Rizika při vývoji a kvalita software
Risks of software development

Martin Vonka

Abstrakt: Článek definuje rizika spojená s vývojem softwaru, na která je nutné dát při řízení softwarového projektu pozor. Pro minimalizaci rizik spojených s vývojem software jsou zde představeny používané metody vývoje a testování software. Pro každý projekt je nutné zvolit vhodnou metodu řízení a testování. Článek rozebírá specifika jednotlivých metod vývoje s ohledem na řízení projektu a včasnou predikci potenciálních hrozeb. Testování je zde chápáno jako ucelený proces, který má jasně definované vstupy a výstupy. Pro sledování kvality vývoje jsou zavedeny metriky, které je možné sledovat v čase, a vývojový projekt tak lze vyhodnotit z hlediska kvality produkovaného kódu a spolehlivosti vlastní aplikace.

Klíčová slova: vývoj software, testování software, rizika při vývoji, analýza rizik

Abstract: The article defines the risks associated with software development, this is necessary to be focused on during the software projects. To minimize the risks associated with software development are presented the methods of development and testing software. For each project, it is necessary to choose an appropriate method of management, and testing. The paper analyzes the specifics of each method development with regard to project management and prediction of potential threats. Testing is like a holistic process, which has clearly defined inputs and outputs. For quality monitoring developments are introduced metrics that can be monitored over time and evaluate and development project in terms of quality and reliability of the code produced.

Key words: software development, software testing, risks in software development, risks analysis

JEL klasifikace: O30

Úvod

Vývoj software se v posledních desetiletích posunul od ad hoc vývoje k rigidním a agilním přístupům vývoje používaným dnes. Jednou ze zásadních změn ve vývoji je zaměření se na rizika projektu a jejich eliminaci. Risk management je proto velmi používané slovo a seznam rizik existuje v každém větším softwarovém projektu, většinou proto, že jej vyžadují firemní procesy nebo zákazník. Rizikům, která dopředu známe, můžeme lépe čelit. Velice důležité je správně je identifikovat; pro každé je třeba určit, jak významně ohrožuje projekt a jak jej můžeme minimalizovat.

Vyvíjet a prodávat kvalitní software je cílem každé vývojářské společnosti. Míra kvality software závisí na způsobu řízení jeho vývoje, zejména však na důkladnosti s jakou je testován před nasazením do provozu. Kvalita je v podstatě stupněm uspokojení potřeb uživatele a má zásadní vliv na provoz aplikace a její přínos uživateli. Různí uživatelé mají různé potřeby a provozují aplikace s různým stupněm náročnosti. To platí samozřejmě nejen pro oblast informatiky, ale i zcela obecně.

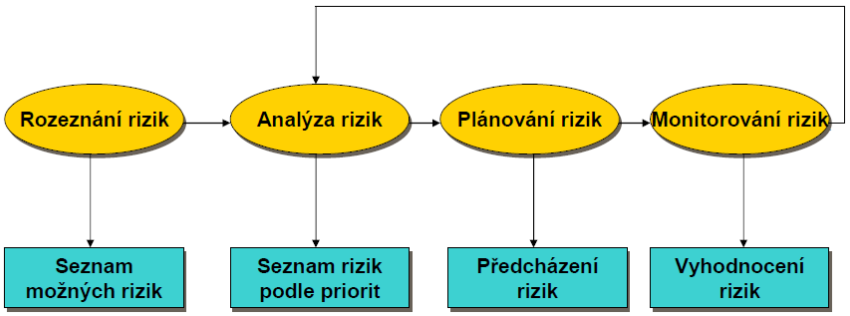
1. Rizika při vývoji software

Identifikace rizik

Prvním krokem při přípravě projektů je důkladná identifikaci rizik a správné zhodnocení z hlediska návrhu řešení, kapacit a časového harmonogramu.

Dalším důležitým krokem je nejen rizika správně identifikovat a formulovat, ale hlavně navrhnout akce na snížení jejich dopadu či úplné odstranění (samozřejmě cenově ospravedlnitelné, což je také velice důležité). Postup identifikace rizik je znázorněn na následujícím obrázku.

