Testování webových aplikací

Část II: Základy testování

Mgr. Anna Borovcová

Toto je pouze jedna ze čtyř částí mojí školící příručky,
celou poskytuji pouze výjimečně a to na požádání.
Připomínky a dotazy pište na anna.borovcova@gmail.com

Obsah

Část II. – Základy testování 3
1. O čem je testování 3
1.1 Definice 3
1.2 O čem je testování 3
2. Chyby 5
2.1 Motivace – některé známé chyby 8
3. Testovací tým 10
3.1 Vlastnosti testera 10
3.2 Pozice v testovacím týmu 11
3.3 Tester versus programátor 12
4. Kategorie testů 17
4.1 Statické a dynamické testování 17
4.2 Černá a bílá skříňka 17
4.3 Automatické a manuální testovaní 18
4.4 Stupně testování 18
4.5 Pokrytí testy 20
4.6 Dimenze kvality 21
5. Dokumentace 23
5.1 Nejdůležitější dokumenty podle praxe 24
5.2 Nejdůležitější dokumenty podle standardu 25
5.3 Testovací nápady 26
5.4 Reportování chyb 28
5.5 Metriky 30
6. Testovací cyklus 35
Obsah celé příručky 36
Zdroje 38

Část II. – Základy testování

1. O čem je testování

1.1 Definice

To, že se testování týká vyhledávání chyb bývá docela jasné, často se však liší chápání jeho záběru, to, jakým způsobem jsou chyby při testování vyhledávány. Některé zdroje popisují testování pouze jako dynamickou kontrolu toho, že chování programu odpovídá specifikaci [12]. Tedy k tomu, abychom aplikaci testovali, je podle této definice nutné její spuštění a existence specifikace nebo alespoň sady podmínek a pravidel, oproti kterým bychom ji kontrolovali. Širší pohled na testování nabízí Hailpern a Santhanam v [22], kteří testování definují jako jakoukoli aktivitu, která odhalí, že chování programu porušuje specifikaci; tedy do své definice počítají i takzvané statické testování, jenž nevyžaduje k vyhledávání chyb spuštění daného kódu. Mezi aktivity statického testování patří zejména různé druhy odborného pročítání kódu. Nejobecněji bývá testování považováno za samostatnou disciplínu, jejímž úkolem je ověřování kvality. V rámci tohoto přístupu bývá testování definováno vágněji, což více odpovídá tomu, jak je v praxi na testování skutečně nahlíženo. Příkladem může být definice od Kanera [11]: Softwarové testování je empirický technický výzkum kvality testovaného produktu nebo služby, prováděný za účelem poskytnutí těchto informací stakeholdrům[1].

V jakémkoli pojetí ale vždy platí, že proces testování je podmnožinou procesu ověřování a plánování kvality. Proto mohou být úkoly testovacího týmu dosti široké a na modelech životního cyklu pozorujeme, že testovací disciplína se nejen protahuje do celého vývoje, ale často zastává místo zajišťování kvality.

Co si však představit pod kvalitou produktu? Ať už definujeme kvalitu jako přínos nějaké osobě [10] nebo stupeň splnění požadavků (podle [12] jde o definici z ISO 9001:2000, Quality Management Systems — Requirements), je důležité mít na paměti, že je na ni nahlíženo vždy ve vztahu k nějaké osobě nebo spíše osobám. Produkt má splnit požadavky konkrétních osob – stakeholdrů, a tyto požadavky nemusí být vůbec zapsány nebo vyřčeny. Za vznikem softwaru stojí nějaká potřeba, tedy kvalitní software je takový, který tyto potřeby plně uspokojuje.

1.2 O čem je testování

Testování je založené na preciznosti vývojového týmu a jeho testerů, na porozumění produktu a zákazníkovi, pro kterého se produkt dělá. Tým se společně snaží vyvinout kvalitní produkt, který by zákazníkovi přinášel prospěch, protože jednou z přirozených motivací lidí je pocit z dobře vykonané práce. Takováto motivace ale nestačí, je potřeba dosažení kvality plánovat a řídit. Na začátku vývoje je třeba stanovit měřítka kvality, zvolit vhodný přístup a připravit implementaci tohoto přístupu, abychom měli jistotu, že nejen produkt, ale i proces jeho vývoje dosahuje požadované kvality. Dále se stanoví záběr testování, vybírají se testy, sbírají data a připravují nástroje, které tým k testování potřebuje. Navíc se kontroluje, zda

[1] Stakeholder – člověk, který má zájem na dané věci.

všechny požadavky na produkt jsou ve formě, aby bylo možno jednoznačně zkontrolovat jejich splnění.

Přestože tato náročná disciplína v mnoha organizacích pohltí 30-50% rozpočtu, stále dost lidí věří, že software nebývá před dodáním dobře otestován [3]. Mnoho rozhodnutí ohledně testování závisí na zkušenostech a osobnostech lidí v týmu. A testování často připomíná boj s vícehlavou saní, protože testování naráží na mnoho omezujících překážek.

První z nich popsal Dijkstra v [19] když napsal:

„…program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."

V překladu:

„…testování programu může být velmi efektivní způsob, jak ukázat přítomnost chyb, ale je beznadějně nevhodné k prokázání jejich nepřítomnosti."

Absenci chyb může ukázat jen formální důkaz, což ale kvůli své náročnosti není v praxi běžná metoda.

Mezi další problémy, se kterými je třeba počítat a vypořádat se s nimi, patří:

- Není možné otestovat kompletně všechny možné případy zadání.
- Je velmi složité odhalit, že bylo ve specifikaci na něco zapomenuto.
- Specifikace není vždy jednoznačná nebo dokonce chybí.
- Testování bývá podceňováno nebo špatně pochopeno.
- Výsledky testování mohou být ovlivněny špatnou komunikací v týmu nebo se zákazníkem.
- Testování musí být přizpůsobeno testovanému produktu, není možné stejným způsobem otestovat různý software.
- V průběhu testování se mohou stát některé testy neefektivními. Testy je nutné neustále přizpůsobovat měnícím se okolnostem.
- Dvě různé chyby se mohou navenek projevovat stejně, nebo jedna chyba se může navenek projevovat různě.
- Testování je citlivé na vstupní data, ta by měla být realistická. Vygenerování jednotvárných dat není ideální.

Samotné vyhledávání chyb k zajištění kvality produktu nestačí, přestože to není na první pohled patrné, v rámci testování je třeba udělat něco i pro to, aby chyba byla opravena. Nalezením chyby začíná další neméně důležitá práce testera a to je napsání přehledného reportu a další komunikace chyby. Pro připomenutí tohoto úkolu existuje následující věta, která je pro testery spíše mottem než definicí:

„Testing is process of searching defects in what more shortest period of time and of the fastest assurance of their correction." [11]

V překladu:

„Testování je proces hledání chyb za co nejkratší čas a s co nejrychlejším zajištěním jejich opravy."

2. Chyby

V některých firmách se vedou vášnivé diskuze o tom, jak by měly být definovány pojmy úzce související s chybou a pečlivě rozlišují mezi takovými pojmy jako je odchylka, událost, anomálie, problém, selhání a mnohé další. Vysvětlení těchto pojmů závisí na firemní kultuře a není potřeba se jimi na akademické úrovni zabývat. Naopak každý tester musí vědět, co při testování hledá a pojem chyby je klíčový.

Definice chyby by se neměla soustředit pouze na specifikaci, protože pak by nastala situace, že projekty bez specifikace by nemohly být testovány, protože by se nikdy nemohla najít žádná chyba. Také by neměla definice zapomínat na očekávání klienta, která nejsou na první pohled patrná a jejichž ohrožení by klient ani nepoznal.

Následující dvě definice předešlým podmínkám vyhovují a jsou do jisté míry ekvivalentní.

Definice od Pattona [7]:

„O softwarovou chybu se jedná, je-li splněna jedna nebo více z následujících podmínek:

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

Definice zformulovaná na základě [3]:

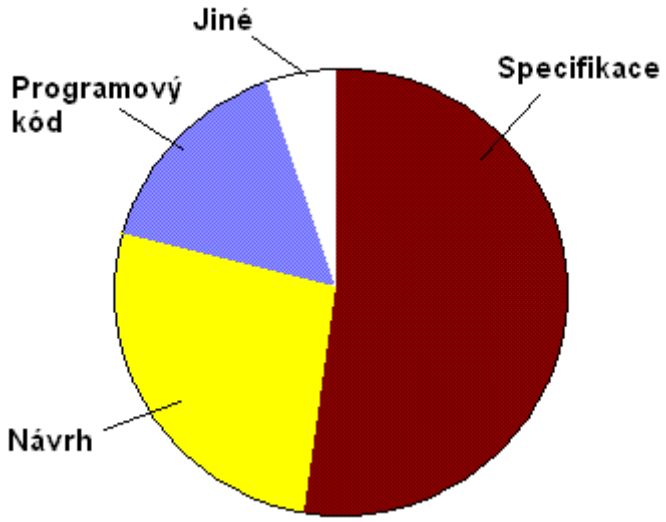
Chyba je cokoli ohledně programu, co podle některého ze stakeholdrů zbytečně snižuje hodnotu programu.

Osobně se přikláním ke kratší z definic, protože by platila zřejmě i v případě, kdyby se ukázalo, že v Pattonově definici ještě nějaký bod chybí. Navíc je pro testery lehčeji zapamatovatelná a tudíž praktičtější.

Další věc, co by měl tester o chybách znát je, kde se nejčastěji vyskytují a kde objevit ty nejdůležitější, jejichž oprava by v pozdějších fázích mohla být hodně drahá. Na taková místa je pak třeba se nejvíce zaměřit.

Řada materiálů o testování uvádí obrázek 3, který znázorňuje, že více jak polovinu všech chyb má na svědomí specifikace. Osobní zkušenosti moje a mých kolegů tomu ale neodpovídají. Je pravděpodobné, že kdysi tyto proporce původu chyb byly správné, ale s pokrokem v softwarovém inženýrství byly změněny.

Přestože ty nejhůře objevitelné nebo s největším dopadem jsou právě chyby mající původ ve specifikaci či v návrhu, nejčastější jsou chyby v programovém kódu nebo vzhledu aplikace, ty činí 50-70%. Navíc je zřejmé, že takováto charakteristika závisí na typu projektu. Vývoj řízený testy nebo schopný analytik či zkušení programátoři mohou procenta chyb podle původu značně ovlivnit.



Obr. 1: Příčiny chyb podle Pattona [10]

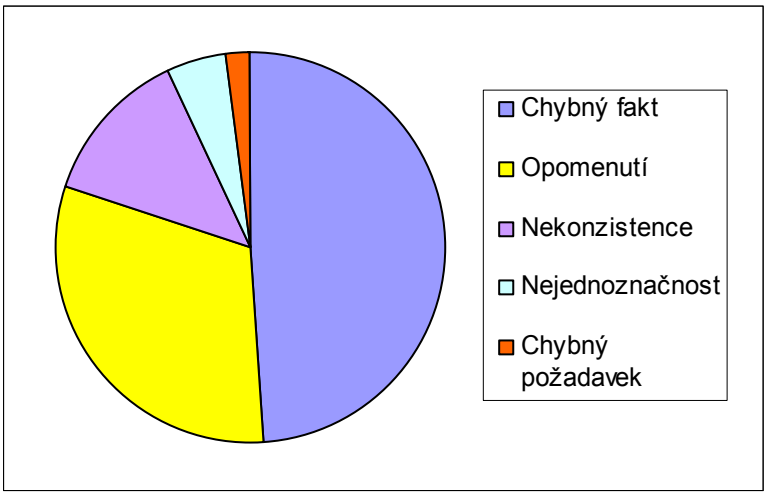
Přesto první věc, která si zaslouží značnou pozornost, je právě specifikace a návrh. Proč? V prvé řadě kvůli dopadu těchto chyb. Příliš pomalý software je jistě pro zákazníka horší, než překlep na jednom z formulářů. Dalším důvodem je, že na základě těchto dokumentů si testeři mohou zjistit něco o přáních zákazníka a seznámit se s vyvíjeným softwarem. Soustředění se na tyto části může vývoji ušetřit nepříjemná překvapení a nejvíce ovlivnit výnosnost projektu.

Pravidlo soustředění se na hledání nejzávažnějších chyb platí i obecně, a proto je třeba první testovat části, které jsou považovány z nějakého důvodu za kritické. Teprve, když ty nejdůležitější části aplikace jsou v pořádku, je vhodné se zaměřit na hledání ostatních chyb.

Kromě důležitých částí aplikace, je nutné podrobnější testování částí, u kterých existuje podezření, že nejsou úplně v pořádku. Okamžité sledování takovýchto podezření často ušetří hodiny až dny práce navíc, než kdyby byly testy zaměřeny jinam. Mezi kód, který vyžaduje takovouto pozornost, patří kód napsaný ve spěchu, nezkušeným programátorem, nebo i konkrétním programátorem, který by zkušený být měl. Většinou nejlepší informace o tom, na který kód se zaměřit, lze získat právě od vývojového týmu.

```
Říká se, že hodina pátečního přesčasu programátora stojí celé pondělí
strávené nad opravami.
```

Zajímavé, i když už ne tak užitečné, je i vědět s jakými druhy chyb se můžeme setkat, z čeho pramení. Nejčastějším neduhem je zřejmě jednoduše chybný fakt, který vyplynul z analýzy problému, ať už k tomu došlo při jakékoli činnosti. Zde například patří i řada bezpečnostních chyb. Špatně odhadneme problém. Další velmi častou chybou je opomenutí a občas ve velmi těsném závěsu, zvláště když se projekt už chýlí ke konci, jsou objevovány nekonzistence. Nejednoznačnosti a chybné požadavky naopak zabírají z celkového počtu chyb jen malou, ale i tak nepřehlédnutelnou část.

