může tester lépe sestavit testovací scénáře, určit hranice hodnot atd. Grey box testu se obzvlášť používá, když se provádí integrační testování dvou modulů od dvou různých dodavatelů a je potřeba otestovat interfacy. Další velice častý případ užití je v případě testování vícevrstvé aplikace, kdy máme kontrolu nad vstupem, výstupem a máme přímý přístup do databáze. Můžeme tak porovnávat všechny tři hodnoty: vstup, hodnotu v databázi a výstup. Tímto způsobem můžeme zjistit, kde dochází k manipulaci s výsledkem, zda na straně klienta, aplikace nebo při zápisu či čtení z databáze nebo přímo v databázi například triggerem, který spustí nějakou uloženou proceduru. Dále můžeme prohlížet vytvořený HTML formulář, analyzovat skripty a práci s cookie. Stejně tak je možné použít tohoto přístupu k penetračnímu testování webové aplikace. Grey box testů se také s oblibou používá v okamžiku, kdy nechceme poskytnout zdrojové ani binární kódy, a zároveň nechceme, aby tester ztrácel čas skenováním sítě a inventarizací. V takovém případě mu informace o topologii sítě a architektuře aplikace poskytneme.

Výhody

- Slučuje v sobě výhody black box i white box přístupu.
- Neintrusivní – přístup ke zdrojovému ani binárnímu kódu není třeba. Je založeno na znalosti funkční specifikace, rozhraní a architektuře aplikace.
- Inteligentní testy – tester je schopen díky znalostem, byť omezeným, napsat inteligentní testovací scénáře zaměřené i na manipulaci s daty a použité komunikační protokoly.

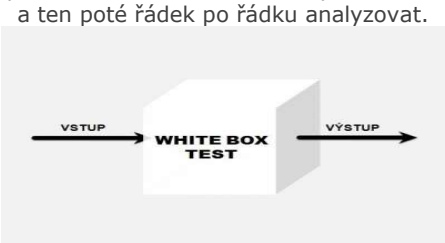
Nevýhody

- Neúplné otestování – to je dáno tím, že binární ani zdrojové kódy nejsou k dispozici a není tak možné otestovat všechny datové toky. Míra pokrytí těchto toků závisí hodně na schopnostech, znalostech a zkušenostech testera.
- Kvalita kódu – stejně jako u black box testu, to že něco funguje podle specifikace a je to odolné proti známým zranitelnostem, neznamená, že kód je efektivní a že aplikace neobsahuje žádný nežádoucí kód.

White box test

White box testování, známé též jako glass box, clear box, open box nebo také strukturální, předpokládá znalosti vnitřní struktury.

Přesněji řečeno vyžaduje znalost vnitřních datových a programových struktur a také toho, jak je systém naimplementován. Testerovi jsou v případě white box testování poskytnuty veškeré informace, to znamená, že má k dispozici nejen příslušnou dokumentaci, ale i binární a zdrojový kód testované aplikace. Tester musí zdrojovému kódu porozumět a analyzovat ho. Někdy se tomuto způsobu testování říká také audit zdrojového kódu. Zahraniční literatura má pro tuto činnost označení 'code-review' exercise. White box testování může probíhat zcela automatizovaně nebo ručně. V praxi se však velice často oba tyto způsoby vhodně kombinují. Existují v zásadě dva typy analytických nástrojů, ty které požadují zdrojový kód a ty, které jsou schopny v případě, že zdrojový kód není k dispozici, provést automatickou dekompilaci binárního kódu a zdrojový kód si de-facto vyrobit a ten poté řádek po řádku analyzovat.



Využití

White box testování je velice vhodné např. pro webové služby a to obzvlášť na počátku vývojového cyklu, kdy vývojář a tester mohou spolupracovat na odhalování chyb. White box test můžeme využít ke zjištění, jak se systém chrání před neautorizovaným přístupem, jak je řízen přístup k jednotlivým částem aplikace a k datům, jak je implementována integritní ochrana nebo jak jsou ukládána hesla. White box test můžeme také použít k nalezení nežádoucího kódu, bezpečnostních chyb a zranitelností.

Nalezení nežádoucího kódu považuji vůbec za největší přínos white box testování. V podstatě v jakékoliv aplikaci může být přítomen nežádoucí kód a aplikace přitom může pracovat zcela správně a v souladu se specifikací. Jde o to, že naprostá většina testů je zaměřena na to, aby se ověřilo, že aplikace funguje správně, resp. že dělá to co má, nebo ještě lépe, že jsou splněny všechny funkční a nefunkční požadavky uvedené ve specifikaci. A v tom je kámen úrazu. Málokdo totiž testuje, zda aplikace náhodou nedělá ještě něco navíc.

Takový nežádoucí kód může provádět v podstatě cokoliv a záleží čistě na fantazii vývojáře, co to bude a kdy se takový kód spustí. Zda bude kód provádět nějakou činnost soustavně nebo bude skrytě čekat na výskyt nějaké konkrétní události. Takovou událostí může být třeba zaslání předem definovaného typu zprávy, stisknutí určité kombinace kláves, zadání určitého jména a hesla, které je uloženo přímo v autentizačním modulu.

Výsledkem činnosti takového kódu pak může např. být získání neoprávněného přístupu do aplikace, převod finančních prostředků na jiný účet, ukládání informacích o klientech společnosti a jejich transakcích do úložiště do kterého má útočník přístup nebo zasílání vybraných informací do e-mailové schránky útočníka, pro tento účel zřízené.

V rámci objektivity je nutné podotknout, že nežádoucí kód nemusel být do aplikace začleněn jen úmyslně, ale mohl zde zůstat i proto, že vývojář si třeba nechal v rámci ladění aplikace někam zapisovat výstup a pak na to zapomněl a kód neodstranil. Čím později se na to přijde, tím větší může být problém dokázat, zda byl nežádoucí kód do aplikace začleněn úmyslně nebo jeho výskyt v aplikaci pramení z nedbalosti. Ať už je ale příčina existence nežádoucího kódu v aplikaci jakákoliv, určitě se shodneme na tom, že je nutné takovýto nežádoucí kód včas odhalit a odstranit.

Při black box nebo grey box testování by k odhalení nežádoucího kódu mohlo dojít spíš náhodou. Např. tak, že by nežádoucí kód svou činností zabíral příliš mnoho místa na disku nebo v paměti, spotřebovával větší šířku pásma, než se předpokládalo, způsoboval nárůst počtu I/O operací, zatěžoval procesor atd. Na to, že se kód nějak prozradí sám, však nelze spoléhat.

Výhody

- Včasné odhalování chyb – analýza zdrojového kódu umožní odhalit chyby, kterých se programátor dopustil ještě dřív, než je kód zkompilován.
- Odhalení nežádoucího kódu – je třeba si uvědomit, že program může kromě požadovaných operací provádět i některé další nežádoucí operace, které mohou zůstat během jiných testů zcela nepovšimnuty.

Nevýhody

- Náročnost – vyžaduje výbornou znalost cílového systému, testovacích nástrojů a programovacích jazyků.
- Vysoké náklady – požaduje specializované nástroje jako jsou analyzátory zdrojového kódu, debuggery atd.