Obrázek 1 Postup identifikace rizik



Zdroj: Mannová, 2006

Rizika, která je třeba před začátkem softwarového projektu vyhodnotit:

a) Rizika velikosti projektu

- Odhad velikosti a náročnosti projektu (vzhledem k již realizovaným projektům).
- Množství znovupoužitého softwaru – má vliv na rychlost a chybovost produktu.

b) Rizika obchodního dopadu

- Jak produkt ovlivní zisk společnosti – je firma závislá jen na tomto produktu?
- Je dodací termín rozumný?
- Množství a kvalita dokumentace produktu, která musí být produkována a dodána zákazníkovi? Náklady na pořízení dokumentace.
- Cena za pozdní dodání, reklamace…

c) Rizika týkající se zákazníka

- Ví zákazník dobře, co chce? Byla provedena dostatečně podrobná analýza informačních potřeb?

d) Procesní rizika

- Jsou používány specifické metody pro softwarovou analýzu?
- Je pravidelně prováděna formální revize stavu projektu?

e) Technologická rizika

- Kompatibilita s jinými produkty.
- Zvolená technologie a model aplikace.

f) Rizika spojená s velikostí týmu a jeho zkušeností

- Mají lidé pracující na projektu odpovídající zkušenosti?
- Je jich dostatečný počet, tak aby byl projekt včas a úspěšně dokončen?

Základní rizikové komponenty:

- Rizika provedení – stupeň nejistoty, že produkt bude odpovídat požadavkům a bude vyhovovat zamýšlenému použití
- Rizika ceny – stupeň nejistoty, že bude dodržen rozpočet
- Rizika podpory – stupeň nejistoty, že software půjde snadno opravovat, upravovat a zlepšovat
- Rizika času – stupeň nejistoty, že časový harmonogram bude dodržen a že produkt bude dodán včas

K projektu je možno sestavit tabulku s hodnocením rizik. Hodnoty vychází ze zkušeností při vedení implementačních projektů ve společnosti NetGenium s.r.o.

Tabulka 1 Tabulka rizik projektu [Procházka, 2009]

Rizika	Kategorie	Pravděp.	Dopad	Řešení
Zákazník změní požadavky	zákazník	70%	2	domluva se zákazníkem
fluktuace členů týmu	tým	50%	1	noví pracovníci, dostatečná rezerva při sestavení týmu
špatný odhad velikosti projektu	produkt	30%	2	dostatečné rezerva při prvním návrhu
špatný odhad nákladů	produkt	30%	2	dostatečné rezerva při prvním návrhu
penále při pozdním odevzdání	obchod	20%	1	zaměstnat více pracovníků, použití časové rezervy nebo dohoda se zákazníkem nedorozumění, špatná komunikace mezi
členy týmu	tým	15%	2	dobrý výběr členů týmu
nedostatek zkušeností členu týmů	tým	10%	2	dobrý výběr členů týmu
nedostatečný počet členů týmů	tým	10%	1	vhodná velikost týmu při sestavení, rezerva v počtu členů týmu
vytvoření defektního produktu	tým	5%	4	pravidelné inspekce, revize, audity, testování
poškození dat	technika	5%	4	zálohování dat, použití diskových polí
nedostatečné školení	tým	5%	2	včasné a správné školení pracovníků
Dopad: 4 - katastrofický, 3 - kritický, 2 - marginální, 1 - zanedbatelný				

Zdroj: Procházka, 2009

Riziky řízený vývoj software je proaktivní, snažíme se identifikovat rizika co nejdříve. Důraz je kladen na přístup k rizikům. Cílem není pouhá identifikace události, která nás může ohrozit, ale návrh akce na její odstranění, na snížení dopadu na (peněžně, časově) akceptovatelnou úroveň, případně přesunutí rizika na externí subjekt (subdodavatel, zákazník) nebo definovávání náhradního řešení.