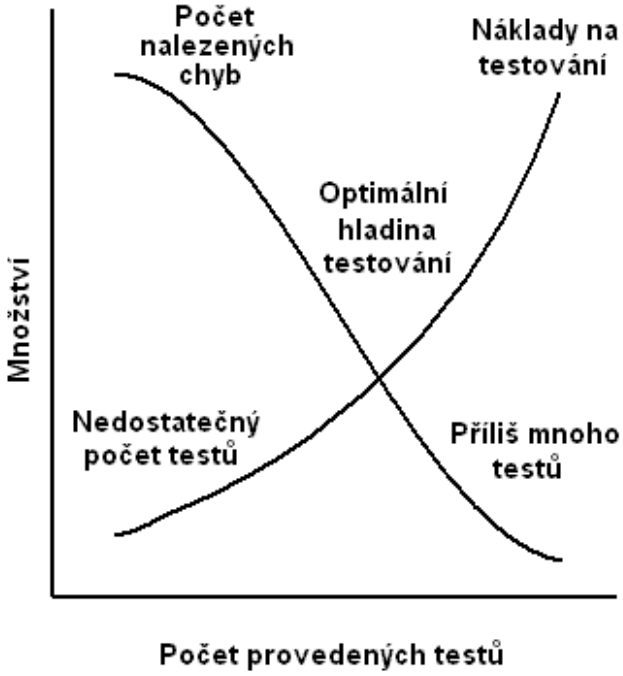


Obr. 2: Druhy chyb podle [20]

Další zajímavou charakteristikou chyb jsou náklady na jejich objevení a opravení, tedy náklady na testování. Otestovat všechny možné případy zadání, všechny situace není možné. Pro názornost stačí uvést známý příklad s kalkulačkou, kdyby šlo vyzkoušet všechny kombinace zadání: 1+0, 1+1, 1+2, 1+…?, 2+0, 2+1…?, a pak odečítání, násobení, dělení, desetinná čísla, záporná čísla, nečíselné znaky atd. atd.

Kdy se tedy vyplatí s testováním skončit? **Jelikož vývoj softwaru je ekonomická činnost, s hledáním dalších chyb by se mělo přestat v okamžiku, kdy náklady na nalezení a opravu chyby jsou v průměru stejné nebo větší než náklady na ponechání chyby v softwaru.** Ve větě jsou dva zamlčené předpoklady, první je, že jsme schopni správného vyčíslení obou nákladů, druhý, že průměrné náklady na chybu nejsou počítány ze všech nalezených chyb, ale jen z chyb nalezených dalšími testy.

Na následujícím obrázku je znázorněn zjednodušený model vztahu nákladů a chyb, jak jej zachycuje Patton v [10].



Obr. 3: Vztah nákladů a testů podle [10]

Patton ve vztahu k obr. 5 tvrdí, že každý softwarový projekt má nějakou efektivní hladinu testování. To je pravda, ale poněkud zjednodušená. Situace se zde komplikuje neuniverzálností testování. Jak už bylo řečeno, každý produkt je jedinečný a testy jsou navrhovány na základě zkušeností a preferencí toho, kdo o jejich provedení rozhoduje. Dva lidé můžou pro jeden produkt navrhnout dvě různé sady testů s různými náklady.

Podobné testy vyhledávají stejný druh chyb. V průběhu testování se tak rychle stávají neefektivními, v takovém případě při nahrazení jiným zásadně odlišným testem počet nalezených chyb náhle rapidně stoupne. Původní pravidlo od Pattona by se tedy dalo zpřesnit tak, že pro každý softwarový projekt a pro každou množinu testů existuje efektivní hladina testování.

Poměrně nízkých nákladů při otestování co největšího počtu částí aplikace a při odhalení různých druhů chyb, jinými slovy při dobrém pokrytí testy, se dosahuje rozdělením testů podle jejich zaměření a vybráním protikladných a doplňujících technik, které jsou snadno proveditelné a lze je levně aplikovat.

S náročnějšími aplikacemi a systémy se ztížilo i vyhledávání chyb. Jeden tester stejně jako jeden programátor už svými znalostmi nepokrývá všechny oblasti vývoje. Z tohoto důvodu se někteří z nich specializují více na jednu nebo několik málo oblastí. Na druhou stranu ve snaze si tuto stále komplikovanější práci zjednodušit, začala vznikat řada nástrojů pro podporu nebo přímo pro testování. Některé nástroje pro automatické testy mohou i částečně nahradit práci testera u daného druhu testu, jindy pouze automatizují jeho práci, kterou pak dokáží kdykoli sami zopakovat. Jsou to však jen nástroje a nedokáží ohodnotit většinu aspektů kvality. Nástroje pouze zlevňují a zrychlují některé testy, bez zkušených testerů se ale projekt obejde hůře než bez nástrojů.

Díky pokroku se výrazně snížila cena opravy chyby. Jednak díky návyku komentovat kód a jednodušším programovacím jazykům, jednak díky lepším překladačům a vývojovým prostředím, ale zejména díky internetu, který umožnil levné a rychlé poskytnutí updatů. Toto byl pro kvalitu velký krok vpřed. Například software, který by jinak nebyl otestován vůbec, většinou proto, že se jedná o osobní nebo školní projekt jednoho nebo více programátorů, může dosáhnout zvýšení kvality tak, že je zpřístupněn zdarma ke stažení a uživatelé jsou požádáni o zaslání nalezených chyb. Na důležitějších projektech, kde na kvalitě více záleží, je internet dalším z nástrojů, kterých testeři mohou využít ke zefektivnění a zrychlení své práce. Vrátíme-li se k větě o tom, nakolik testovat, aby se to vyplatilo, vidíme, že nyní se vyplatí dodávat kvalitnější produkty, než tomu bylo v minulosti.

2.1 Motivace – některé známé chyby

3. prosince 1999 se při pokusu modulu Mars Polar Lander o přistání na Marsu došlo ke ztrátě komunikace. Modul se už neozval. Následující vyšetřování odhalilo, že nejpravděpodobnější událostí, ke které mohlo dojít, je předčasné vypnutí motoru a následné zřícení modulu na povrch Marsu.

Původně měl modul přistát pomocí motoru a padáku, po jehož rozvinutí a asi 70 až 100 sekund před přistáním by se vysunuly přistávací nohy. Jakmile by se nohy modulu dotkly povrchu, k čemuž sloužil kontakt nastavující určitý bit, přívod paliva do motoru by byl vypnut.

Pravděpodobnou příčinou bylo předčasné nastavení bitu, kdy kontakt vyhodnotil odpor a vibrace vzduchu při přistání jako povrch Marsu. Modul testovalo několik týmů, každý se zaměřením na jinou část modulu nebo jeho mise. Zapomněli však na testování přechodů, jak předchozí část mise ovlivní následující. A tak pokud byl kritický bit předčasně nastaven,

jeden tým to ignoroval, jelikož se jich to netýkalo, a druhý začínal testy na čistém prostředí po restartu počítače a nastavení na defaultní hodnoty, takže s nenastaveným bitem [7], [45].

Z definice chyby jako čehokoli, co snižuje hodnotu systému v očích některého stakeholdra, vyplývá, že je velmi prospěšné zjistit, čím je daný systém ovlivněn nebo co vše ovlivňuje.

Další známá softwarová událost se stala 25. února 1991 v Saudské Arábii, v Dhahranu, kdy kvůli chybě v obranném raketovém systému Patriot, nedošlo k odhalení a pokusu o zneškodnění nepřátelské rakety, která následně usmrtila 28 vojáků.

Příčinou byla chyba v systémových hodinách, která způsobovala zkreslení času v závislosti na době běhu systému. Při incidentu byl systém v provozu přes 100 hodin, přičemž systémové hodiny byly vlivem chyby posunuty o třetinu sekundy. Při takto rychle pohybujícím se cíli, jako je raketa, dělala odchylka při určení pozice 600 metrů. Systém tedy raketu detekoval, ale když nic nenašel na dané pozici, byla označena jako falešný poplach a vymazána ze systému.

Chyba byla ohlášena už dva týdny před incidentem, jako prevence bylo doporučeno pravidelně rebootovat systém. Příslušní zmocněnci ale nevěděli, jak často je to pravidelně. Update byl dodán výrobcem den po incidentu [46].

Ponaučení z této situace je, za prvé, že i ta nejmenší nepřesnost za desetinnou čárkou, třeba při špatném zaokrouhlování, se může po nějaké době nakupit natolik, že její následky už nebudou tak zanedbatelné, jak se zprvu zdálo. Za druhé, uživatelé potřebují přesně vědět, co je chybou a jak chybný systém používat než dojde k opravě.

Jedny z nejzávažnějších chyb byly objeveny v softwaru zařízení Therac-25, radiologickém přístroji pro léčbu nádorů pomocí x-paprsků nebo elektronů. Následkem těchto chyb byli lidé, kteří měli tu smůlu, že při jejich léčení se projevily, vystaveni obrovským dávkám ozáření. Od června 1985 do ledna 1987 je známo šest případů, kdy byl pacient vystaven tenkému paprsku obrovské radiace, tři z nich na následky ozáření zemřeli [29].

Příčinou byla řada pochybení při vývoji, kdy produkt nepodstoupil podrobné prozkoumání. Části softwaru byly převzaty ze starší verze zařízení, přičemž byly zároveň odstraněny hardwarové pojistky proti chybám. Bezpečnost plně závisela na softwaru. Dalším pochybením byla ničím nepodložená přílišná důvěra v produkt. Po prvních incidentech zaměstnanci firmy, která Therac-25 vyvinula, tvrdili zákazníkům, že je nemožné, aby příčinou potíží byl jejich produkt. A ani poté nebyli schopni sami problém nasimulovat, dokud uživatelé neodhalili sami, jak přesně k chybě dochází.

Vnitřní příčinou těchto smrtelných chyb v softwaru pak bylo přetečení proměnné a neplánovaná závislost na načasování procesů, což je jev zvaný „race condition“.

Ze zkušeností za několik posledních let vyplývá mnoho ponaučení, nejdůležitější ale je, že každý složitější software lze přivést do stavu, kdy se bude chovat neočekávaně. Na takovéto situace je třeba se připravit, snažit se snížit nejen jejich pravděpodobnost, ale i dopad.

3. Testovací tým

3.1 Vlastnosti testera

Existuje mnoho blogů a stránek o testování, kde se můžete seznámit s tím, jak testeři uvažují. Je to zvláštní zkušenost, protože neříkají jen, dělejte to, nebo ono, v článku sice vyjádří svůj názor, ale hlavně kladou čtenářům otázky a nechávají je, ať se sami nad tím zamyslí a určí, co si z toho odnesou. Mezi dobře udělané blogy plných takovýchto článků patří např. testertested.blogspot.com.

Velká část toho být testerem je o kritickém myšlení. Pro dobré testery není nic obecně platné. Zpochybňují vše o produktu i o samotném testování. Kladou otázky a skrz ně poznávají produkt, který testují. Kladení otázek je jeden z nejlepších způsobů, jak testeři dělají svou práci.

Následující seznam obsahuje vlastnosti, kterými se dobří testeři vyznačují. Není třeba mít všechny následující vlastnosti, vynikající testeři mívají třeba jen pár z nich, ale vhodně skloubené.

Pro testera je vhodné:

- Mít odlišné myšlení od průměru. Díky tomu vymýšlejí testy, které by ostatní nenapadly, používají software neobvyklým způsobem. Sada testů, kterými tak produkt má projít, rychle roste.
- Být zvědavý, snažit se objevit skryté. Testeři se nesnaží rozbít nebo pokazit produkt, ten už pokažený je, testeři se snaží odhalit nakolik je pokažený, a ukázat ostatním, co je jim skryto, konkrétně zviditelnit chyby.
- Být pečlivý a trpělivý. Pocit z dobře odvedené práce bývá největší odměnou, protože málokdo umí poznat, zda je testování dobře odvedené nebo ne.
- Umět dobře komunikovat s okolím. Vysvětlení nalezených chyb a zajištění jejich opravy je podstatná část práce, ke které má tester ve skutečnosti jen jediný nástroj – svoji schopnost komunikovat.
- Rád se učit novému. Stát se dobrým testerem je celoživotní úkol, není možné se dostat do bodu, kdy si člověk myslí, že o testování se už nemá co učit.
- Být průbojný. Často testeři přijdou na projekt a dozví se pouhý zlomek toho, co potřebují. Neví pořádně, kde najít dokumentaci, co se ohledně projektu rozhodlo, na jaké otázky se ptát a koho. Vyžádat si půl dne času někoho zkušenějšího, kdo má hodně práce, vyžaduje odvahu. Ale bez toho není možné testovat dobře a efektivně.

Z osobní komunikace s Pradeepem Soundararajanem[2]:

„Good tester, as the world doesn't know, is someone with skills. When I say skills, it means - observation, hearing, questioning, critical thinking, analysis, general systems thinking, epistemology, self critique, techniques, approaches, knowledge, tools, brainstorming, scripting, technology, communication, interpersonal skills, management, theory..."

Přeloženo:

„Dobrý tester, jak svět neví, je někdo se schopnostmi. Když říkám schopnosti, myslím - postřeh, poslech, dotazování, kritické myšlení,

[2] Pradeep Soundararajan je výborný indický tester a autor blogu testertested.blogspot.com

analytičnost, obecné systémové myšlení, nauku o poznávání, sebekritiku, techniky, přístupy, znalosti, nástroje, brainstorming, skriptování, technologii, komunikaci, mezilidské vztahy, řízení, teorii…"

Dobrou lekcí o testování je příběh o diskuzi Bena Simo s Michaelem Boltnem[3] [38], který se odehrával zhruba takto:

MB: „Co čekáš, že dostaneš, když si objednáš pivo?"
BS: „Pivo".
MB: „Jasně. A očekáváš, že ho dostaneš co nejrychleji?"
BS: „Ano."
MB: „Očekáváš, že bude mít správnou teplotu?"
BS: „Ano."
MB: „Očekáváš, že bude mít správnou míru?"
BS: „Ano."
MB: „Očekáváš, že bude mít správnou barvu?"
BS: „Ano."
MB: „Očekáváš, že to bude stejné pivo, jako jsi si objednal?"
BS: „Ano."
MB: „Očekáváš, že bude mít správný poměr pěny?
BS: „Ano."
MB: „Očekáváš, že obsluha, která ti ho přinese, bude vhodně upravená?
BS: „Ano."
MB: „Očekáváš, že bude obsluha přívětivá?"
BS: „Ano."
MB: „Očekáváš, že bude stůl čistý a ne politý nebo ulepený?"
BS: „Ano."
MB: „Očekáváš, že v pivu nebude nic plavat?"
BS: „Ano."
MB: „Očekáváš, že sklenice na pivo bude čistá?"
BS: „Ano."
MB: „Očekáváš, že si k tomu pivu budeš moci sednout?"
BS: „Ano."
…

Ten rozhovor může pokračovat ještě hodně dlouho a nezáleží na tom, koho se ptáte, nebo jaké jsou odpovědi. Jsou to otázky, které člověk klade, které z něj dělají testera.