Typy testů

Cílem tohoto článku je stručně charakterizovat základní typy testů během životního cyklu vývoje SW.

Způsoby testování byly popsány již v samostatných článcích white box test, black box test a grey box test. Vzhledem k tomu, že vývoj SW zpravidla probíhá v několika fázích, je zřejmé, že v jednotlivých fázích budeme provádět rozdílné typy testů. Jednotlivé testy můžeme rozdělit např. podle V-modelu asi takto:

1.) Unitní testy

Každý vývojář by si měl svou práci vždy sám otestovat. V rámci testování by se měl zaměřit i na to, jestli používá vhodné algoritmy, návrhové vzory, datové typy atd.

2.) Integrační testy

Cílem těchto testů je zjistit, zda spolu jednotlivé moduly spolupracují tak, jak mají.

Regression test

Ověření, zda se v důsledku změny neobjevila chyba tam, kde to předtím již bylo v pořádku.

Incremental Integration Test

Provádí se v okamžiku, kdy je ke stávajícím již otestovaným modulům přidán další.

Smoke test

Test, zda je systém připraven na hloubkový systémový test, aniž by spadnul. To je důležité, protože kdybychom tento test přeskočili, mohlo by se stát, že by v hloubkovém systémovém testu došlo k pádu a v testování by se nemohlo pokračovat.

3.) Systémové testy

Cílem těchto testů je ověřit, že produkt splňuje všechny požadavky.

Recovery test

Účelem je otestovat, jak rychle a zda vůbec se produkt vzpamatuje po pádu systému, HW chybě, výpadku proudu atd. Tento požadavek by měl být uveden v SRS dokumentu v sekci nefunkčních požadavků.

Security test

Odhalují, jak a zda vůbec se systém chrání před neautorizovaným přístupem, jak jsou ukládána hesla, jak je řízen přístup, jak je implementována integritní ochrana. Slouží k nalezení nežádoucího kódu, bezpečnostních chyb, zranitelností.

Stress test

Cílem je ověřit, zda při velké zátěži, která může být vygenerována automaticky např. provedením velkého počtu složitých dotazů, a nedostatku zdrojů, nedojde k chybě, která by se za normálního provozu neobjevila.

Performance test

Při tomto testu systém odolává velkému počtu různých požadavků a sleduje se, jaká je jeho odezva, resp. jak je ovlivněn výkon aplikace, např. jak rychle je aplikace schopna na jednotlivé typy požadavků odpovídat. Tím lze vysledovat, které části systému je třeba věnovat větší pozornost a provést v ní příslušné optimalizace (Může to být refaktorizace kódu, ale stejně tak i prosté vytvoření indexů nad tabulkou databáze.)

Installation test

Testuje se, jak probíhá instalace a odinstalce produktu na dané platformě.

4.) Akceptační testy

Alpha test

Tyto testy ve vývojovém prostředí provádí někdo jiný než vývojový tým, ten to jen sleduje a poskytuje podporu.

Beta test

Tyto testy probíhají v prostředí zákazníka a chyby jsou reportovány vývojářům dohodnutou cestou pomocí připravených formulářů.

Acceptance test

Provádí zákazník ve svém testovacím prostředí a ověřuje, zda produkt je v souladu s požadavky ve specifikaci a zda se v jeho prostředí chová tak, jak má.

Long Term Test

Tyto testy slouží k odhalení chyb, které se zpravidla vyskytnou až po delší době používání aplikace.

Poznámka: Uvedené rozdělení není jediné možné a ani se nesnaží být vyčerpávající.

Test status report

Co je to test status report? Jednoduše řečeno, jedná se o zprávu o výsledcích testování, kterou zpravidla Test manager předkládá vlastníkovi.

Stejně jako všechny ostatní dokumenty, které v rámci testování vzniknou, i tento dokument by měl obsahovat jednoznačný identifikátor. Vzhledem k tomu, že ne vždy se povede plán testování dodržet, mělo by se ve výsledné zprávě objevit, co přesně se testovalo a kdo, kdy, kde a jak testování prováděl. Jedině tak je možné splnit podmínku opakovatelnosti a měřitelnosti. Podmínka opakovatelnosti a měřitelnosti testů znamená, že jestliže bude test nad stejnými daty proveden znovu a podle stejných testovacích scénářů a za použití stejných testovacích skriptů, měli bychom dosáhnout velice podobných výsledků. Záměrně nepíši stejných, protože k jisté odchylce u určitých typů testů může dojít a nelze tuto skutečnost považovat za chybu.

V test status reportu by měl být uveden odkaz na:

- plán testování,
- testovací scénáře,
- testovací skripty,
- protokol o předání SW k testování,
- testovací logy a
- případně test incident report.

Pokud testovaný SW vykazoval nějaké odlišnosti od specifikace, mělo by se to v reportu objevit. V reportu by měly být též uvedeny odchylky od testovacího plánu, scénářů a skriptů. Stejně tak, pokud nějaké vlastnosti nebyly testovány, měl by být uveden důvod. Co je vůbec nejdůležitější a co každého zajímá asi ze všeho nejvíc, je počet a typ:

- odhalených defektů,
- opravených defektů a
- neopravených defektů.

Po seznámení se s výsledky testování je na vlastníkovi, aby rozhodl o dalším postupu, např. zda je možné aplikaci nasadit do produkčního prostředí nebo ne. Ostatně toto rozhodnutí by měl vždy učinit vlastník, neboť on nese odpovědnost.

Defect management

V tomto příspěvku se dozvíte, co je to defect management, jaký je životní cyklus defektu. Co je to test incident report a jaké informace by měl obsahovat.

Vzhledem k tomu, že ne vždy test projde, je potřeba mít definován závazný postup, jak tento stav řešit. Vznešeně se této činnosti říká defekt management (defect management). V podstatě jde o to určit, kdo, komu a jakým způsobem by měl defekty hlásit a jak by s nimi mělo být dále nakládáno. Pokud se nepoužívá na správu defektů žádný pokročilý SW nástroj, vytváří se tzv. test incident report, ve kterém by mělo být uvedeno:

- kdo test prováděl,
- jaký skript byl spuštěn,
- o jaký se jednalo testovací případ,
- jaký byl vstup,
- jaký byl očekávaný výstup a
- jaký byl skutečný výstup,
- na jaké HW a SW konfiguraci k chybě došlo a
- kdy se tak stalo.

Životní cyklus defektu je možno jednoduše popsat takto:

- Detekce – test neprošel, skutečný výsledek se liší od předpokládaného výsledku, říkáme, že tester právě objevil defekt.
- Evidence – tester, který defekt objevil, ho dohodnutým způsobem nahlásí. Nejčastěji ho zapíše do nějaké aplikace a uvede všechny náležitosti, které jsme si uvedli výše.
- Analýza – defekt koordinátor analyzuje nahlášený defekt a ověří, zda se opravdu jedná o defekt a zda již nebyl nahlášen jiným testerem a není již v systému evidován.
- Prioritizace – vlastník by měl každému defektu přidělit priority kód, který by měl vyjadřovat jeho naléhavost.
- Přiřazení – defekt koordinátor rozhodne, které vývojářské skupině nebo vývojáři daný defekt přiřadí k vyřešení.
- Potvrzení – vývojářský tým potvrdí nebo odmítne přijetí defektu k řešení. V případě, že odmítne, musí se najít správný řešitel.
- Opravení – vývojář analyzuje defekt, navrhne řešení, provede příslušnou úpravu v kódu, otestuje a předá SW k opětovnému testování.
- Ověření – testovací tým zopakuje příslušné testy a o ověří, zda se v nově dodané verzi SW defekt již neobjevuje.
- Uzavření – poté, co se testováním prokáže, že defekt byl opravdu odstraněn, je možné ho označit jako uzavřený.