3.2 Pozice v testovacím týmu

Na větších projektech, na které nestačí jeden tester, se testeři sdružují do testovacích týmů, v rámci nichž mají určené své role. Spektrum těchto rolí nebo pozic je dáno firemní kulturou a používanou metodikou. Nejčastěji se objevují hierarchické názvy pozic:

- Tester (junior / senior)
- Test designér
- Test analytik
- Test manager

Při diskuzi lidí z různých firem, popřípadě dokonce z různých týmů, je vhodné si vždy nejprve ujasnit, co se pod kterou rolí – pozicí – skrývá. Bez zacházení do detailů bývají pravomoci a zodpovědnosti k pozicím přidělovány následujícím způsobem. Tester slouží jako

[3] Michael Bolton je považován za skvělého testera a experta na testování. V současné době zejména poskytuje školení a konzultace ohledně testování, přičemž ovlivňuje řadu testerů na celém světě.

obecné označení člověka, který se zabývá testováním. V hierarchii rolí v testovacím týmu ale zastává nejnižší pozici. Zpravidla pak provádí nenáročnou práci, která je buď co nejpřesněji definovaná nebo není kritická. Zkušenější testeři, kterým je už přiznána jakási zodpovědnost, bývají test senioři nebo test designéři, kteří kromě provádění testů, je i sami navrhují. Ještě důležitější pozici zastává test analytik, který už velkou měrou zodpovídá ze testování a připravuje ho. Test manager, kterého občas nahrazuje projekt manager pak testování řídí.

Kromě rozdělení rolí na základě hierarchie se testeři odlišují i svou specializací. Vybírají si skupinu druhů testů, které jsou pro ně zajímavé, a na jejich provádění se pak specializují. Mohou se zaměřit i na konkrétní druh aplikace, jazyk nebo nástroj. Takováto zaměření ale bývají méně častá, jelikož potřeba specializace vychází hlavně z nemožnosti ovládnout do hloubky všechny aspekty různých druhů testů.

3.3 Tester versus programátor

František Kunský se ve své diplomové práci Testování softwaru a testovací role [5] snaží přirovnávat testera a programátora. Přirovnávat k sobě dvě profese je velmi ošidné, to se rovněž můžeme ptát, zda je tester méně nebo více než astronom, právník nebo třeba politik. Samozřejmě zde jsou jisté schopnosti, které se vyžadují jak u programátorů, tak testerů a lidé pracující v těchto profesích spolu potřebují úzce spolupracovat. Jenže testeři toho mají hodně společného i s jinými profesemi. Kreativita a vynalézavost by měla být vlastní i umělcům a vědcům. Diplomatické jednání je vyžadováno i u právníků a politiků. Odvaha stát si za svým je ceněna napříč různými profesemi.

Zeptala jsem se proto několika expertů na testování z ciziny, jak je nahlíženo na pozici testera u nich a na jejich porovnání pozice testera a programátora zejména po stránce finanční a respektu, jelikož tyto měřítka jsou tradičně používána pro srovnávání profesí.

Tito odborníci se shodují, že testeři jsou méně respektováni a jejich práce obvykle zlehčována. Tento názor však podle nich zastávají lidé, kteří testování nerozumí a netuší, co obnáší. Práce dobrého testovacího týmu není tolik viditelná jako práce programátorů. Rovněž z průzkumu vyplývá, že představa o tom, kdo je to tester se liší mezi širší veřejností a odborníky.

Z emailu od Michaela Boltna:

„I think that testers are less respected than programmers in many places. It probably has more to do with the organization than with the country. It seems to be a world-wide thing.

Some people have suggested that being a good tester is harder than being a good programmer. The best programmers that I have ever known -- and the best testers -- are good critical thinkers. When people suggest that it's easier to be a good tester than a good programmer, I don't think that those people have thought about the issues very much.

When a tester is really good, few people tend to be aware of it. When a developer solves a big problem, many people get to see that and use the program that she wrote. When a tester discovers big problems, those problems tend to get corrected before many people see them. So the value of a good tester is not always obvious. “

Přeloženo:

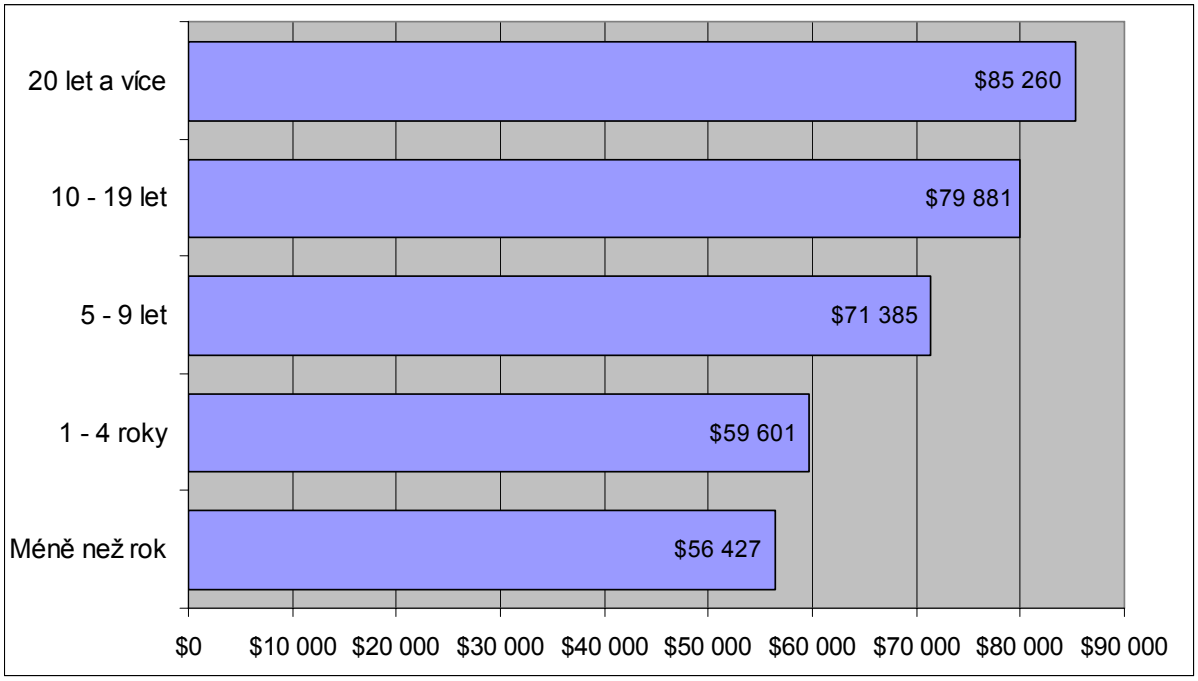
„Myslím, že testeři jsou méně respektováni než programátoři na mnoha místech. Pravděpodobně to má co do činění s organizací než se zemí. Zdá se, že je to celosvětová věc.

Někteří lidé naznačují, že být dobrým testerem je těžší než být dobrým programátorem. Nejlepší programátoři, co znám -- a nejlepší testeři -- jsou dobří v kritickém myšlení. Když lidé naznačují, že je lehčí být dobrým testerem než dobrým programátorem, nemyslím, že tito lidé o tom moc přemýšleli.

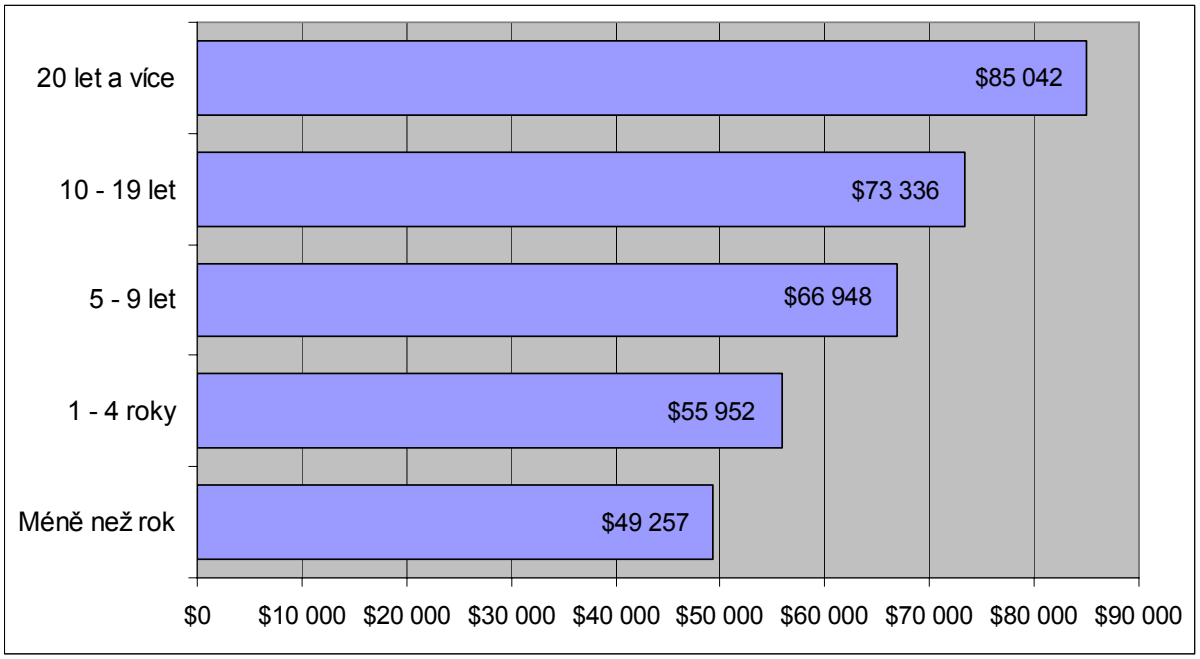
Když je tester velmi dobrý, málo lidí si je toho vědomo. Když vývojář vyřeší velký problém, hodně lidí to vidí a používají program, který napsal. Když tester objeví velké problémy, tyto problémy bývají opraveny dříve, než si jich hodně lidí všimne. Tedy hodnota dobrého testera nebývá vždy zřejmá."

Za testera bývá hlavně považován člověk, který je sebrán z ulice, dostane krok za krokem popsáno, co má dělat s produktem a tento scénář do puntíku plní. Oproti tomu experty na testování je za testera považován člověk s vlastnosti popsanými v kapitole Vlastnosti testera a s nadšením k testování. Takovýto člověk, scénáře popisující co s programem dělat, píše a sám je aktivně zkoumá.

Porovnáme-li čistě platy testerů a programátorů jako je tomu na obr.6 a obr.7, vidíme, že medián platů testerů je nižší.



Obr. 4: Mediány platů podle zkušenosti v letech na pozici softwarový inženýr/ vývojář / programátor v USA [33]



Obr. 5: Mediány platů podle zkušenosti v letech na pozici test / quality assurance inženýr v USA [34]

Na druhou stranu nesmíme zapomínat, že se jedná o mediány a plat závisí od celé řady schopností. Testeři ovládající dobře programování nebo mající jiné klíčové znalosti mohou dostávat větší plat než by dostal stejně zkušený programátor.

Například z bývalých hackerů se často stávají spíše testeři zaměřující se na bezpečnost než programátoři. "Jedna možnost je chovat se příkře. Jiná je uznat, že tito lidé jsou zažraní do bezpečnosti," řekl Kevin Kean, ředitel Microsoftu pro bezpečnostní odezvu, o hackerech. [23].

Z emailu od Cema Kanera[4]:

„Students from my lab who had a combination of programming and testing skill, but who were willing to work as testers have often received job offers for more money than comparable students for programming jobs."

Přeloženo:

„Studenti z mojí třídy, kteří mají kombinaci programovacích a testovacích dovedností, ale kteří jsou ochotni pracovat jako testeři, často dostávají pracovní nabídky s větším platem než porovnatelní studenti za programování."

Z emailu od Pradeepa Soundararajana:

"Bad testers are obviously paid less and good testers are paid more. For instance, I (I claim to be a good tester) get paid at least thousand dollars more than a good developer of my experience."

[4] Cem Kaner je profesorem softwarového inženýrství na floridském institutu technologie a ředitelem floridského technologického centra pro vzdělávání a výzkum v oblasti testování softwaru.

Přeloženo:

"Špatní testeři jsou očividně placeni méně a dobří testeři více. Například já (považuji se za dobrého testera) dostávám zaplaceno alespoň o 1000 dolarů více než dobrý vývojář se stejnou zkušeností."

V knize Software testing fundamentals Marnie L. Hutcheson vykládá o opačné zkušenosti, kdy testerem se stávali ti nejlepší programátoři a zároveň zmiňuje jeden z možných důvodů, proč testování není adekvátně uznávané [2]:

"One of my first mentors when I started testing software systems had been a tester in a boom-able industry for many years. He explained to me early on how a very good analyst could get promoted to programmer after about five years of reviewing code and writing design specifications; then after about five years in development, the very best programmers could hope for a promotion into the system test group. The first two years in the system test group were spent learning how to test the system.

…

Few universities offer software testing classes. Even fewer require software testing classes as part of the software engineering curriculum. Unfortunately, this sends the message to business and the development community that software testing is not worthwhile."

Přeloženo:

"Jeden z mých prvních mentorů, když jsem začala testovat softwarové systémy, byl testerem v rozvíjejícím se průmyslu po mnoho let. Brzy mi vysvětlil, jak velmi dobrý analytik může být povýšen na programátora po pěti letech čtení kódů a psaní specifikací návrhu; pak po asi pěti letech ve vývoji ti nejlepší programátoři mohou doufat v povýšení do skupiny systémového testování. První dva roky ve skupině systémového testování stráví učením se jak systém testovat.

…

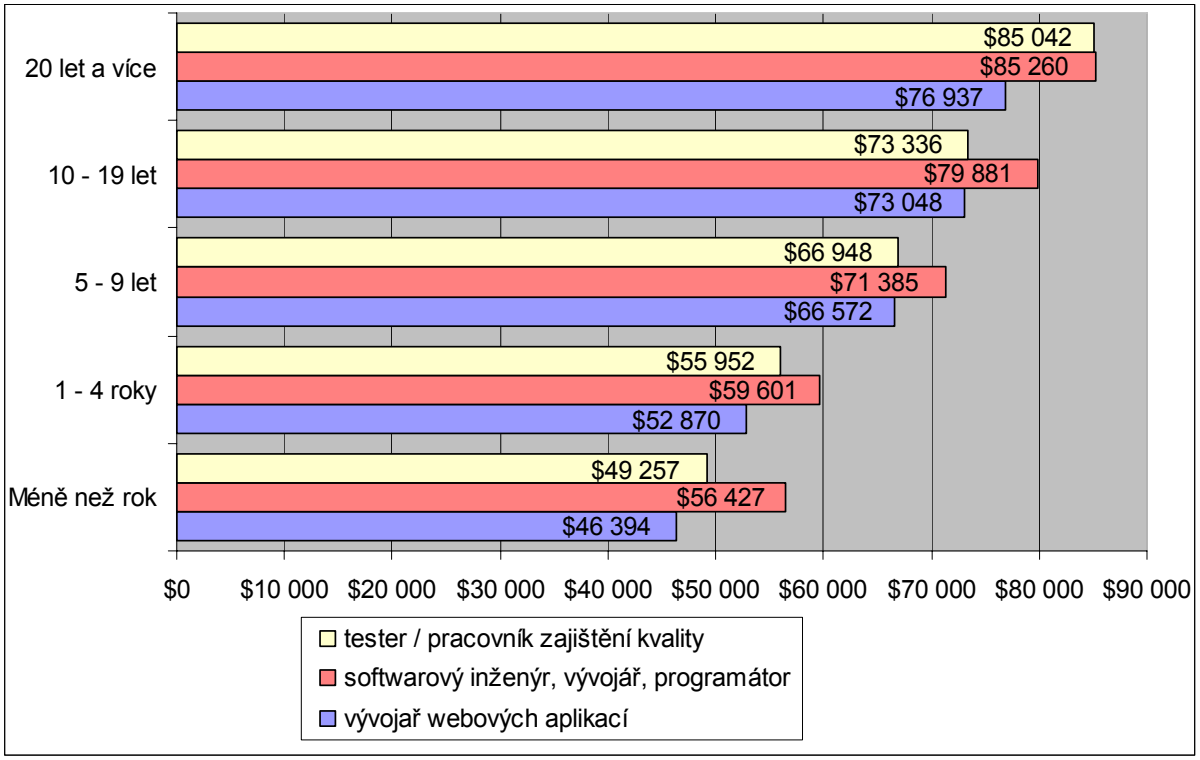
Málo univerzit nabízí semináře testování softwaru. Dokonce ještě méně jich vyžaduje testování softwaru jako součást vzdělání v softwarovém inženýrství. Naneštěstí tím zasílají vzkaz byznysu a vývojářské komunitě, že testování nestojí za to."

Podle tohoto zdroje [2] bylo odhadováno, že ke konci devadesátých let dvacátého století, tedy zhruba před deseti lety, pouze deset procent testerů a vývojářů má za sebou nějaké školení v technikách testování.

V české republice během těchto deseti let vznikla řada školení ohledně testování, kvalita a zejména rozsah těchto školení však stále není dostačující a představuje jen zlomek vědomostí, které by zkušenější tester měl mít.

Z tabulek a dalších průzkumů vyplývá, že plat zkušených testerů často přesahuje plat zkušených programátorů, důvodem k nižšímu mediánu zřejmě je i to, že opravdu schopných testerů je v poměru k celkovému počtu testerů méně než jak je tomu u programátorů.

Stejný jev můžeme ale pozorovat i u různých druhů vývojářů. Například řada programátorů v dnešní době začíná tvorbou webových stránek, medián platů programátorů webových aplikací tak vychází dokonce nižší než u testerů.



Obr. 6: Mediány platů podle zkušenosti v letech na pozice vývojáře, vývojáře webových aplikací a testera [22-24]

4. Kategorie testů

Jeden z prvních problémů při testování je rozhodnutí jak testovat, určení

- jaké druhy testů budou použity,
- kdy a do jaké míry budou aplikovány,
- jaké nástroje a data je třeba připravit.

Vzniklé množině testů se pak říká testovací mix. Spektrum možných testů je velmi široké, proto se třídí do kategorií podle různých hledisek. Obecně platí, že při plánování testů by žádná kategorie neměla být opomenuta a alespoň jeden její zástupce by se měl nacházet v testovacím mixu. O jednom z dělení testů už byla zmínka v předešlých kapitolách a to o statickém a dynamickém testování.

4.1 Statické a dynamické testování

Toto rozdělení vychází z toho, zda je k provedení testu potřeba software spustit. Statické testování nevyžaduje běh softwaru, proto je možné s ním začít ještě před vytvořením prvního prototypu. V současné době je poněkud opomíjeno, což je škoda, protože rychle objevuje chyby, které by se mohly stát kritickými. Navíc je ho možné aplikovat i na složitě otestovatelné části, ke kterým by testeři museli komplikovaně vytvářet dynamické testy a úroveň kvality testování těchto částí by byla nedostačující nebo by byla dosažena s většími náklady. Správně provedené statické testování, například procházení kódu, bývá velmi efektivní při hledání určitých druhů chyb. Dále budou podrobněji probrány některé statické techniky v samostatné kapitole.

Dynamické testování vyžaduje existenci spustitelné verze softwaru a probíhá hlavně na základě poskytování různých vstupů a posuzování výstupů testovaného programu.

4.2 Černá a bílá skříňka

Poněkud odlišné je rozdělení testů podle toho, na základě jakých znalostí o produktu k němu přistupujeme. Při testování černé skříňky se zaměřujeme na vstupy a výstupy programu bez znalosti, jak je naimplementován. Produkt je černou skříňkou, do které se nelze podívat, vidíme jen jak vypadá a jak se chová navenek. Smyslem je analyzovat chování softwaru vzhledem k očekávaným vlastnostem tak, jak ho vidí uživatel. Do této kategorie spadají skoro všechny druhy testů uživatelských rozhraní, akceptační testy, testování podle scénářů, které krok za krokem provádějí uživatele tím, co má zadat a jaké jsou očekávané reakce systému.

Při testování bílé skříňky, v angličtině se jí říká taky skleněná (glass box) nebo průhledná (clear/transparent box), má tester přístup ke zdrojovému kódu a testuje produkt na základě něj. Vidí nejen co se děje na povrchu skříňky, ale i vnitřní reakce systému. Tím poněkud ztrácí pohled uživatele, ale může lépe odhadnout, kde hledat chyby a kde ne.

Mezi těmito kategoriemi vznikla ještě třetí, testování šedé skříňky, kdy víme něco málo o implementaci a vnitřních pochodech produktu, ale ne tolik, aby to bylo považováno za testování bílé skříňky. Například nemáme k dispozici celý zdrojový kód, ale html kód webových stránek nebo informace o designu aplikace.

4.3 Automatické a manuální testovaní

Podle toho, zda jsou testy prováděny člověkem nebo softwarem, se rozlišuje manuální a automatické testování. Pokud test vyžaduje lidské ohodnocení a úsudek nebo rozličné přístupy, které není třeba zaznamenat a pravidelně opakovat, je vhodnější manuální testování. Pro opakované spouštění velkého množství testů nebo testu s velkým množstvím generovaných dat, stejně jako pro zátěžové testování je dobré použít automatické testy. Důležitým hlediskem je taky to, zda vytvořené automatické testy budou levnější, než ručně provedené testy.

4.4 Stupně testování

Na základě toho na jaké úrovni se testování provádí, v jakém časovém horizontu od napsání kódu, se testování dělí na pět stupňů:

- Testování programátorem (Developer testing)
- Testování jednotek (Unit testing)
- Integrační testování (Integration testing)
- Systémové testování (System testing)
- Akceptační testování (Acceptance testing)

Jednotlivé stupně často provádí různí lidé a kód je na každém stupni testován z trochu jiného úhlu pohledu.

První se provádí **testování kódu programátorem** bezprostředně po jeho napsání, přičemž testování má ponejvíce podobu neformálního komponentového testování, které je založené na principu: Otestuj, co jsi napsal. Málokterý programátor by odevzdal kód aniž by ho alespoň jednou nespustil a nevyzkoušel.

Cíle tohoto testování jsou dva. Zaprvé programátor kontroluje, jestli kód dělá to, co on zamýšlel. Zadruhé hledá jestli kód funguje správně, protože v tomto okamžiku je oprava vzniklých chyb nejméně nákladná.

Na rozdíl od testování programátorem, je kód v následujících fázích testován z hlediska uživatele, zda dělá to, co uživatel požaduje nebo očekává.

Testování jednotek je proces podrobného testování co možná nejmenších částí kódu. U objektově orientovaného kódu se jedná o testování jednotlivých metod a tříd. Protože tyto testy se zapisují rovněž ve formě programového kódu, můžou je provádět programátoři pod dohledem testovacího týmu, který vyhodnocuje výsledky, nebo testeři se znalostmi programování. Pro usnadnění a rychlejší psaní těchto testů se používají nástroje na bázi frameworků. Tyto testy jsou automatické, snadno opakovatelné a dobře se zpracovávají. Vzniklé testy se nazývají test unitami. Pokud jsou napsány předem, jako tomu je u testy řízeného vývoje, můžou nahradit testování kódu programátorem.

Příklad kódu testu napsaného pomocí JUnit frameworku v rámci testování jednotek :

```
import junit.framework.*;

public class ChangeTest extends TestCase {

private Money old;
private String newCur1;
```

```
private String newCur2;
private float exRate;
Rates rateC = new Rates();

public ChangeTest(String name) {
      super(name);
}

// příprava prostředí na test
protected void setUp() throws Exception {
      old = new Money(25, "CZK");
      newCur1 = new String("CZK");
      newCur2 = new String("EU");
      exRate = rateC.getRateCZtoEU();
}

// úklid prostředí po testu
protected void tearDown() throws Exception {

}

public void testChangeSame() {
      Money expected = new Money(25, "CZK");
      assertEquals(expected, old.change(newCur1));
}

public void testChangeDiff() {
      Money expected = new Money(25*exRate, "EU");
      assertEquals(expected, old.change(newCur2));
}

}
```

Integrační testování se zaměřuje na zajištění správného chování nových podsystémů poté, co jsou zapojeny společně s ostatními částmi systému. Integrační testy mohou být jednoduché a automatické, ale i komplexní a manuální, záleží na situaci a systému. Testy připravuje hlavně testovací tým, a pokud se jedná o testy automatické, spouštět je pak může kdokoli.

Během **systémových testů** je aplikace testována jako celek, a proto tyto testy bývají prováděny v pozdějších fázích vývoje. Pokud se jednotlivé testovací týmy předtím zaměřovaly jen na určité funkčnosti, které spadaly pod jejich kompetenci, jako tomu bylo v případu Mars Polar Lander, při systémových testech by měly být testovány obsáhlé scénáře procházející napříč celým systémem a simulována běžná práce uživatele se systémem. Součástí této fáze je provádění řady různých druhů testů jako jsou testy výkonnosti, spolehlivosti, bezpečnosti atd. Tyto testy bývají vytvářeny testovacím týmem, který by měl obsahovat specialisty na zastoupené druhy testů.

Akceptační testy jsou součástí převzetí produktu zákazníkem a ověření, že software je připraven na nasazení do ostrého provozu. Často bývají prováděny podle scénářů, které jsou připraveny předem a dohodnuty oběma stranami. Software je při tom testován zákazníkem nebo jím pověřeným testovacím týmem. V rámci přípravy můžou být tyto testy simulovány i ze strany dodavatele, který zjišťuje, nakolik je systém připraven a hledá poslední chyby.

4.5 Pokrytí testy

Další možným dělením testů je podle toho, jakou část vyvíjeného softwaru pokrývají a jakým způsobem. Pokrytí je zároveň i metrikou, která nám říká, kolik z produktu jsme už otestovali a kolik nám zbývá. Pokrytí kódu se měří v procentech, kolik z celkového počtu sledovaného je pokryto testy. Nejprve je třeba se rozhodnout, jakým způsobem budeme pokrytí testy hlídat. Pokrytí bereme v úvahu vzhledem k

- požadavkům,
- funkcím,
- a zdrojovému kódu.

Zatím co první dva druhy pokrytí se chápou intuitivně, třetí typ - pokrytí kódu, v angličtině pod názvem code coverage, bývá popsán formálněji a rozdělen na další podtypy. U kritických aplikací, kde se klade důraz na bezpečnost a spolehlivost, je běžným požadavkem vysoké až úplné pokrytí kódu některého typu. Pokrytí kódu byla jedna z prvních technik systematického testování a bývá spojováno s automatickým testováním. Například právě jedna z výhod testovaní jednotek je, že umožňuje sledování pokrytí kódu. Existuje však řada způsobů, které se liší v pečlivosti s jakou požadují, aby byl kód otestován.

Pokrytí příkazů nebo taky řádků je tou nejjednodušší variantou. Pouze kontroluje, zda daný příkaz, řádek byl proveden v průběhu testování. Formálně množina testů T splňuje kritérium pokrytí příkazů pro daný kód K, jestliže pro každý příkaz p náležící kódu K existuje test t z množiny T, že při provedení t bude spuštěn příkaz p.

Pokrytí příkazů pouze kontroluje, zda existuje vůbec nějaký test na daný příkaz, ne pečlivost testu. Tento rozdíl je patrný u jakýchkoli podmínek, kdy pro pokrytí jejího příkazu stačí libovolné její vyhodnocení.

Příklad stoprocentního pokrytí příkazů bez zjištění chyby:

```
int* p = NULL;
if (x>0)
      p = &x;
*p = 1;
```

Při otestování tohoto kódu pro x=5 dosáhneme pokrytí všech příkazů. Ale není otestována varianta, kdy podmínka selže, takže ani nemusíme zjistit, že program selže při zapisování čísla na neplatnou adresu.

Pokrytí hran (větví, rozhodnutí) požaduje, aby v případě podmínek byly testovány jak pro kladné tak záporné vyhodnocení. Představme si graf, kdy příkazy tvoří uzly a možné přechody mezi nimi hrany. Pak za sebou provedené příkazy tvoří hranu a podmínka uzel, ze kterého vedou dvě hrany, jedna pro true a jedna pro false hodnotu. Pak množina testů T splňuje kritérium pokrytí hran pro daný kód K, jestliže pro každou hranu h výše popsaného grafu existuje test t z množiny T, že při provedení t projde výpočet hranou h.

Přestože toto pokrytí je o něco podrobnější, u složitějších podmínek může dojít opět k pokrytí bez detekování chyby.

Příklad stoprocentního pokrytí hran bez zjištění chyby:

```
if (a>0 && (b>0 || c>0))
```

```
a = a+b/c
```

Pokud mezi testy zahrneme případy, kdy podmínka selže, protože například a je -5, a případ kdy a, b i c jsou kladné, dosáhneme úplného pokrytí hran, ale neodhalíme v tomto případě možné dělení nulou z důvodu špatně zapsané podmínky.

Pokrytí podmínek pak vyžaduje nejen kladné a záporné vyhodnocení těchto podmínek, ale jedná-li se o složenou podmínku, také všech podčástí. Množina testů T splňuje kritérium pokrytí podmínek pro daný kód K, jestliže splňuje kritérium pokrytí hran a pro každou složenou podmínku a každou její část p, existují testy t, u z množiny T, že při provedení t se p vyhodnotí kladně a při provedení u záporně.

Pokud se vynechá část, že pokrytí podmínek musí splňovat pokrytí hran, tak k nepokrytí hran a pokrytí podmínek dochází v případě, že pro všechny možné vstupy je vždy celá podmínka vyhodnocena jako kladná nebo záporná.