Poznámka: Všimněte si prosím, že testeři chybu obvykle označují jako defekt nebo incident a vývojáři zase jako bug. Kromě toho se chyba dost často označuje také jako anomaly, error, failure, fault, problem nebo variance.

A jak říkáte chybám, na které narazíte během testování SW vy a v čem je evidujete?

Load test

Cílem tohoto testu je zjistit, jak se systém bude chovat v reálném provozu, kdy k němu bude současně přistupovat velké množství uživatelů.

Ze zkušenosti mohu říci, že na některé chyby se dá přijít až v okamžiku, kdy je k danému systému přihlášeno velké množství uživatelů, kteří současně využívají služeb, které daný systém poskytuje.

Možná si říkáte, že na provedení takového testu nic není, vždyť přeci máme nástroje, které slouží k provedení automatizovaných testů webových aplikací a vytvořit pomocí takového nástroje testovací skripty, které budou simulovat chování typických uživatelů vaší aplikace je snadné, stejně jako spustit tento skript třeba 1000krát. Ano, tímto způsobem můžeme snadno vyzkoušet, jak se aplikace bude chovat, když k ní bude připojeno současně 1000 uživatelů.

Problém je, že pokud chcete opravdu simulovat provoz v reálném prostředí, nemůže se spokojit s tím, že vytvoříte testovací prostředí, které bude odpovídat produkčnímu, a v aplikaci založíte testovací účet a pak pod ním necháte robota se 1000x přihlásit a provést předpřipravenou sadu skriptů. Tím byste jen otestovali, jak se systém chová v případě, že by si jeden jediný uživatel vytvořil 1000 sezení. A řešením dokonce není ani vytvoření 1000 různých účtů.

Takhle byste např. nikdy nezjistili, že systém začne při určitém počtu současně připojených uživatelů zobrazovat data jiného uživatele. **Vy potřebujete vytvořit nejen odpovídající množství účtů, ale zároveň musíte nastavit práva v systému tak, že každý účet bude mít přístup jen ke svým datům a ta data navíc budou jedinečná.** Nestačí tedy jednoduše zkopírovat stejná data na tisíc míst, to byste zase nic nepoznali.

Vy ta data budete muset buď vyrobit, nebo zkopírovat databázi uživatelů z produkčního prostředí do testovacího a použít reálná data. V takovém případě zvažte, zda neprovést i scrambling těch dat. Určitě ale nezapomeňte na důsledné oddělení testovacího a produkčního prostředí, protože pokud tak neučiníte, mohlo by se snadno stát, že servery z testovacího prostředí začnou komunikovat se servery z produkčního prostředí.

Tip: Od věci také není během load testu nechat ve vašem systému pracovat skutečné uživatele, neboť tím se ještě více přiblížíte reálnému provozu. No a nakonec můžete provést ještě stress test a pokud i tím aplikace úspěšně projde, máte velkou šanci, že stejně dobře si vaše aplikace povede i v okamžiku, kdy ji vystavíte do internetu. Hodně štěstí.

Jak provádíte load testing vy a jaké nástroje používáte?

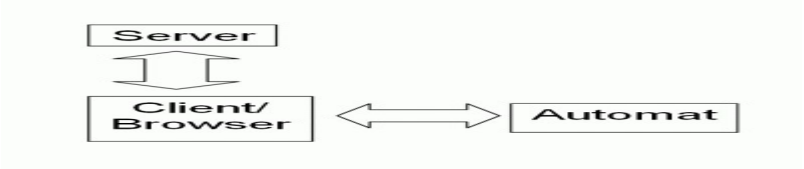
Automatizované testy aplikací typu klient-server

V tomto příspěvku se podíváme na automatizované testy aplikací typu klient-server, které mohou být v zásadě dvojího typu.

V obou případech musíme na nějakém stroji spustit robota, který bude automatizované testy provádět. Rozdíl spočívá pouze v tom, že buď bude robot simulovat uživatele dané aplikace nebo stroj, který uživatel používá.

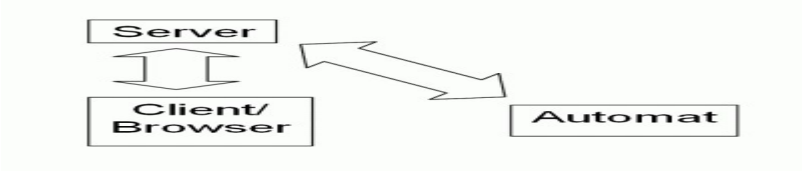
Simulace uživatele

V tomto případě bude robot provádět vybrané operace tak, jak by je prováděl uživatel systému. Výzvou, kterou musí automatizace tohoto typu řešit, je především porozumět" obsahu zobrazovaných údajů, neboť uživatel na obrazovce vidí hned, jaký je výsledek, ale automat musí nejprve správně identifikovat pole či oblast, kde se výsledek zobrazuje a následně rozpoznat výsledek, což bývá složité zejména u testování tlustých Windows klientů. Dalším problémem je krátká životnost těchto testů, neboť změna rozvržení stránky si může vynutit i přepsání kódu testu.



Simulace stroje

V tomto případě se klient úplně obchází a testuje se čistě funkcionalita serveru – volání serverových funkcí (requestů na server) se simuluje buď jako http Request, Remote Function Call, WebService Call nebo jinou další technikou, kdy užitá technika závisí na typu klienta a typu prováděného testu. Výhodou této metody je, že je odolná vůči změnám v GUI aplikace. Její nevýhodou naopak je, že se vůbec netestuje strana klienta a pokud na ní dochází k nějakým důležitým činnostem (typicky validace vstupů), zůstává tato část neotestována.



Nástroje pro automatizované testování

Na trhu je poměrně široký výběr nástrojů pro provádění automatizovaných testů ovšem s velkými rozdíly pokud jde o komfort a údržbu scénářů a rozsah podporovaných technologií.

Enterprise řešení

Před několika lety ovládaly trh specializované firmy Mercury Interactive a Rational Software. Dnes je situace jiná pouze v tom, že se obě staly součástí globálních firem HP a IBM. Vedle nich existuje řada menších firem s podílem méně než jednotky procent trhu. Software této kategorie využívají významní provozovatelé systémů jako finanční instituce, telco operátoři apod. Typická cena instalovaných licencí u těchto uživatelů je v řádu milionů korun. Jednotková cena za „seat license" čili jmenovaného uživatele jednoho produktu se uvádí kolem 8000 dolarů (150000 Kč). Cena konkurentních uživatelů bývá vyšší. TestPartner uvádí 9127 USD za „1 Concurrent User License". Ovšem pro běžné použití nikdy nestačí zakoupení jedné uživatelské licence pro jeden produkt, a navíc ani licenční podmínky to nemusí umožňovat.