Příklad stoprocentního pokrytí podmínek bez zjištění chyby:

```
if  x <> 0  then  y = y - x;
if  y > 1  then y = y/x;
```

Pokud množina testů bude obsahovat testy, kde x je různé od nuly, nebo je sice nula, ale y není větší než jedna, bude pokrytí podmínek úplné, ale chyba dělení nulou touto množinou testů nebude odhalena.

Pokrytí cest sleduje, zda jsou otestovány všechny možné průchody kódem. Množina testů T splňuje kritérium pokrytí cest pro daný kód K, jestliže splňuje kritérium pokrytí podmínek a pro každou cestu C v grafu kódu spojující vstupní a výstupní uzel grafu a obsahující nejvýše n cyklů existuje test t z množiny T, že při provedení t projde výpočet cestou C.

Pokrytí cest už představuje velmi pečlivé otestování kódu, ale jeho použití má velká omezení. Jednak počet cest programem roste až exponenciálně a jednak ne všechny cesty je možné provést z důvodu závislosti mezi daty a příkazy.

Existují i další typy pokrytí kódu, které nejsou tak časté a které se většinou specializují na určité části kódu, jako například pokrytí tabulek, cyklů, volání atd.

Více o pokrytí kódu je napsáno v [18], definice v této části vycházely z materiálu [8].

4.6 Dimenze kvality

Další dělení testů vychází z něčeho, čemu se říká dimenze kvality. Jsou to různé aspekty vyvíjeného produktu, ke kterým se vztahují požadavky a očekávání uživatelů. Jejich seznam je důležitý hlavně proto, aby nebylo opomenuto sesbírat požadavky a připravit testy pro žádnou z těchto dimenzí.

Dimenze kvality:

- Funkčnost (**F**unctionality) – zajišťuje správné chování funkcí systému
- Použitelnost (**U**sability) – zda je systém uživatelsky přívětivý, zda se s ním dobře pracuje

- Spolehlivost (**R**eliability) – zda se chová stejně za všech okolností, zvláště po přetížení, po chybě
- Podporovatelnost (**S**upportability) – zda se systém dobře instaluje, nemá problémy s cílovými hardwarovými a softwarovými konfiguracemi a další vlastnosti související s údržbou systému.
- Výkon (**P**erformance) – zda systém není pomalý a zvládne větší množství současně pracujících uživatelů
- Ostatní (+) – Zde patří třeba lokalizovatelnost - snadný převod do jiných jazyků, kompatibilita – možnost kombinace s jiným softwarem nebo hardwarem, bezpečnost a všechny další požadavky, které se nehodí do předchozích kategorií.

Tyto dimenze bývají někdy označovány zkratkou FURSP+, kterou tvoří začáteční písmena anglických názvů jednotlivých dimenzí.

5. Dokumentace

Součástí procesu testování je neodmyslitelně tvorba dokumentace, bez které by se projekt potýkal s řadou problémů, hlavně v komunikaci a se ztrátou znalostí při fluktuaci lidí v testovacím týmu. Dalo by se říct, že o testování se dá hovořit pouze v tom případě, když existuje alespoň nějaká forma dokumentace. V případě menších projektů může být dokumentace i hodně neformální, stačí zaznamenávání reportů chyb a jejich stavů, případně navíc plán toho, co se bude testovat a jakými druhy testů, vyvěšený na nástěnce.

V jiných případech bývá dokumentace naopak značně podrobná s přesně definovaným obsahem, připravovaná podle metodiky nebo dokonce standardů. Přímo tvorbou testovací dokumentace se zabývá standard IEEE-829, který je z tohoto důvodu pro účely dokumentace nejznámější. Často je možné se také setkat se standardem ISO 9000-3, která se zabývá kvalitou softwaru.

Rozhodnutí o formě dokumentace a použití standardů záleží na požadavcích vývojového a testovacího týmu a zákazníka. Jako návod mohou posloužit odpovědi následující otázky:

1) **Je dokumentace testování vaším produktem nebo nástrojem?** Pokud slouží testovací dokumentace pouze pro tým, záleží hlavně na jeho požadavcích, které na dokumentaci klade. Dokumentace se bude vyvíjet tak, aby poskytovala co největší užitek za co nejmenší cenu. Při psaní se bude hlavně hledět na účel, který má splnit. Oproti tomu, pokud zákazník, pro kterého je softwarový produkt vyvíjen, bude požadovat odevzdání testovací dokumentace a zaplatí si za ni, stane se tato dokumentace samostatným produktem. Pak je to zakázka, které je třeba věnovat patřičnou pozornost a dokumentace bude odpovídat jakémukoli standardu nebo metodice, jakou zákazník bude požadovat.
2) **Odvíjí se požadovaná kvalita od smyslu aplikace nebo je určována požadavky trhu?** Pokud smyslem aplikace je ovládání procesů, které mohou ohrozit lidský život nebo způsobit obrovské ztráty, může být testovací dokumentace vyžadována při auditech nebo u soudních pří a být důkazem, že společnost při vývoji nic nezanedbala. Pokud bude rozhodovat hlavně poměr kvalita/cena, kterého vyvíjená aplikace dosahuje, bude opět na prvním místě užitečnost. Dokumentace by měla poskytovat všechny aspekty, které testovací tým potřebuje, ale nebude se vytvářet nic navíc, bez čeho by se dalo lehce obejít. Při uvažování o tom, co vše se vyplatí vytvářet, je dobré zvážit odpovědi na další otázky.
3) **Jak často se bude měnit specifikací požadované chování aplikace nebo vzhled? Jsou požadavky jasné? Očekávají se v budoucnu časté změny?**
Podrobná dokumentace bude při častých změnách rychle stárnout a bude brzo stejně neplatná nebo její údržba zabere větší procento času testovacího týmu. Oproti tomu při stabilním návrhu aplikace je možné čerpat z výhod podrobnější dokumentace. Například by se mohlo vyplatit psaní a zdokumentování automatických testů, u kterých se dá v této situaci usoudit, že s minimální údržbou postupně ušetří týmu více času, než kolik stála jejich tvorba.
4) **Jak dlouho bude dokumentace používána?**
Při psaní stejných dokumentů, z nichž jeden bude používán měsíc a druhý rok, a ani u jednoho se nepředpokládá, že by během této doby přestal být aktuální, je pravděpodobné, že podrobnější text se vyplatí u toho dlouhodobějšího dokumentu. Jednak z toho důvodu, že vynechané informace by se díky fluktuaci členů nedaly tak lehce zjistit jinde. A jednak proto, že zapsání podrobností ušetří čas, který by

čtenáři strávili zjišťováním podrobností. Za rok si daný dokument bude potřebovat přečíst více lidí než kolik lidí si přečte druhý dokument za měsíc, ale zapsání podrobností v obou případech bude trvat stejně.

5) **Může už nějaká existující dokumentace alespoň částečně nahradit dokumentaci testů?** Někdy jsou vytvářeny scénáře, vedoucí uživatele k tomu, jak přesně má aplikaci testovat, a popisující, jaké je její očekávané chování. Tyto scénáře jsou jedním z dokumentů testování. Jindy se spíše spoléhá na úsudek a schopnosti testerů, na čemž je založené například průzkumné testování, kdy tester poznává a analyzuje aplikaci zároveň. V takovém případě se scénář, jak má tester postupovat, nepíše, a pro případné zjištění informací o aplikaci je možné využít specifikací, uživatelských manuálů a dalších už existujících ale aktuálních dokumentů.
6) **Komu všemu je dokumentace testování určena? Bude se využívat pro školení, výzkum či jiné činnosti nepatřící do testovacího procesu?** Je-li co nejlépe odhadnut přínos dokumentu, dokážeme snadno odhadnout, jestli se vyplatí ho vytvořit a kolik času na jeho napsání věnovat.

Pomoci při rozhodování mohou i měřítka zabývající se cenou dokumentace. Sleduje se zejména, kolik stojí tvorba dokumentace testování na jeden testovací případ nebo jeden případ užití a kolik stojí údržba testování na jeden testovací případ nebo jeden případ užití. O use casu nebo-li případu užití se dá říci, že představuje jednu konkrétní funkčnost softwaru, používá se při use case analýze, která bývá součástí specifikace.

5.1 Nejdůležitější dokumenty podle praxe

Testovací plán je dokument řídící celý proces testování, proto téměř všechny zdroje považují testovací plán za nejdůležitější dokument. To ale je pravdou pouze pro samotný proces, řada projektů se bez něj obejde nebo dokonce testeři ani neví, zda takový plán existuje a kde ho najít.

Testovací plán obsahuje určení rozsahu testování, definuje jaké testy budou na co aplikovány, jaké zdroje je třeba zajistit včetně nástrojů, dat a lidí, určuje zodpovědnosti, stanovuje pravidla a identifikuje rizika.

Testovací plán bývá na větších projektech prvním vytvářeným dokumentem, připravuje se dřív, než se začne testovat, někdy bývá některou řídící pozicí v rámci testování vytvořen ještě dřív než je stanoven tým testerů.

Seznam testovacích nápadů nebo myšlenek (test idea list) je dokument, do kterého si testeři píší své návrhy na testy, které by stály za to vyzkoušet při testování. Tyto nápady jsou roztříděné na základě toho, který prvek aplikace zkoumají. Testovací myšlenkou je např. dělení nulou, zadání speciálních znaků, ovládání menu klávesnicí.

Testovací případ, známý i pod anglickým označením test case je seznam kroků testujících jeden konkrétní případ, který při používání testované položky může nastat. Součástí je i doprovodný výčet množiny vstupů, podmínek a očekávaných reakcí. Testovací případy vznikají na základě testovacích nápadů a případů užití získaných např. ze specifikace.

Příklad testovacího případu - pokus o změnu hesla na aktuální heslo:

```
Počáteční bod: Hlavní stránka aplikace
Předpoklady: Aktuální platné heslo uživatele je A456pk
```

```
Menu -> Osobní nastavení -> Změna hesla
Do políčka Aktuální heslo zadejte:  A456pk
Do políčka Nové heslo zadejte: A456pk
Do políčka Potvrzení hesla zadejte: A456pk
Potvrďte tlačítkem OK.

Očekávaná reakce: Aplikace zahlásí chybovou hlášku: „Dané heslo už
bylo v minulosti použito.“ Heslo není změněno, což se projeví mimo jiné
tak, že nedojde k žádným změnám v historii hesel.
```

Testovací scénář slučuje několik za sebou jdoucích testovacích případů do tématicky uspořádaného scénáře, např. testování změny hesla.

Testovací skript je přepis testovacího příběhu nebo scénáře do programovacího jazyka, scénáře jsou určené k automatickým testům aplikace.

Testovací data jsou sebraná skutečná nebo podle předem určených pravidel vygenerovaná data určená pro účely testování. Přestože nemívají většinou podobu dokumentu, v praxi bývají považována za součást dokumentace.

Hlášení chyby je oznámení popisující možnou chybu, postup vyvolání jejího projevu, její dopad, význam, stav opravy a další náležitosti chyby. Hlášení chyb je možno považovat za produkt testování, proto je množina těchto reportů ve skutečnosti nejdůležitějším dokumentem.

Test log slouží k zaznamenání proběhlých testů. Mívá tabulkovou podobu s informacemi kdo, kdy, jaké testy spustil a s jakým výsledkem.

Test result dokument obsahuje souhrnné informace o provedeném testování typicky za jeden testovací cyklus, začínající odevzdáním nové verze vyvíjeného produktu testovacímu týmu. Test result dokument poskytuje podrobné hodnocení kvality dané verze a zvoleného přístupu k testování.

Test evaluation dokument zpětně hodnotí proces testování, nejen za jednotlivé testovací cykly ale za dobu celého vývoje. Je to dokument na stejné úrovni jako testovací plán, ale dívá se na problém z opačného konce, poskytuje zpětnou vazbu, jak byl proces testování úspěšný a co je možné do příště zlepšit.

5.2 Nejdůležitější dokumenty podle standardu

Standard IEEE 829 definuje osm dokumentů pro proces testování softwaru, standard přitom ale nevyžaduje používání celé množiny, pouze popisuje jejich podobu.

- Testovací plán
- Test design specification: definuje zvolený testovací přístup, co se bude testovat a jak
- Test case specification: obsahuje vytvořené testovací případy
- Test procedure: popisuje spouštění testů .

- Test item transmittal report: specifikuje které části aplikace mají být otestovány. Vytváří se zvláště, pokud testuje několik týmů nebo je vývoj rozvětven do několika souběžných projektů
- Test log
- Test incident report: obecnější forma hlášení chyby. Reportuje se jakákoli událost vzniklá při testování
- Test summary report: Stejné jako test report, hlásí souhrnné výsledky testování a hodnotí proces testování.

5.3 Testovací nápady

Seznam testovacích nápadů může obsahovat úrovně pohledu, čím je představa o aplikaci konkrétnější, tím podrobnější můžou nápady být. V případě, že vytváříme seznam testovacích myšlenek pro celou kategorii webových aplikací, vzniklé nápady budou mít spíše podobu druhů testů, např. testování aplikace s různou rychlostí připojení k internetu, na různých prohlížečích a jejich nastavení, atd.

Testovací nápady vytvořené pro určitý typ vstupu, jako je pole očekávající pouze celočíselné hodnoty, jsou naopak velmi konkrétní a lehce znovupoužitelné na všechna taková pole obsažená i v jiných aplikacích.

Z tohoto důvodu nejčastěji bývají generovány myšlenky pro konkrétní aplikaci, se kterou je tester dobře seznámen.

Seznam testovacích nápadů se nejlépe vytváří za pomoci techniky brainstormingu poblíž počítače dostupného k internetu a za pomoci předem připravené struktury kategorií podle které se nápady budou generovat. Ta slouží k zacílení přemýšlení určitým směrem což podporuje vymýšlení nových nápadů.

Příklad krátkého seznamu nápadů vygenerovaných se zaměřením na aplikaci provozující internetový obchod [43]. Vytvoření seznamu testovacích nápadů zabere několik hodin nebo i dní, tento 15 minutový výstup proto nemůže být kompletní:

```
Strávený čas: 15 minut brainstormingu.
-------Start--------------------
Špatná použitelnost:
      • Uživatel nemůže přidat položku do košíku přímo z výsledků
        vyhledávání.
      • Uživatel neví kolik položek má v košíku a nezná celkovou cenu v
        každém časovém okamžiku.
      • Uživatel musí projít příliš mnoho stránek než dokončí objednávku.
      • Je obtížné používat systém: složité přidat, odstranit a
        aktualizovat.
      • Není vidět konečná cena nebo její odhad.
      • Vyhledávání je příliš složité nebo je těžké najít pole pro
        vyhledání na stránce.
      • Nelze najít nápovědu.
      • Formulář pro zpětnou reakci zákazníků je nedostupný.
Výpočetní chyby:
      • Odstranění/přidání položky do košíku neaktualizuje celkovou cenu.
      • Záporný počet položek sníží celkovou cenu.
      • Slevy nejsou správně započítány.
      • Poštovné a daně nejsou správně započítány.
```

- Funkce přepočítání selže.

Lokalizace:

- Pole registračního formuláře neakceptují rozšířené znaky cizích abeced.
- Pokud jsou tyto rozšířené znaky zadány při registraci, naruší se jimi databáze.
- Není možné aplikaci přeměnit na vícejazyčnou.
- Není možné přijmout objednávku do cizí země a integrovat poštovné do jiných zemí.

Selhání ISP/Web hosta:

- Uživatel se úspěšně zaregistruje nebo odešle objednávku, ale email s potvrzením mu nedojde.
- Nevratná ztráta dat na hostujícím centru.
- Zálohovací programy selžou na hostujícím serveru a způsobí ztrátu dat.

Síťové problémy

- Odkaz do databáze zboží selže.
- Odkaz do databáze uživatelských profilů selže.

Kompatibilita

- Stránky neodpovídají standardu HTML (W3C).
- Chybí kompatibilita s některými možnými platebními kartami nebo systémy.

Škálovatelnost[5]

- Přidání do vozíku, odeslání objednávky a vyhledávání trvá příliš dlouho během špičky.
- Žádostem vyprší doba platnosti během špičky

Bezpečnost

- Otestovat sílu šifrování.
- Otestovat zranitelnost vůči útoku přetečení bufferem.
- Otestovat zranitelnost vůči SQL injection.

Soukromí klienta

- Zkontrolovat, zda existuje politika na ochranu soukromí.
- Zkontrolovat vypršení platnosti cookies: zkontrolovat, zda někdo může přistupovat k obsahu košíku předchozího uživatele v případě sdílení počítače.
- Otestovat existenci automatického odhlášení v případě neaktivity.
- Neschopnost odmítnout účast v analýzách návštěvnosti a prodeje.

Selhání webového serveru

- Chybí uživatelsky přívětivá chybová stránka, v případě, že požadovaná stránka není nalezena.
- Server padá pod velkou zátěží.

Selhání softwaru třetích stran

- Selhání systému ověřujícího platnost platební karty

--------Konec----------

[5] Škálovatelnost je schopnost vypořádat se s růstem a změnami, bez nutnosti větších nebo vůbec nějakých zásahů do systému.

5.4 Reportování chyb

Reportování nalezených chyb je nedílnou součástí práce testera, která by se neměla odbývat. Srozumitelný dobře vystihnutý report může pomoci urychlit nalezení a odstranění chyby, zatímco špatný report bude přehazován i několikrát mezi testerem a vývojářem, který ho bude špatně interpretovat nebo vracet s žádostí o vysvětlení některých věcí. Nesrozumitelný report může stát až desítky hodin práce navíc, nebo chyba dokonce nebude opravena vůbec.

Osvědčeným způsobem jak zajistit kvalitní reporty je definovat jejich přesnou podobu, kterou je třeba dodržet a určit zkušeného testera, který bude za reporty a systém, kam se ukládají, zodpovědný. Reporty se mohou vytvářet a ukládat v textovém nebo tabulkovém editoru, ale častěji se používá speciálně k tomu vytvořený program. Těmto programům se říká bug report systémy nebo bugzilla.

Z povahy testování, jak byla vykreslena, vyplývají pro reporty dva důležité postřehy:

- Defekt report je u většiny testerů nejviditelnějším produktem jejich práce. Reporty mohou být jediná věc, co o testerovi bude spousta jeho nadřízených a kolegů znát. Jeho reputace tak bývá založena na tom, co do nich napíše.
- Nejlepším testerem není ten, který zahanbí nejvíce programátorů, ale ten, jemuž se podaří dostat co nejvíce chyb do opraveného stavu.

Dobře napsané hlášení chyby se snaží dodržet následující pravidla vždy, kdy je to možné:

- Je stručné,
- přesné a podrobné,
- napsané v neutrálním tónu (bez ironie, ponižování členů týmu),
- zaobírá se pouze jedinou chybou,
- ale postihuje ji v její nejobecnější podobě – zda má chyba i jiné projevy, zda se vyskytuje i jinde, např. zda je na jednom nebo všech formulářích,
- popisuje, jak je možné chybu reprodukovat
- a na koho chyba bude mít dopad,
- poskytuje co nejvíce informací pro hledání problému při ladění,
- předkládá důkazy o chybě.

Aby hlášení mohlo poskytnout všechny tyto informace, tak se po objevení chyby ještě před napsáním provádí takzvaný follow up testing (v češtině pod méně známým pojmem následné testování), které je zaměřené na zjištění skutečností výskytu a dopadu chyby. Toto následné testování trvá od 10 minut až do několika hodin v případě špatně reprodukovatelných chyb. Tyto špatně reprodukovatelné nebo se jim taky říká nereprodukovatelné chyby mizí a objevují se na první pohled zcela náhodně. Problém je v tom, že chyby se projevují pouze za jistých podmínek, pokud neznáme všechny podmínky, které je třeba nastavit, aby došlo k projevu chyby, vzniká dojem nereprodukovatelné chyby. V takovém to případě je nejlepším řešením si okolnosti chyby poznamenat bokem, promluvit si s programátory, zda neví, co by mohlo být další podmínkou k vyvolání chyby a při každém jejím opětovném výskytu, zkusit znovu nalézt postup, jak chybu nasimulovat.

Hlášení i zatím nereprodukovatelné chyby patří do bug report systému, protože je chybou a její oznámení i když není úplné, je užitečnou informací. Proces výroby softwaru ovlivňuje hodně lidí, a proto má i hodně lidí, kteří mají na něm zájem, jsou jeho stakeholdery. Pro každého z nich může být důležitý jiný aspekt kvality. Chyba patří do bugzilly, pokud ji tam některý ze stakeholderů chce, pokud v jeho očích snižuje hodnotu vyvíjeného softwaru.

Příklad hlášení chyby:

```
Shrnutí:          Rozhozený vzhled na mnoha místech aplikace
Závažnost:        C
Priorita:         1
Komponenta:       GUI
Prostředí:        Internet Explorer 6.0, SP2
Zadavatel:        Anna Borovcová
Datum nahlášení:  1.4.2007
Příloha:          C:\chyby\bug54.jpg

Popis:

      Přestože je Internet Explorer jednou z cílových platforem, vzhled
aplikace se po zobrazení v tomto prohlížeči liší od schváleného vzhledu,
ze kterého vychází uživatelská dokumentace.
      Špatné zobrazení se projevuje zejména posunutými tlačítky,
překrývajícími se prvky a rámečky, kterým občas chybí jedna nebo i více
bočních čar a špatné viditelnosti některých prvků, viz přiložený obrázek.

      Špatné zobrazení bylo nalezeno v následujících částech GUI:
      - v logu aplikace
      - v záložce stahování
      - v okně stahování položky a vytvoření nové záložky
      - v záložce server

      Další části se zobrazují správně.
```

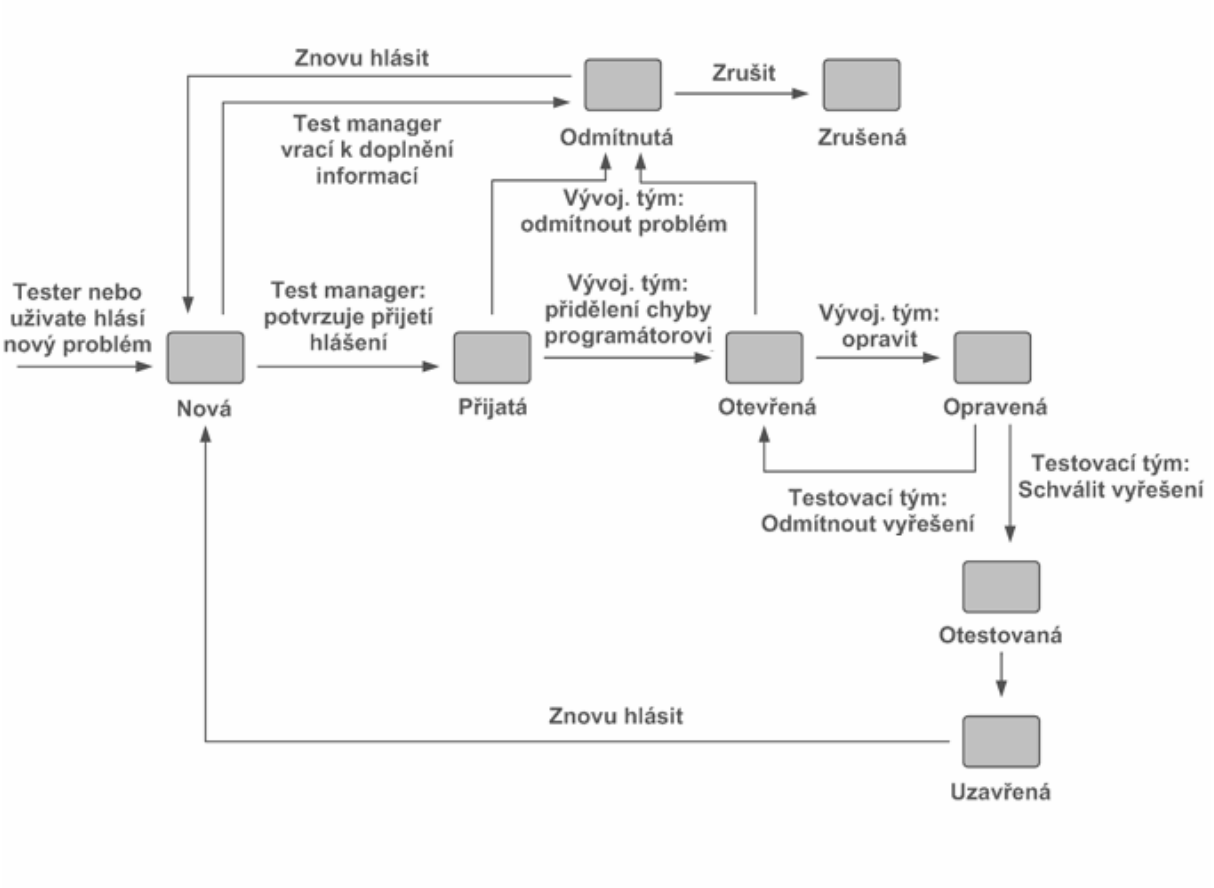
Různé chyby snižují kvalitu softwaru různou mírou, proto při reportování chyb se uvádí závažnost, která vypovídá o tom, nakolik chyba brání používání aplikace a priorita, která určuje pořadí, ve kterém se chyby budou opravovat.

Běžné dělení chyb podle závažnosti (v angličtině severity):
A – V důsledku chyby není možné pokračovat v operaci
B – Došlo k závažné chybě, ale je možné chybu obejít a pokračovat
C – Důsledkem chyby jsou jen menší nepříjemnosti

Při prohlížení většího množství chyb se zobrazují hlášení ve formě seznamu, který obsahuje jen ty nejdůležitější údaje charakterizující chybu. Při prohlížení seznamu chyb vždy čtenáře nejvíce zajímá shrnutí, které v jedné větě popisuje podstatu chyby, dále pak závažnost a další údaje. Bug report systém neslouží jen programátorovi a testerovi, jen pár lidí bude číst hlášení chyby celé, většina si prolétne jen seznamy a vyhledá, co potřebuje nebo co ji zajímá. Proto je třeba věnovat shrnutí chyby stejnou pozornost, jakou věnují novináři titulku svého článku.

Po zapsání chyby do systému pro reportování, hlášení nezůstává neměnné, ale prochází vlastním životním cyklem. Kde se v tomto cyklu hlášení chyby nalézá je zaznamenáno v kolonce stav. Životní cykly chyb se liší nejen mezi projekty, ale i podle toho, zda chybu hlásí zákazník nebo tester, zda je projekt ve vývoji nebo se jedná o servis. Příklad možného životního cyklu hlášení je na obrázku 9.

Základním účelem hlášení je poskytnout vše potřebné k tomu, aby chyba byla správně a rychle opravena. Často jsou proto hlášení impulzem k diskuzi mezi členy vývojového týmu. Tato diskuze je často součástí hlášení a dokumentuje, co a proč se s chybou děje. Například pokud tester vrátí opravu programátorovi, tak mu vysvětlí, proč nepovažuje chybu za opravenou. Pokud není možné diskuzi vést přímo jako součást chyby, je nejvhodnější přidat k chybě alespoň kolonku pro zdůvodnění poslední změny.



Obr. 7: Životní cyklus hlášení chyby

5.5 Metriky

Plánování a hodnocení procesu testování zahrnuje identifikaci měřítek a kritérií pro určení dosažené úrovně kvality. Při testování se pracuje s řadou neznámých, které je nemožné zjistit. Nejdůležitější hodnoty, kterými jsou celkový počet chyb v produktu a na kolik by přišlo neopravit nějakou chybu, je možné pouze hrubě a náročně odhadovat. Proto se hledání odpovědí na otázky, jak daleko je testování a jak je úspěšné, zaměřuje na kombinaci metrik, které jsou na projektu snadněji spočitatelné. Výběr měřítek záleží hlavně na tom, jaké odpovědi na otázky ohledně procesu testování jsou považovány za uspokojující.

Příklad, jak vypadají odpovědi na dotazy ohledně procesu testování na projektu bez použití metrik vyprávěný Hutchesnovou v [2]:

"The director asked the tester, "So you tested it? It's ready to go to production?
The tester responded, "Yes, I tested it. It's ready to go."
The director asked, "Well, what did you test?"
The tester responded, "I tested *it*."
In this conversation, I was the tester. It was 1987 and I had just completed my first test assignment on a commercial software system. I had spent six months working with and learning from some very good testers. They were very good at finding bugs, nailing them down, and getting

development to fix them. But once you got beyond the bug statistics, the testers didn't seem to have much to go on except *it*. Happily, the director never asked what exactly *it* was."

Přeloženo:

„Nadřízený se ptá testera, "Takže jste to otestovali? Je to připravené na produkci?"
Tester odpoví, "Ano, otestovali jsme to. Je to připravené."
Nadřízený se ptá, "Co jste testovali?"
Tester odpoví, "Testoval jsem *to*."
V této konverzaci, jsem já byla testerem. Byl rok 1987 a já zrovna dokončila moje první přiřazení na testování komerčního softwarového systému. Strávila jsem šest měsíců prací a učením se od některých velmi dobrých testerů. Byli jsme velmi dobří v hledání chyb, následném testování a zajišťování oprav. Ale vyjma statistik o chybách, testeři neměli nic než *to*. Naštěstí, nadřízený se nikdy nezeptal, co přesně je *to*."