HP Quality Center je webový systém pro komplexní řízení testování, původně z dílny Mercury Interactive. Využívá technologii client-server a má pět hlavních modulů Releases, Requirements, Test Plan, Test Lab and Defects Pro vlastní testování slouží dva z nich – Test Plan se používá k tvorbě a organizaci *test cases,* buď pro manuální nebo automatické testy. Test Lab je modul sloužící ke spuštění testů uložených v Test Plan. V případě manuálních testů vede testera při vyplňování výsledku testu. U automatizovaného testu ukládá výsledek testu pro srovnání s uloženým očekávaným výsledkem. **HP Quick Test Professional** je nástroj sloužící primárně pro automatizaci regresního testování funkcí, které vyžadují interakci s uživatelem. Je určen pro webové rozhraní nebo aplikace využívající MS Windows. Slouží pro zachycení uživatelových (testerových) akcí do skriptu, který pak použije HP Quality Center k provedení testu.

IBM Rational Functional Tester je nástroj s podobnými funkcemi jako HP Quick Test Professional. Pochází však z divize Rational Software. Akce uživatele-testera na testované aplikaci jsou zachyceny jako Java nebo Visual Basic.net skript. Od verze 8.1 zachycuje nástroj také obrazovky testované aplikace. To výrazně usnadňuje pozdější změny testovacího scénáře, které mohou být provedeny pomocí GUI, které tester zná, bez nutnosti měnit skript. Tester také specifikuje kontrolní body testu. V průběhu testu se při dosažení kontrolního bodu zaznamenají zvolené parametry (hodnota položky, stav objektu apod.), které pak slouží k porovnání s očekávanými údaji.

Obdobné nástroje s menším podílem na trhu jsou například **TestPartner** firmy Micro Focus nebo **SilkTest** původně od firmy Borland je dnes také součástí portfolia firmy Micro Focus, která v roce 2009 Borland koupila.

Střední kategorie

Pod úrovní produktů kategorie enterprise existuje řada firem nabízejících méně univerzální a zpravidla méně škálovatelné produkty v cenové kategorii od stovek do tisíce dolarů, které ovšem mohou adekvátně splnit účel, pro který byly vytvořeny. Uveďme dva příklady takových nástrojů.

Wapt je nástroj pro testování zátěže webových stránek a aplikací s webovým rozhraním. Produkt simuluje zátěž stovek až tisíců uživatelů na testované webové stránce, a to včetně simulace chování uživatelů na příslušných stránkách nebo aplikaci. Cenová hladina licencí je od 350 USD za licenci pro jeden počítač, s koupeným počtem licencí cena za jednotku klesá.

TestComplete je nástroj společnosti AutomatedQ pro automatizované unit, regresní, a funkční testování a pro zátěžové testování http. Nástroj je určen pro Windows a webové aplikace. Podporuje .NET, WPF, Silverlight, Ajax, Java, JavaFX, Flex a Flash; prohlížeče Internet Explorer 8 a Firefox 3.5, mobilní systémy Windows Mobile, Pocket PC, Smartphone support a OS Windows 7, Vista, XP, 2000, Windows Server 2003 and 2008. Cena rozlišuje Enterprise Edition, od 2000 USD za jednu „named user" licenci nebo 4500 USD za jednu „concurrent user" licenci. Obdobné licence pro Standard Edition jsou 1000 a 3000 USD. Standardní verze ovšem nemá schopnost testovat webové aplikace nebo http load testing.

Open source řešení

Otevřenost existující ve vývojovém prostředí webových technologií vedla ke vzniku řady open source řešení, která lze přizpůsobit a uplatnit tam, kde jsou k dispozici vývojoví pracovníci s touto kvalifikací. Mezi osvědčená řešení patří následující frameworky:

Selenium je framework pro testování webových aplikací. Poskytuje nástroj pro záznam a playback scénářů bez nutnosti znalosti skriptovacího jazyka. Součástí Selenia je doménově specifický jazyk (DSL) umožňující psát testy v prostředí Java, Ruby, Groovy, Python, PHP a Perl. Je podporován na platformách Windows, Linux a Macintosh.

JUnit je testovací framework pro jazyk Java. Hrál významnou roli při tvorbě techniky test-driven development, kdy vývoj softwaru je rozdělen na krátké vývojové cykly, z nichž každý začíná konstrukcí testovacího případu. Junit byl aplikován na další jazyky – Ada (AUnit), PHP (PHPUnit), C# (NUnit), Python (PyUnit), Fortran (fUnit), Delphi (DUnit), Free Pascal (FPCUnit), Perl (Test::Class and Test::Unit), C++ (CPPUnit), R (RUnit) a JavaScript (JSUnit).

Jmeter je projekt Apache Jakarta, který slouží pro zátěžové testování webových aplikací a služeb. Lze ho použít pro testování JDBC, FTP, LDAP, webservices, JMS, HTTP a generických TCP.

Jtest je produkt pro testování a statickou analýzu kódu v jazyku Java. Vyvíjí a podporuje jej společnost Parasoft, která nabízí několik typů licencí. Produkt slouží pro generování scénářů pro unit testy, regresní testování, statickou analýzu a code review.

Downgrade test

Myslíte si, že když úspěšně otestujete přechod na novou verzi systému a v rámci testování se neobjeví žádné problémy, že máte vyhráno?

Věřte, že problémy se obvykle objeví až po nějaké době používání a v některých případech jsou dokonce takového rázu, že není jiné cesty, než se zase vrátit k předchozí funkční verzi. Možná si teď říkáte, v čem je problém, máte přeci zpracovaný a otestovaný DRP a sekundární server pro případ výpadku. Nainstalujete na něj předchozí verzi, uživatele přesměrujete na něj a nikdo si ničeho ani nevšimne.

Testy dopadly výborně

Možná, ale co když vám řeknu, že spolu s novou verzí aplikace došlo i ke změně DB schématu? Zjišťovali jste, k jakým všem změnám s nasazením nové verze systému skutečně dochází? Obvykle se to nedělá, takže předpokládám, že jste se spokojili s tím, že funkční a performance testy dopadly výborně, systém obsahuje všechny požadované vlastnosti a je rychlejší a stabilnější než předchozí verze. Uvědomujete si ale, že když na sekundární server nainstalujete předchozí verzi aplikace, tak nebude s novou verzí DB pracovat. Dobře, tak nainstalujeme i předchozí verzi DB a data obnovíme ze zálohy, ne? Omyl, data, která jste mezitím pořídili, jsou v nové verzi systému uložena i v nové struktuře, neboť je použito nové DB schéma, takže do starého schématu je budete muset převést. Už jste to někdy dělali? Že ne, pak máte problém. Vaši DB administrátoři umí sice provést backup a restore celé DB, dokážou optimalizovat dotazy, vytvářet nad příslušnými tabulkami indexy, ale DB schéma nenavrhovali. To dělala firma, co vám daný systém dodala, takže se budete muset obrátit na ní.