Bach a Bolton v [16] popisují, co se může v praxi skrývat za odpovědí, že to funguje.

"Jerry Weinberg has suggested that nailing "it works" may mean "We haven't tried very hard to make it fail, and we haven't been running it very long or under very diverse conditions, but so far we haven't seen any failures, though we haven't been looking too closely, either." In this pessimistic view, you have to be on guard for people who say *it* works without checking *even once* to see if *it* could work."

Přeloženo:

„Jerry Weinberg podotkl použití "funguje to" může znamenat, "Moc jsme se nesnažili to shodit a nespouštěli jsme testy moc dlouho nebo za různých podmínek, ale dosud jsme nenašli žádné selhání, ačkoli jsme se na to taky nedívali dostatečně podrobně." Z tohoto pesimistického pohledu, je třeba se mít na pozoru před lidmi, kteří říkají že *to* funguje, bez toho, aby *to* zkontrolovali *aspoň jednou*."

Z výše uvedených příkladů je vidět, jak vypadá testování bez použití metrik, a jsou v nich uvedené některé příznaky problémů, kterým je vhodné věnovat pozornost, protože povědomí o testování má pouze úzký okruh lidí v testovacím týmu. Mimo tento tým nemají nadřízení dostatečné informace o procesu testování, což může nejen vést k prodražování testování, ale hlavně k oddělení nositelů rozhodnutí od nositelů zodpovědnosti. Testovací tým nebývá zodpovědný za kvalitu ani rozhodnutí ohledně ní, většinou je poradním orgánem, který získává a předkládá informace potřebná pro rozhodnutí. Pokud není schopný poskytnout podklady a jen říká, jaké to rozhodnutí má být, manager zodpovědný za rozhodnutí většinou poslechne, tím pádem tým dělá rozhodnutí za něj.

Řešením není přenesení zodpovědnosti za tým, ten totiž nemá dostatečné pravomoci, navíc pokud není efektivní, tak sám nepřijme patřičná opatření a neprozradí nadřízeným, že by měli minimálně část týmu vyhodit a najmout někoho zkušenějšího.

V test reportu nebo v odpovědích na otázky nadřízených by se mělo vyskytovat několik základních nebo odvozených metrik, aby stav projektu byl co nejlépe odhadnut.

Metriky pro odhad kvality bývají založené na:

- chybách,
- testech
- nebo kódu.

Metriky založené na chybách

Samotný počet chyb není až tak důležitý a jako metrika moc význam nemá, ale v kombinaci s dělením chyb podle závažnosti, komponenty, ve které se chyba nalézá, stavu opravy, a zda chybu objevili testeři nebo zákazník vzniká sada užitečných měřítek, jejichž porovnávání a poměry už poskytují hodnotné informace.

Je-li řečeno, že na projektu je 130 otevřených chyb a zbývají dva měsíce z celkově ročního projektu do ukončení, tak bez dalších informací by jakákoli představa o kvalitě aplikace byla téměř určitě zavádějící. Více nám napoví srovnání s podobným projektem, předchozím test reportem ze stejného projektu nebo rozdělení podle závažnosti a komponenty.

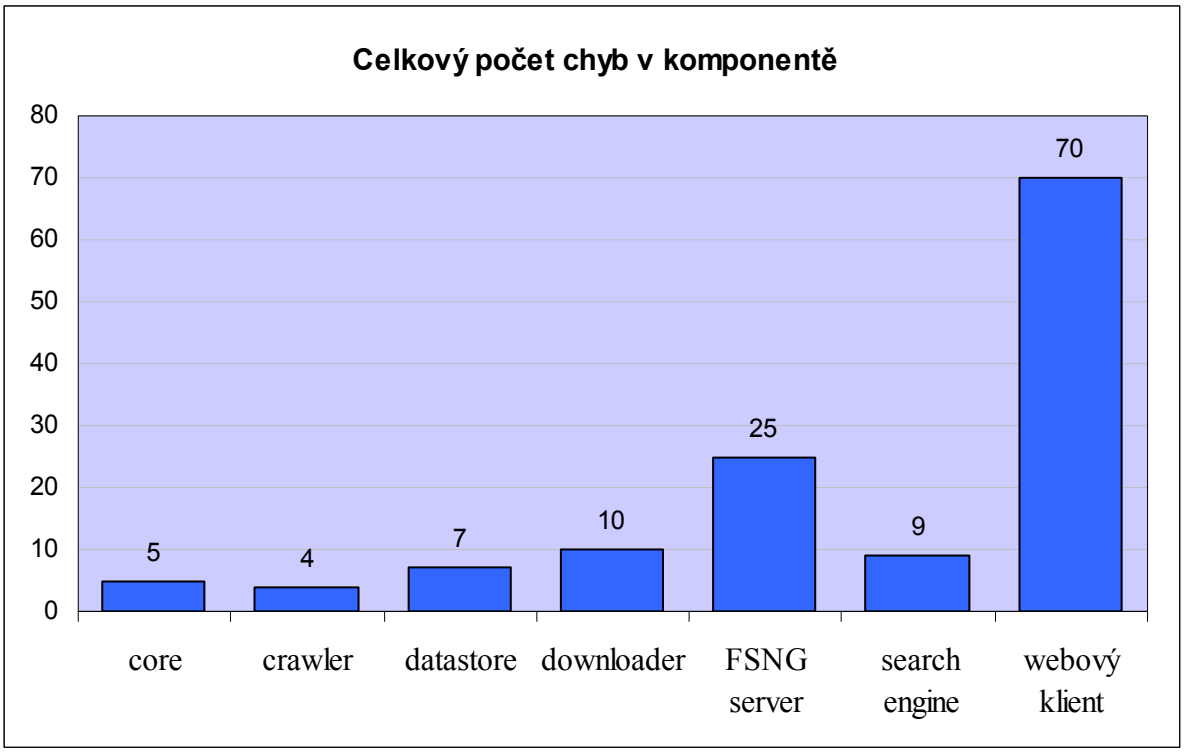
Z následujících statistik vidíme, že nejhůře na tom jsou komponenty webový klient, FSNG server, downloader a core. Chyby u webového klienta nejsou nijak závažné, ale pokud by ho viděl zákazník, pravděpodobně by byl kvalitou velmi znechucen. K tomu abychom odhadli nakolik velkým problémem jsou další tři komponenty, je zapotřebí vědět, jak jsou tyto komponenty velké a konkrétně co za chyby jsou u nich hlášeny.

Pokud by tedy manažer dostal jako odpověď na otázku, jak je na tom projekt s kvalitou, tabulku z níže uvedeného příkladu, snadno by si udělal představu o tom, jaké další otázky položit.

Příklad dělení chyb podle komponenty:

Komponenta	Závažnost A	Závažnost B	Závažnost C	Celkový počet chyb
core	4	1	0	5
crawler	0	4	0	4
datastore	1	6	0	7
downloader	4	5	1	10
FSNG server	5	17	3	25
search engine	3	4	2	9
webový klient	5	20	50	70

Tabulka 1: Dělení chyb podle komponenty a závažnosti



Obr. 8: Dělení chyb podle komponenty

Předchozí odpověď nadřízenému byla založena pouze na počtu chyb a jejich dělení. Kombinací jednoduchých cena, čas, počet chyb vznikají sofistikovanější metriky. Následující metriky uvádí Hatchesnová v [2]:

$$\text{Bug fix rate} = \frac{\text{počet opravených chyb}}{\text{počet nalezených chyb}} \text{ x } 100$$

- kolik procent chyb bylo opraveno. Namísto počtu opravených chyb se v praxi počítá s počtem uzavřených chyb, tedy s chybami, které byly úspěšně vyřešeny nebo byly zamítnuty.

$$\text{Efektivnost testu} = \frac{\text{chyby nalezené testem}}{\text{celkový počet nalezených chyb}} \text{ x } 100$$

- kolik procent z chyb v dané verzi bylo testem objeveno. Porovnává se efektivnost mezi testy nebo v čase a pokud test nepřináší dostatečné výsledky, vymění se za jiný.

$$\text{Test Effort Performance} = \frac{\text{opravené chyby nalezené během testu}}{\text{celkový počet nalezených chyb}} \text{ x } 100$$

- procentuelní přínos testu.

$$\text{Rychlost nalezení chyby} = \frac{\text{počet nalezených chyb}}{\text{počet hodin}}$$

- kolik chyb je průměrně nalezeno za hodinu. Počítá se stejně jako následující metriky v rámci jednotlivých verzí předaných k testování. Ze začátku projektu jsou obecně chyby nalézány rychleji než v závěrečné fázi.

$$\text{Cena nalezení chyby} = \frac{\text{náklady}}{\text{počet chyb}}$$

$$\text{Rychlost nalezení a zahlášení chyby} = \frac{\text{počet hlášení chyby}}{\text{počet hodin}}$$

$$\text{Cena hlášení chyby} = \frac{\text{náklady na hlášení chyby}}{\text{počet chyb}}$$

Cena nalezení a hlášení chyby = cena nalezení chyby + cena hlášení chyby

- sčítají se jen jednotky za stejnou verzi aplikace. Jakmile cena za nalezení a hlášení chyby stoupne nad cenu ponechání chyby v aplikaci, a nejedná se jen o přechodný stav, je lepší testování ukončit.

Metriky založené na testech

Systém projde testy za dvou různých situací, buď nejsou testy dostatečné nebo je připraven do provozu. Za předpokladu, že množina testů, kterými má aplikace projít, je známa, je možné sledovat, kolika procenty testů aplikace už dokáže projít. Protože testy stejně jako chyby se dají dělit do mnoha skupin, a ne u všech je použití této metriky vhodné, uvažují se v praxi pouze funkční testy aplikace. Jednotkou počtu testů bývá jeden test case nebo u automatizovaných testů jeho ekvivalent, případně test unita, pokud se jedná o testování jednotek.

Metriky založené na kódu

Metriky vzniklé na základě programovacího kódu tvořícího aplikaci už byly zmíněny u rozdělení testů, protože se odvíjejí od některého typu pokrytí kódu testy. Tyto metriky vypovídají o tom, kolik procent příkazů/hran/podmínek/cest bylo otestováno.

Na závěr kapitoly bych uvedla příklady možných odpovědí na otázku: „Kolik času zabere dokončení jednoho testovacího cyklu?“ Předpokládá se, že testovací případy pro verzi už jsou připravené. Tento příklad je upravenou verzí dotazu, který je původně součástí znalostního testu [40]. Smyslem je připomenout, že čím více informací je poskytnuto, tím kompletnější je odpověď. V tomto případě je nejlepší poskytnout přímo všechny odpovědi.

```
      Pokud předpokládáte, že pouze provedeme testy, zabere nám to 5 dní.
      Pokud předpokládáte, že při provádění testů nalezneme chyby a my jich
najdeme a prošetříme a nahlásíme 20, zabere nám to 7 dní.
      Pokud předpokládáte, že při provádění testů nalezneme 20 chyb,
prošetříme je, nahlásíme, pomůžeme programátorům s jejich opravou,
otestujeme opravu a budeme hledat 20 dalších chyb, které mohli vzniknout
při opravách, bude nám to trvat 10 dnů.
```

6. Testovací cyklus

Proces testování v průběhu vývoje softwaru má podobu neustále se opakujícího cyklu, který začíná předložením nové verze testovacímu týmu. Cyklus má stejnou formu, ať má nová verze podobu dokumentace nebo kódu, liší se ale obsahem cyklu, kterým je příprava či provedení testů.

Po předložení je zkontrolována správnost verze, zda si požadavky nebo jiné části dokumentace navzájem neodporují, kontroluje se testabilita aplikace, schopnost být otestován. V případě kódu, se provádí tzv. smoke test nebo taky build verification test, jehož smyslem je potvrzení, že byl kód jednotlivých komponent správně sestaven dohromady. Pokud komponenty k sobě nepasují, v aplikaci se najednou objeví velké množství chyb. Testovací tým by mohl strávit dny jejich hledáním a reportováním, a pak opětovným přetestováním a zavíráním hlášení, v okamžiku, kdy by dostal správnou verzi. Pokud se tedy objeví velké množství chyb v už hotových a otestovaných částech aplikace, měla by být nová verze vrácena konfiguračnímu managerovi nebo jinému členu týmu, který sestavuje kód aplikace a ne podstoupena zbytečnému podrobnému testování.

V dalším kroku probíhá samotná příprava či provedení testů podle připraveného plánu a hlásí nalezené chyby do bug report systému. V posledním kroku se sepisují výsledky testovacího cyklu, hodnotí testovací přístup, případně navrhují zlepšení pro příští cyklus.

Podle použité metodiky a náročnosti vývoje může být testovací cyklus zasazen do mnohem formálnějšího procesu. V rámci cyklu pak navíc probíhají následující úkoly:

- Dohodnutí cíle testů – Úkoly testovacího týmu v daném cyklu mohou být různé, od nalezení co největšího množství chyb přes odhad celkové kvality až po nalezení scénářů, které jsou vždy úspěšné a mohou být předvedeny zákazníkovi.
- Určení předmětu testů a hodnotících kritérií – Na kterou oblast aplikace bude testování zaměřené, jaké budou hodnotící kritéria bývá většinou předmětem testovacího plánu, který se ale vztahuje na celý proces testování, v rámci cyklu probíhá už jen ujasnění na základě cílů stanovených pro daný cyklus.
- Příprava testovacího prostředí – Před každým spuštěním testů je nezbytné připravené stabilní prostředí, které není ovlivněno předchozími testy. V rámci nového cyklu toto prostředí může být změněno, aby vyhovovalo nové verzi předané k otestování.
- Analýza výsledků a reportování – – Formální verze cyklu končí sepsáním dříve zmíněného formálního test report dokumentu.