Nechodí, stále nechodí

Obrátíte se tedy na firmu, a co myslíte, že se dozvíte? Nejspíš, že žádný jejich zákazník problém nehlásil, všem nová verze systému bezvadně funguje a jsou s ní naprosto spokojeni. Co jste proboha čekali, že se dozvíte? Doufám, že jste si nemysleli, že by renomovaná firma, jejíž systém získal i mnohá ocenění, vypustila do světa novou verzi systému bez důkladného otestování?

Jak je tedy ale možné, že vám ten jejich úžasný systém nefunguje? Nejspíš proto, že váš systém byl řekněme poněkud víc customizován, aby splnil vaše náročné požadavky. Mimochodem, byly to tenkrát pěkné blbosti, co jste požadovali. Firma vám sice tenkrát doporučovala provést optimalizaci procesů, ale vy jste si prosadili svojí a teď to tady máte.

Obnova provozu

Dobře, byla to tenkrát možná chyba, ale jak z té šlamastiky ven? Pro dodavatele systému by neměl být problém převést data z nového DB schématu do starého. Skript na převod ze starého do nového přeci má, jinak by nemohl proběhnout upgrade. Úpravou instalačního skriptu by tak mělo být možné provést downgrade na předchozí verzi a pokračovat na ní dál, dokud nebude problém ve vaší nové verzi odstraněn. Možná, že k chybě došlo u dodavatele systému, že opravdu neotestoval vaší custmizovanou verzi, která byla od verzí ostatních zákazníků natolik odlišná, těžko říci. Další možností je posoudit, nakolik jsou problémy v nové verzi závažné a zda není možné s nimi systém provozovat a čekat na dodání opravného patche.

Závěr: Do svého testovacího plánu byste měli zahrnout i downgrade test, nikdy nevíte, co se může stát a vy byste měli mít vždy možnost vrátit se zpět k předchozí funkční verzi.

Vícevrstvá architektura: popis vrstev

Vícevrstvá architektura se často označuje jako multi-tier nebo ještě častěji jako n-tier, kde n vyjadřuje počet vrstev, ze kterých se vícevrstvá architektura skládá.

Jako vícevrstvá se označuje proto, že funkčnost aplikace je rozdělena mezi několik vzájemně spolupracujících vrstev, které spolu komunikují přes definované rozhraní. Nejběžnějším příkladem vícevrstvé architektury je architektura třívrstvá, kterou používá mnoho webových aplikací. V takovém případě rozlišujeme vrstvu, která se stará o uživatelské rozhraní, vlastní logiku aplikace a databázi.

Tier vs. layer aneb není vrstva jako vrstva

Pokud budeme hovořit o vícevrstvé architektuře, tak dříve či později narazíme na pojmy tier a layer, které bývají dost často zaměňovány, ač je mezi nimi dost podstatný rozdíl. Vzhledem k tomu, že pojem tier a layer nelze do češtiny dost dobře přeložit, budu se raději tam, kde to bude nutné, držet anglických termínů tier a layer. Rozdíl mezi nimi je ten, že pokud se hovoří o tiers, máme na mysli fyzickou HW vrstvu, zatímco v případě layers máme na mysli logickou SW vrstvu.

Horizontální vs. vertikální přístup, aneb zvyšujeme výkon

Pokud budeme aplikaci již od počátku vyvíjet jako vícevrstvou, můžeme se kdykoliv později rozhodnout z nejrůznějších důvodů pro umístění jednotlivých logických vrstev (layers) na dedikované fyzické stroje (tiers). Jedná se o tzv. vertikální přístup (vertical approach), kdy předem vyhradíme určité stroje k provádění konkrétních úloh. Nic nám však nebrání v horizontálnímu přístupu (horizontal approach), který spočívá čistě v posílení výkonu daného tieru např. přidáním dalšího serveru do clusteru, tím může být zajištěno vyrovnávání zátěže (load balancing). Oba přístupy je samozřejmě možné kombinovat. Takový přístup se pak nazývá diagonální (diagonal approach).

Optimální počet vrstev

Nelze stanovit, jaký počet vrstev je optimální, vždy to závisí na konkrétním nasazení, potřebách uživatelů a celkové architektuře řešení. Mohou nastat případy, kdy navýšení počtu vrstev může celému řešení uškodit a naopak. Jde o to, jakým způsobem spolu jednotlivé vrstvy komunikují. Layers, které jsou navrhnuty, aby komunikovaly v rámci jednoho tier, mohou využívat tzv. chatty interface. Do doby, kdy jsou tyto layers umístěny na jednom tier (rozuměj serveru), nám to nemusí vadit, ale v okamžiku, kdy se rozhodneme je umístit na samostatné tiers (servery) by nám to začít vadit mohlo, protože by mohla výrazně vzrůst meziserverová komunikace. Z těchto důvodů by layers měly používat spíše tzv. chunky interface. Popsání rozdílu mezi těmito dvěma interfacy bych se chtěl věnovat v některém z příštích příspěvků.

Pojmenování vrstev

Počet vrstev a jejich pojmenování není jednotné. My si stručně popíšeme 5-vrstvou architekturu webové aplikace, která bude obsahovat tyto tiers: client, presentation, business, integration a enterprise. U každé vrstvy si uvedeme další možné pojmenování a účel, ke kterému slouží. Pokusím se o high level popis, protože se nechci věnovat konkrétní technologii nebo frameworku. Tam, kde to budu považovat za nutné, zmíním konkrétní produkt, aby bylo možné si představit, co se za danou vrstvou v praxi skrývá.

Client tier

Tato vrstva se někdy nazývá také jako GUI tier nebo presentation tier a komunikuje s prezentační vrstvou. Klientem může být, co se týká HW, stolní počítač nebo notebook stejně jako mobilní telefon. Snahou vývojáře je, aby jeho aplikace byla pokud možno nezávislá na platformě, na které bude spuštěna. Běžně rozlišujeme tři typy klientů: tenký (thin/slim/lean), tlustý (thick/fat) a chytrý (smart/rich). Tenký klient je obvykle browser, do kterého je aplikace po zadání URL stažena ve formě (X)HTML stránky, kdy vzhled bývá často upraven CSS a může a nemusí být pro její správný běh vyžadováno povolení např. JS. Tlustý klient je aplikace, která se musí na zařízení uživatele nainstalovat. Polotlustý klient se nachází někde mezi tenkým a tlustým klientem a často využívá JAVA, ActiveX, Flash, Silverlight.

Presentation tier

Tato vrstva, někdy nazývaná také jako web tier nebo presentation logic tier obsahuje obvykle dvě layers, pokud je jako architektonický vzor (architectural pattern) použit MVC (Model-View-Controller) a to Controller a View. Obvykle běží na webovém serveru, tím může být např. IIS nebo Apache. Tato vrstva je odpovědná za poskytování statického obsahu jako jsou HTML

stránky, CSS, JavaScript, obrázky, animace a video a to na základě požadavků, které přicházejí z vyšší vrstvy a komunikaci s business vrstvou. V případě požadavku na dynamicky generovaný obsah nebo službu je zaslán požadavek na aplikační server. Webový server může využívat cachování a tím počet požadavků na business vrstvu minimalizovat, to je žádoucí především proto, aby se uživateli stránky zobrazovaly co nejrychleji. Na webovém serveru se také velice často nachází serverový certifikát, kterým server prokazuje svou identitu klientovi. S webovým serverem též klient navazuje šifrované SSL/TLS spojení. Obvykle je to jediná vrstva, která má veřejnou IP adresu a je přímo dostupná z internetu.

Business tier

Tato vrstva někdy také nazývaná jako business logic tier nebo domain tier se nachází na aplikačním serveru, kterým může být např. Tomcat, GlassFish nebo Websphere. Pokud je jako architektonický vzor použit MVC, nachází se zde Model. Tato vrstva sousedí s prezentační a integrační vrstvou. Na této vrstvě se provádí kromě samotných výpočtů a vyhodnocování i autentizace (authentication), autorizace (authorization) uživatele a případně i personalizace (personalization) jeho profilu. Aplikační servery mohou být kvůli rozložení zátěže (load balancing) zapojeny v clusteru.

Integration tier

Tato vrstva, nazývaná někdy také jako data access tier sousedí s vrstvou business a enterprise. Jejím cílem je zajistit business tier přístup k datům, aniž by se musela vyšší vrstva starat o to, jaká DB byla použita. Díky této vrstvě resp. jejímu obecnému rozhraní může být databáze změněna, aniž by musela být změněna business vrstva. Ta bude s integrační vrstvou komunikovat pořád stejně a integrační vrstva zajistí, že data budou zapsána nebo načtena správně.

Enterprise tier

Tato vrstva nazývaná někdy též jako data tier nebo persistence tier, bývá často označována jako nejnižší vrstva a obsahuje data v podobě nějaké relační databáze (RDBMS), objektové databáze (ODBMS) nebo prostého souboru (file). Může se jednat o MySQL, MS SQL, Oracle, ale i textové soubory ve formátu CSV, ASC, XML. Enterprise tier přijímá požadavky od vyšší vrstvy (integration tier) na čtení, zápis nebo mazání a stará se uložení a poskytování dat.

HW a OS tier

O následujících 2 vrstvách se literatura věnující se vývoji vícevrstvých aplikací příliš často nezmiňuje. A tak není divu, že otázka výběru vhodného operačního systému a HW bývá dost často při vývoji vícevrstvé aplikace podceňována a nezřídka se tak zákazník dočká nepříjemného překvapení, protože v jeho produkčním prostředí aplikace nefunguje stejně jako v testovacím prostředí dodavatele.

OS tier

Je třeba si uvědomit, že každá vrstva musí běžet na nějakém operačním systému a ten může být dokonce na každé vrstvě v jiné verzi nebo se dokonce může jednat i OS poskytovaný různými výrobci. V takové situaci jsme potom nuceni spravovat skutečně heterogenní prostředí, ve kterém běží několik různých verzí MS Windows, Unix. Linux.

HW tier

Každý operační systém musí běžet na nějakém hardwaru a ten může být též od různých výrobců. V takovém heterogenním prostředí tak můžeme najít 32bitové i 64bitové stroje, RISC i CISC architekturu využívající symetrický i paralelní multiprocessing.

Poznámka: Webový i aplikační server mohou běžet na jednom fyzickém serveru, potom ale není možné hovořit o presentation a business tier neboť fyzické hranice mezi nimi se stírají. Nezřídka se na jednom serveru nachází presentation, business i integration vrstva, takový tier se poté označuje jako application a dostáváme se tak na 3-vrstvou architekturu. Pokud je vícevrstvá aplikace postavena nad mainframem, přidává se mezi presentation logic layer a business logic layer tzv. assembling logic layer, která se stará o navázání a udržení spojení s mainframem.

PCI DSS: konkrétní bezpečnostní opatření

V tomto příspěvku se dozvíte, co je to PCI DSS, pro koho je závazný, jaký si klade cíl a především jak chce tohoto cíle dosáhnout.

Cílem PCI DSS (The Payment Card Industry Data Security Standard) je zamezit karetním podvodům a to zavedením vhodných bezpečnostních opatření u společností, které data držitele karty zpracovávají, přenášejí nebo uchovávají. Ač PCI Security Standards Council skromně uvádí, že se jedná jen o 12 požadavků, jde ve skutečnosti o soubor 197 naprosto konkrétních bezpečnostních opatření, u kterých je navíc i uvedeno, jakým způsobem ověřit, že byly splněny.

Vzhledem k tomu, že tento obsáhlý soubor opatření je volně ke stažení na adrese www.pcisecuritystandards.org okomentuji zde stručně jen jednotlivé požadavky a do závorky uvedu, kolik opatření daný požadavek generuje.

Build and Maintain a Secure Network

- Požadavek č. 1: „Install and maintain a firewall configuration to protect cardholder data" požaduje, aby byly nainstalovány firewally, nastavena na nich odpovídající pravidla a byla vytvořena a udržována dokumentace popisující datové toky, nastavení pravidel, uvedena odpovědnost jednotlivých osob za správu firewallů a popsán proces provádění změn. Dále standard požaduje, aby kontrola nastavení firewallu byla prováděna minimálně každých 6 měsíců. (18 opatření)
- Požadavek č. 2: „Do not use vendor-supplied defaults for system passwords and other security parameters" požaduje, aby byla na všech systémech a zařízeních změněna defaultní hesla, zakázány nepoužívané služby a odinstalováno vše nepotřebné. Vpřípadě wireless zařízení by se měl místo defaultního WEP protokolu používat WPA nebo WPA2. Pokud ho zařízení nepodporuje, mělo by se upgradovat. Kromě toho by jednotlivé služby měly běžet na samostatných serverech. Ke konzoli serveru by se měl administrátor připojovat výhradně přes SSH nebo SSL. (9 opatření)

Protect Cardholder Data

- Požadavek č. 3: „Protect stored cardholder data" požaduje, aby do určeného a chráněného úložiště byly ukládány informace jen nezbytně nutné a uchovávány jen po nezbytně nutnou dobu. Měla by se používat pouze silná kryptografie, data by se měla šifrovat raději na úrovni disku než na úrovni souborového systému. Šifrovací klíče by měly být bezpečně distribuovány, ukládány, likvidovány, měly by se periodicky měnit a přístup k nim by měl být omezen. Dále je vyžadována segregace rolí a vícestupňová kontrola přístupu. Správa klíčů by měla být popsána. Pokud jde o zobrazování čísla karty, tak je povoleno zobrazovat maximálně jen prvních šest a poslední čtyři číslice. (18 opatření)
- Požadavek č. 4: „Encrypt transmission of cardholder data across open, public network", požaduje, aby byla nasazena silná kryptografie, SSL/TLS nebo IPSEC protokol a neměl by se v žádném případě používat WEP. Číslo karty by nemělo být posíláno mailem apod. (3 opatření)

Maintain a Vulnerability Management Program

- Požadavek č. 5: „Use and regularly update anti-virus software or programs" požaduje, aby byla nasazena ochrana proti malwaru a pokud možno na všech systémech a byl zajištěn automatický upgrade a vznikaly logy zaznamenávající činnost těchto antivirových prostředků. (3 opatření)