Obsah celé příručky

Část I. – Úvod.....Chyba! Záložka není definována.
1. Historie.....Chyba! Záložka není definována.
2. Základní kameny.....Chyba! Záložka není definována.
3. Webové aplikace.....Chyba! Záložka není definována.
4. Charakteristiky testování a vývoje webových aplikací.....Chyba! Záložka není definována.
5. Metodiky pro web.....Chyba! Záložka není definována.
5.1 Klasické metodiky a web ..... **Chyba! Záložka není definována.**
5.2 Agilní vývoj..... **Chyba! Záložka není definována.**
5.3 Další metodiky ..... **Chyba! Záložka není definována.**
Část II. – Základy testování .....Chyba! Záložka není definována.
6. O čem je testování .....Chyba! Záložka není definována.
6.1 Definice ..... **Chyba! Záložka není definována.**
6.2 O čem je testování ..... **Chyba! Záložka není definována.**
7. Chyby.....Chyba! Záložka není definována.
7.1 Motivace – některé známé chyby ..... **Chyba! Záložka není definována.**
8. Testovací tým.....Chyba! Záložka není definována.
8.1 Vlastnosti testera ..... **Chyba! Záložka není definována.**
8.2 Pozice v testovacím týmu..... **Chyba! Záložka není definována.**
8.3 Tester versus programátor..... **Chyba! Záložka není definována.**
9. Kategorie testů.....Chyba! Záložka není definována.
9.1 Statické a dynamické testování ..... **Chyba! Záložka není definována.**
9.2 Černá a bílá skříňka..... **Chyba! Záložka není definována.**
9.3 Automatické a manuální testovaní ..... **Chyba! Záložka není definována.**
9.4 Stupně testování ..... **Chyba! Záložka není definována.**
9.5 Pokrytí testy..... **Chyba! Záložka není definována.**
9.6 Dimenze kvality ..... **Chyba! Záložka není definována.**
10. Dokumentace .....Chyba! Záložka není definována.
10.1 Nejdůležitější dokumenty podle praxe..... **Chyba! Záložka není definována.**
10.2 Nejdůležitější dokumenty podle standardu ..... **Chyba! Záložka není definována.**
10.3 Testovací nápady..... **Chyba! Záložka není definována.**
10.4 Reportování chyb ..... **Chyba! Záložka není definována.**
10.5 Metriky..... **Chyba! Záložka není definována.**
11. Testovací cyklus.....Chyba! Záložka není definována.
Část III. – Testy webových aplikací.....Chyba! Záložka není definována.
12. Testy a techniky.....Chyba! Záložka není definována.
12.1 Funkční testy: ..... **Chyba! Záložka není definována.**
12.2 Testy použitelnosti ..... **Chyba! Záložka není definována.**
12.3 Testy spolehlivosti..... **Chyba! Záložka není definována.**
12.4 Výkonnostní testy..... **Chyba! Záložka není definována.**
12.5 Testy podpory..... **Chyba! Záložka není definována.**
12.6 Bezpečnostní testy..... **Chyba! Záložka není definována.**
12.7 Další ..... **Chyba! Záložka není definována.**
13. Používání nástrojů.....Chyba! Záložka není definována.
14. Automatizace funkčních testů .....Chyba! Záložka není definována.
14.1 Canoo WebTest..... **Chyba! Záložka není definována.**

14.2 JWebUnit ..... **Chyba! Záložka není definována.**
14.3 Rational Functional Tester ..... **Chyba! Záložka není definována.**
15. Automatizace výkonnostních testů ..... Chyba! Záložka není definována.
15.1 Apache JMeter ..... **Chyba! Záložka není definována.**
15.2 Grinder ..... **Chyba! Záložka není definována.**
15.3 LoadRunner ..... **Chyba! Záložka není definována.**
16. Bezpečnost webových aplikací ..... Chyba! Záložka není definována.
16.1 Běžné bezpečnostní problémy ..... **Chyba! Záložka není definována.**
16.2 Dělení bezpečnostních problémů ..... **Chyba! Záložka není definována.**
16.3 Proces testování bezpečnosti ..... **Chyba! Záložka není definována.**
16.4 Nástroje ..... **Chyba! Záložka není definována.**
Část IV. – Praktický příklad ..... Chyba! Záložka není definována.
17. Příklad testovací plánu ..... Chyba! Záložka není definována.
1) Testovaná aplikace ..... **Chyba! Záložka není definována.**
2) Cíl testování ..... **Chyba! Záložka není definována.**
3) Testovací přístup ..... **Chyba! Záložka není definována.**
4) Kritéria ne/připravenosti k předání ..... **Chyba! Záložka není definována.**
5) Zdroje ..... **Chyba! Záložka není definována.**
6) Role ..... **Chyba! Záložka není definována.**
18. Příklad testovacího scénáře ..... Chyba! Záložka není definována.
19. Příklad test result dokumentu ..... Chyba! Záložka není definována.
1) Souhrnné zjištění ..... **Chyba! Záložka není definována.**
2) Testy dokumentace ..... **Chyba! Záložka není definována.**
3) Odhad stavu komponent ..... **Chyba! Záložka není definována.**
4) Příprava testovacích scénářů ..... **Chyba! Záložka není definována.**
5) Zhodnocení použitého přístupu ..... **Chyba! Záložka není definována.**
Seznam obrázků ..... Chyba! Záložka není definována.
Zdroje ..... Chyba! Záložka není definována.
Přílohy ..... Chyba! Záložka není definována.

Zdroje

Literatura

[1] Beck, Kent: *Extrémní programování*, Grada Publishing, Praha, 2002

[2] Hutcheson, Marnie L.: *Software Testing Fundamentals: Methods and Metrics*, John Wiley & Sons, 2003

[3] IBM, Rational software: *TST170 Principles of Software Testing for Testers*, v 2002.05.00, Instructor Guide

[4] Kadlec, Václav: *Agilní programování: Metodiky efektivního vývoje softwaru*, Computer Press, Brno, 2004

[5] Kunský, František: *Testování software a testovací role*, diplomová práce, VŠE, 2005

[6] Microsoft ACE Team: *Výkonnostní testování webových aplikací .NET* , Grada, 2004

[7] Patton, Ron: *Testování softwaru*, Computer Press, Brno, 2002

[8] Pavelka, Jan: *Zajištění jakosti projektů*, materiály k předmětu Softwarové inženýrství (sweng56.ppt), KSI MFF UK, 2005

[9] Pouzar, Lukáš: *Automatizované testování*,bakalářská práce, VŠE, 2007

[10] Weinberg, Gerald. M.: *Quality Software Management: Volume 1, Systems Thinking*, Dorset House Publishing Company, Incorporated, 1991

[11] Unicorn: *General test ideas,* kód artefaktu: USO.UES.TES/UES.TES.GTI, 2006

Zdroje z Internetu

[12] Abran, Alain; Moore, James W.: *SWEBOK, Guide to the Software Engineering Body of Knowledge*, IEEE Computer Society, 2004 Version
Dostupné z WWW:
www.inf.ed.ac.uk/teaching/courses/seoc/2006_2007/resources/SWEBOK_Guide_2004.pdf
Datum čerpání zdroje: 10.3.2008

[13] Als, Adrian; Greenidge, Charles: *The waterfall model,* 2005
Dostupné z WWW:
http://scitec.uwichill.edu.bb/cmp/online/cs22l/waterfall_model.htm
Datum čerpání zdroje: 7.2.2008

[14] Andrews, Mike: *How to break web software,* Google TechTalks, 2006
Dostupné z WWW:
http://video.google.com/videoplay?docid=5159636580663884360
Datum čerpání zdroje: 10. 11. 2007

[15] Aston, Philip; Fitzgerald, Calum: *The Grinder*
Dostupné z WWW:
http://grinder.sourceforge.net
Datum čerpání zdroje: 24.7.2008

[16] Bach, James; Bolton, Michael: *Rapid Software Testing,* v2.1.3, Satisfice, Inc., 2007
Dostupné z WWW:
http://www.satisfice.com/rst.pdf
Datum čerpání zdroje: 9. 4. 2008

[17] Canoo Engineering AG: *Canoo WebTest*
Dostupné z WWW:
http://webtest.canoo.com/
Datum čerpání zdroje: 2.7.2008

[18] Cornett, Steve: *Code Coverage Analysis*, Bullseye Testing Technology
Dostupné z WWW:
http://www.bullseye.com/coverage.html
Datum čerpání zdroje: 3.4.2008

[19] Dijkstra, Edsger W.: *The Humble Programmer*,
Dostupné z WWW:
http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF
Datum čerpání zdroje: 25.2.2008

[20] Faigl, Jan: *Úvod do softwarového inženýrství,* (lecture11), ČVUT
Dostupné z WWW:
http://lynx1.felk.cvut.cz/mep/files/slides/lecture11.pdf
Datum čerpání zdroje: 1.4.2008

[21] gantthead.com: *Process/Project WWP - webWAVE Development Process*
Dostupné z WWW:
http://www.gantthead.com/process/processMain.cfm?ID=2-8435-2
Datum čerpání zdroje: 3.3.2008

[22] Hailpern, Brent T.; Santhanam, Padmanabhan: *Software debugging, testing and verification*. IBM Systems Journal, 2002, roč. 41, č. 1.
Dostupné z WWW:
http://www.research. ibm.com/journal/sj/411/hailpern.pdf
Datum čerpání zdroje: 9.3.2008

[23] Churý, Lukáš: *Z černého trhu na trh volný,* programujte.com, 2005
Dostupné z WWW:
http://programujte.com/index.php?akce=clanek&cl=2005102101-z-cerneho-trhu-na-trh-volny
Datum čerpání zdroje: 7.11.2007

[24] IBM: *Rational Functional Tester*
Dostupné z WWW:
http://www-306.ibm.com/software/awdtools/tester/functional/index.html
Datum čerpání zdroje: 8.7.2008

[25] IBM Internet Security Systems: *X-Force® 2007 Trend Statistics*, IBM Global Services, Leden 2008
Dostupné z WWW:
http://www-935.ibm.com/services/us/iss/pdf/etr_xforce-2007-annual-report.pdf
Datum čerpání zdroje: 27.5.2008

[26] Jakarta-jmeter Wiki editors: *HowManyThreads*
Dostupné z WWW:
http://wiki.apache.org/jakarta-jmeter/HowManyThreads
Datum čerpání zdroje: 20.7.2008

[27] Jakarta-jmeter Wiki editors: *JMeterFAQ*
Dostupné z WWW:
http://wiki.apache.org/jakarta-jmeter/JMeterFAQ
Datum čerpání zdroje: 20.7.2008

[28] Larman, Craig; Basili, Victor R.: *Iterative and Incremental Development: A Brief History*
Dostupné z WWW:
http://www2.umassd.edu/SWPI/xp/articles/r6047.pdf
Datum čerpání zdroje: 4.3.2008

[29] Leveson, Nancy: *Medical Devices: The Therac-25*, 1995
Dostupné z WWW:
http://sunnyday.mit.edu/papers/therac.pdf
Datum čerpání zdroje: 18.3.2008

[30] Meier, J.D.; Farre, Carlos a kol: P*erformance Testing Guidance for Web Applications*, patterns & practices, Microsoft Corporation, 2007
Dostupné z WWW:
http://msdn.microsoft.com/en-us/library/bb924375.aspx
Datum čerpání zdroje: 18.7.2008

[31] Pavlíčková, Jarmila; Pavlíček, Luboš: *Nástroje na zvyšování kvality kódu*
Dostupné z WWW:
http://honor.fi.muni.cz/tsw/2004/226.pdf
Datum čerpání zdroje: 8.6.2008

[32] PayScale: *Salary Survey Report for Job: Software Developer, Web Applications*
Dostupné z WWW:
http://www.payscale.com/research/US/Job=Software_Developer%2c_Web_Applications/Salary
Datum čerpání zdroje: 25. 3. 2008

[33] PayScale: *Salary Survey Report for Job: Software Engineer / Developer / Programmer*
Dostupné z WWW:
http://www.payscale.com/research/US/Job=Software_Engineer_%2f_Developer_%2f_Programmer/Salary
Datum čerpání zdroje: 25. 3. 2008

[34] PayScale: *Salary Survey Report for Job: Test / Quality Assurance (QA) Engineer (Computer Software)*
Dostupné z WWW:
http://www.payscale.com/research/US/Job=Test_%2f_Quality_Assurance_(QA)_Engineer_(Computer_Software)/Salary
Datum čerpání zdroje: 25. 3. 2008

[35] Potter, Bruce; McGraw, Gary: *Software Security Testing*, THE IEEE Computer Society, září/říjen 2004
Dostupné z WWW:
http://www.cigital.com/papers/download/bsi4-testing.pdf
Datum čerpání zdroje: 6.5.2008

[36] Samurin, Alex: *Orthogonal Array Testing*, leden 2006
Dostupné z WWW:
http://www.geocities.com/xtremetesting/OrthogonalArrayTesting.html
Datum čerpání zdroje: 5.4.2008

[37] Schooff, Peter: *Does the Security Industry Have a Future?*
Dostupné z WWW:
http://www.schneier.com/news-060.html
Datum čerpání zdroje: 20.5.2008

[38] Simo, Ben: *Testing Lessons from Beer*
Dostupné z WWW:
http://testertested.qualityfrog.com/TLFB.mp3
Datum čerpání zdroje: 23.3.2008

[39] Soundararajan, Pradeep: *Automation replaces humans - The truth about what kind of humans it replaces*
Dostupné z WWW:
http://testertested.blogspot.com/2008/03/automation-replaces-humans-truth-about.html
Datum čerpání zdroje: 1. 5. 2008

[40] Soundararajan, Pradeep: *The most challenging software testing quiz*
Dostupné z WWW:
http://testertested.qualityfrog.com/TMCSTQ.swf
Datum čerpání zdroje: 9. 12. 2007

[41] SourceForge: *JWebUnit*
Dostupné z WWW:
http://jwebunit.sourceforge.net
Datum čerpání zdroje: 7.7.2008

[42] The Open Web Application Security Project
Dostupné z WWW:
http://www.owasp.org
Datum čerpání zdroje: 7.6.2008

[43] Vijayaraghavan, Giri; Kaner, Cem: *Bug Taxonomies: Use Them to Generate Better Tests*, STAR EAST 2003
Dostupné z WWW:
http://testingeducation.org/a/bugtax.pdf
Datum čerpání zdroje: 20.5.2008

[44] Ward, Stan; Kroll, Per: *Building Web Solutions with the Rational Unified Process: Unifying the Creative Design Process and the Software Engineering Process*, Context Integration, Rational Software, 1999
Dostupné z WWW:
http://www.dcc.uchile.cl/~luguerre/cc61j/recursos/76.pdf
Datum čerpání zdroje: 20.3.2008

[45] Wikipedia contributors: *Mars Polar Lander*, Wikipedia, The Free Encyclopedia
Dostupné z WWW:
http://en.wikipedia.org/wiki/Mars_Polar_Lander
Datum čerpání zdroje: 22. 3. 2008

[46] Wikipedia contributors: *MIM-104 Patriot*, Wikipedia, The Free Encyclopedia
Dostupné z WWW:
http://en.wikipedia.org/wiki/MIM-104_Patriot#Failure_at_Dhahran
Datum čerpání zdroje: 22. 3. 2008

[47] Wikipedia contributors: *Web application*, Wikipedia, The Free Encyclopedia
Dostupné z WWW:
http://en.wikipedia.org/wiki/Web_application
Datum čerpání zdroje: 12.11.2007