- Požadavek č. 6: „Develop and maintain secure systems and applications" – požaduje, aby byly nasazovány patche a to nejpozději do jednoho měsíce od jejich uvolnění. Společnost by si měla zajistit, že bude informována o nově objevených zranitelnostech, např. tím, že bude odebírat nějaký bulletin. Měl by být stanoven bezpečný proces vývoje SW. Mělo by se vyvíjet v souladu s metodikou OWASP a důkladně by se mělo i testovat. Mělo by být oddělené vývojové, testovací a produkční prostředí. Jedna osoba by neměla mít přístup do vývojového, testovacího a produkčního prostředí. K testování a vývoji by se neměla v žádném případě používat produkční data a před nasazením do produkce by měla být odstraněna testovací data a účty. (30 opatření)

Implement Strong Access Control Measures

- Požadavek č. 7: „Restrict access to cardholder data by business need-to-know" požaduje, aby přístup k datům byl řízen na principu need-to-know a uživatel v systému disponoval jen právy nezbytně nutnými k výkonu dané práce. (7 opatření)
- Požadavek č. 8: „Assign a unique ID to each person with computer access" požaduje, aby každé osobě bylo přiděleno jedinečné ID. Vzdálený přístup by se měl realizovat přes VPN a měla by být použita dvoufaktorová autentizace. Hesla by se měla měnit každých 90 dní, jejich minimální délka by měla být nastavena na 7 znaků a měla by obsahovat písmena a čísla. Historie hesel by měla být nastavena na hodnotu 4 a uživatel by měl mít k dispozici maximálně 6 pokusů o přihlášení, poté by mělo dojít k uzamčení účtu na 30 minut. (20 opatření)
- Požadavek č. 9: „Restrict physical access to cardholder data" požaduje, aby fyzický přístup k serverům a síťovým prvkům byl umožněn pouze vybraným osobám. Každá osoba by měla nosit viditelně visačku. Pohyb osob by měl být monitorován. Logy o návštěvnících by měly být uchovávány minimálně po dobu třech měsíců. Média by měla být bezpečně uchovávána, přepravována a skartována. (19 opatření)

Regularly Monitor and Test Networks

- Požadavek č. 10: „Track and monitor all access to network resources and cardholder data" požaduje, aby bylo dokumentováno, co se má logovat a jaké údaje má auditní záznam obsahovat. Čas na všech serverech musí být synchronizovaný. Musí být zajištěno, že auditní záznam nemůže být modifikován, proto by se měl log zálohovat na centrální server. Auditní záznamy by měly být k dispozici minimálně rok dozadu. Logy by měly být vyhodnocovány nejméně 1x denně. (22 opatření)
- Požadavek č. 11: „Regularly test security systems and processes" požaduje, aby se skenování zranitelností provádělo interně a externě a to minimálně každé 3 měsíce a po každé významné změně. Interní a externí penetrační testy by se měly provádět minimálně 1 ročně a též po každé významné změně. Funkce IDS/IPS by měla být též ověřována. (6 opatření)

Maintain an Information Security Policy

- Požadavek č. 12: „Maintain a policy that addresses information security" požaduje, aby byla vydána bezpečnostní politika, související bezpečností standardy a byli s nimi prokazatelně seznámeni všichni zaměstnanci. Ti by se měli též školit a minimálně ročně by měli být přezkušováni. (42 opatření)

Kromě těchto 12 požadavků obsahuje standard ještě přílohu, ve které jsou uvedeny další 4 opatření. Vzhledem k tomu, že 201 opatření, které by měli obchodníci a další organizace implementovat, je opravdu dost a jejich zavedení může být pro mnohé z nich časově i finančně značně náročná záležitost, uvolnil PCI Security Standards Council spreadsheet, který umožňuje s jednotlivými opatřeními lépe pracovat. Nevím, co přesně vedlo PCI SSC k tomu, že všude uvádí jen 12 požadavků, když ve skutečnosti jich je mnohem více. Můžeme se jen domnívat, že nechtěl obchodníky a další organizace hned na začátku vystrašit.

Závěr: PCI DSS přináší soubor naprosto konkrétních a rozumných bezpečnostních opatření, která jsou navíc aplikovatelná i ve společnostech, pro které není tento standard primárně určen.

Test status report

Co je to test status report? Jednoduše řečeno, jedná se o zprávu o výsledcích testování, kterou zpravidla Test manager předkládá vlastníkovi.

Stejně jako všechny ostatní dokumenty, které v rámci testování vzniknou, i tento dokument by měl obsahovat jednoznačný identifikátor. Vzhledem k tomu, že ne vždy se povede plán testování dodržet, mělo by se ve výsledné zprávě objevit, co přesně se testovalo a kdo, kdy, kde a jak testování prováděl. Jedině tak je možné splnit podmínku opakovatelnosti a měřitelnosti. Podmínka opakovatelnosti a měřitelnosti testů znamená, že jestliže bude test nad stejnými daty proveden znovu a podle stejných testovacích scénářů a za použití stejných testovacích skriptů, měli bychom dosáhnout velice podobných výsledků. Záměrně nepíši stejných, protože k jisté odchylce u určitých typů testů může dojít a nelze tuto skutečnost považovat za chybu.

V test status reportu by měl být uveden odkaz na:

- plán testování,
- testovací scénáře,
- testovací skripty,
- protokol o předání SW k testování,
- testovací logy a
- případně test incident report.

Pokud testovaný SW vykazoval nějaké odlišnosti od specifikace, mělo by se to v reportu objevit. V reportu by měly být též uvedeny odchylky od testovacího plánu, scénářů a skriptů. Stejně tak, pokud nějaké vlastnosti nebyly testovány, měl by být uveden důvod. Co je vůbec nejdůležitější a co každého zajímá asi ze všeho nejvíc, je počet a typ:

- odhalených defektů,
- opravených defektů a
- neopravených defektů.

Po seznámení se s výsledky testování je na vlastníkovi, aby rozhodl o dalším postupu, např. zda je možné aplikaci nasadit do produkčního prostředí nebo ne. Ostatně toto rozhodnutí by měl vždy učinit vlastník, neboť on nese odpovědnost.

Rizika virtualizace

O virtualizaci serverů se v poslední době dost často mluví v souvislosti se snižováním nákladů. Ostatně není se čemu divit, protože pokud má na základě doporučení výrobců většina aplikací pro svůj běh vyhrazen samostatný server, jsou potom výpočetní střediska plná jednoúčelových serverů. Taková společnost pak má velké množství serverů, jejichž výpočetní kapacitu není schopna plně využít a také provoz těchto serverů je pro ni zbytečně nákladný. U každého takového serveru totiž musí řešit minimálně otázku jejich umístění, napájení a chlazení. To představuje náklady, které rozhodně nejsou zanedbatelné. Myšlenka snížit náklady prostřednictvím virtualizace je tedy vcelku logická, neboť je založena na předpokladu, že když na jednom fyzickém serveru poběží více virtuálních serverů, které budou efektivně využívat zdroje, kterými fyzický server disponuje, stačí otázku umístění, napájení a chlazení řešit jen pro tento server.

Druhy virtualiazace

- Emulace – softwarově se emuluje hardwarová platforma, která není fyzicky dostupná. Vzhledem k tomu, že se musí kompletně emulovat hardwarová platforma, může dojít k viditelnému zpomalení systému, na kterém se emulátor spouští.
- Plná virtualizace – není potřeba provádět žádné modifikace operačního systému ani aplikací, které mají běžet ve virtuálním serveru. Hostovaný OS je důsledně oddělený od HW. Vzhledem k tomu, že je nutné emulovat řadu operací jako je přístup k paměti, většinu instrukcí CPU, přístup na disk, není možné dosáhnout plného výkonu.
- Paravirtualizace – vzhledem k tomu, že se simulace HW neprovádí, lze dosáhnout mnohem vyššího výkonu a s menší režií, je však nutné modifikovat jádro hostovaného OS.

Jak virtualizace funguje

Na fyzický server se nainstaluje nějaký virtualizační software, který je schopen přidělovat systémové prostředky jednotlivým virtuálním serverům podle potřeby a aktuálního zatížení. Základem virtualizačního SW je tzv. hypervizor, což je softwarová vrstva mezi hardwarem a virtualními servery, která se stará o přidělování prostředků jednotlivým virtuálním serverům. Důležité je, že virtuální server nikdy nepřistupuje k HW přímo, ale využívá služeb, které poskytuje hypervizor. Velikost hypervizoru se pohybuje v řádech několika málo KB až MB. Nabízí se tedy úvaha, že čím menší bude velikost hypervizoru, tím bude jednodušší a tedy i odolnější proti útokům.

Pokud se zamyslíte nad tím, jak virtuální server funguje, jistě sami přijdete na to, že asi nebude příliš dobrý nápad používat virtuální server pro takovou aplikaci, která provádí velký počet I/O operací. A to z toho důvodu, že ke sběrnici stejně jako k paměti nebo procesoru nemá váš virtuální server přímý přístup. Nezapomínejme na to, že všechny požadavky na přístup k HW musí jít vždy přes hypervizora, pokud se nejedná o virtualizaci na úrovni hardwaru nebo firmwaru, takže k určitému zpomalení, byť v mnoha případech nepodstatnému, zde přesto dochází.

Pro některé aplikace však toto zpomalení může být kritické. Nasazení do produkčního prostředí by proto měl předcházet důkladný performance test, jedině tak se lze vyhnout nepříjemnému překvapení. Po virtualizaci lze pozorovat pokles výkonu systému v řádech několika málo procent. Pozitivní ale je, že např. přenos dat mezi dvěma guesty na stejném hostu je výrazně rychlejší než mezi dvěma fyzickými systémy, protože nejste omezováni rychlostí přenosové soustavy.

Rizika virtualizace

Virtualizace bezesporu vede ke snížení nákladů, ale zároveň nějak musíme zvládat rizika, která jsme možná předtím, když každá aplikace běžela na svém vlastním fyzickém serveru, moc řešit nemuseli.

Zhoršení kvality poskytování služeb

ICT může podlehnout euforii a začít na stávajícím fyzickém serveru vytvářet další a další instance virtuálních serverů. Především proto, že vytvořit další virtuální server je mnohem jednodušší než zakoupit další fyzický server. Časem tak může dojít ke zprovoznění většího počtu virtuálních serverů než kolik jich je fyzický server schopen zvládnout a ICT nebude moci dostát svým závazkům. Na plánování nového virtuálního serveru by se proto měl podílet celý tým zahrnující správce systému, aplikace, databáze, sítě apod., tak jako jste to nejspíš dělali, když jste si pořizovali nový fyzický server.

Útok na fyzický server

Je třeba si uvědomit, že fyzický server, na kterém poběží několik virtuálních serverů, bude sice pořád čelit stejným hrozbám jako předtím, ale pravděpodobnost výskytu některých hrozeb bude spíše vyšší, než nižší. Především proto, že pro útočníka se stane tento server mnohem zajímavějším cílem. Když se mu totiž podaří tento server vyřadit z provozu, vyřadí tím z provozu i všechny virtuální servery, které na něm běží a způsobí tak mnohem větší škodu.

Útok na hypervizora

Hypervizor vytváří prostředí pro běh virtuálních strojů, izoluje je od sebe a zprostředkovává přístup k fyzickým zdrojům. Zranitelnost hypervizoru je velice nízká, stejně jako pravděpodobnost hrozby útoku. Avšak v okamžiku, kdy se útočníkovi podaří kompromitovat hypervizora, získá přístup ke všem běžícím virtuálním serverům a tedy i aplikacím a datům v nich zpracovávaných. Toto riziko je potřeba monitorovat, protože to, že zatím nebyl dokumentován úspěšný útok na hypervozora neznamená, že k němu nemůže v budoucnu někdy dojít.

Spuštění nezabezpečeného systému

Rychlost a snadnost, s jakou lze klonováním vytvářet nové virtuální stroje může vést k tomu, že dojde ke spuštění systému, který nebude obsahovat aktuální antivirové prostředky, patche a nebude splňovat bezpečnostní standardy. Takový systém pak bude zranitelný a útočník těchto zranitelností může zneužít např. k tomu, aby získal práva administrátora na tomto serveru.

Obnova neaktuální verze

Rychlost a snadnost s jakou lze provést obnovu systému v případě havárie může vést k obnově nějaké starší verze. A starší verze mohla např. obsahovat jinou business logiku. Toho si zpočátku nemusí nikdo všimnout, ale najednou se začnou objevovat staré chyby, které již byly jednou odstraněny. V okamžiku, kdy si těchto chyb někdo všimne, může již být pozdě.

Nedostatečné oddělení serverů

Virtualizace umožňuje snadno vytvořit síťová propojení mezi jednotlivými virtuálními servery, které byly dříve od sebe odděleny firewallem nebo se mezi nimi nacházel nějaký IDS/IPS.

Vyšší chybovost

Snadnost a rychlost s jakou lze provádět změny a přidávat další virtuální servery přináší i vyšší možnost zanesení chyby např. od značně přetíženého nebo nezkušeného správce.

Špatně nastavená práva

Mohou být špatně nastavena oprávnění pro manipulaci s virtuálními systémy.

Závěr: Ochraně serveru, na kterém běží mnoho virtuálních serverů, bychom měli věnovat zvýšenou pozornost a implementovat vhodná opatření vedoucí ke snížení rizik. Pokud se rozhodnete virtualizovat, volte takové řešení, které vám umožní provozovat virtualizaci nad více fyzickými stroji, především proto, aby bylo možné zakoupením a integrací dalšího fyzického stroje celkový výkon podle potřeby zvyšovat. V opačném případě budete, co se zvyšování výkonu týká, značně omezeni možnostmi stávajícího fyzického serveru. Dále doporučuji začít nejprve s virtualizací testovacího a vývojového prostředí. Poté můžete pokračovat s méně důležitými aplikacemi a teprve až když získáte dostatek zkušeností, můžete přistoupit k virtualizaci aplikací pro vás kritických.

Poznámka:Správa virtuálních serverů je časově podstatně méně náročná než správa klasických fyzických serverů. Rovněž doba pořízení a nasazení nového serveru se může zkrátit z několika měsíců nebo dnů až na několik málo hodin nebo minut. Z těchto důvodů se dá očekávat i snížení počtu zaměstnanců ICT, kteří mají správu serverů na starost.