Řešení rizik

Iteraktivní projekt se skládá z řetězce iterací, které na sebe vážou určitá specifická rizika. Výstup předchozí iterace je vstupem pro iteraci následující. Jednotlivé rizikové

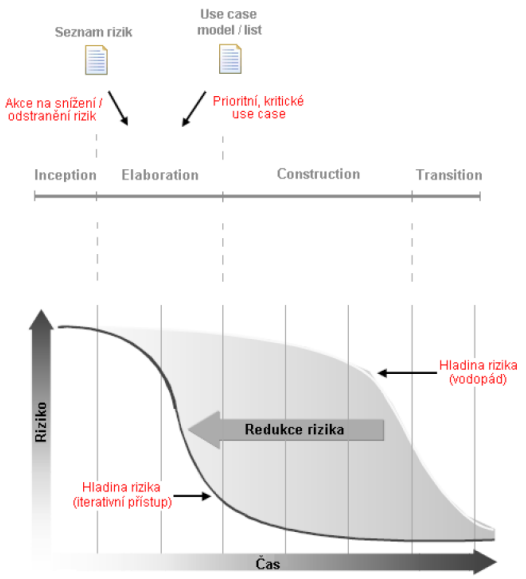
faktory tímto procesem zvyšují pravděpodobnost svého výskytu. Projekt jako pouhý sled iterací a návrh pro pouze aktuálně známé požadavky by však byl krátkozraký a vyžadoval by zásadní refaktoring v každé další iteraci. V rámci vývoje software musíme dělat jistý konceptuální návrh architektury (tím myslíme i implementaci – jediné ověření schůdnosti návrhu), která je nejvhodnější pro daný projekt. K dosažení výše zmíněných cílů nám slouží fáze, které seskupují iterace za různým účelem:

- **Inception** – zaměřena na nalezení shody na cílech projektu (vize), identifikace rizik, definice roadmap, vytvoření týmu.
- **Elaboration** – zaměřena na snížení rizik a implementaci architektury včetně kritických scénářů aplikace.
- **Construction** – zaměřena na inkrementální vývoj zbývajících scénářů aplikace na základě vytvořené architektury.
- **Transition** – zaměřena na předání výsledku zákazníkovi, akceptační testy, školení, doladění dokumentace.

V životním cyklu iterativního vývoje software je tedy pro řešení rizik určena tzv. Elaboration (Rozpracování) fáze. Seskupení iterací do fází má důležitý přínos v zaměření se na rizika a architekturu, nejedná se tedy o pozůstatek vodopádu, jak tyto fáze popisuje mnoho autorů.

Obr. 2 ukazuje, jakým způsobem a kdy se zaměřujeme na rizika. Prioritní akce definované v seznamu rizik musíme provést jako první, abychom snížili úroveň rizik na akceptovatelnou úroveň (viz křivka na obr. 2). Akce ze seznamu rizik a implementace kritických scénářů se tedy stanou cíli jednotlivých iterací seskupených do Elaboration fáze. Cílem je vyzkoušet technologii, pokusit se implementovat složité algoritmy, vyzkoušet různé druhy architektury, abychom případně mohli zasáhnout a zvolit technologii, jinou architekturu, abychom věděli, zda jsme pochopili doménu, zda je tým schopný pracovat v těchto podmínkách. V případě, že neexistují varianty, nelze pokračovat dále a tudíž jsme nuceni projekt ukončit. Toto díky riziky řízenému přístupu zjistíme velmi brzy a neúspěch projektu nás nestojí tolik. V úspěšném případě budeme mít na konci Elaboration fáze k dispozici implementovanou a otestovanou architekturu = část produktu (podle typu projektu od 20ti do 60ti % celkového řešení). Díky tomuto postupu můžeme v Construction fázi použít více vývojářů a vyvíjet zbytek produktu relativně bez překvapení.

Obrázek 2 Hladina rizika u iterativních projektů



Zdroj: Procházka, 2009

Minimalizace rizik

Některá rizika lze minimalizovat pomocí zavedených opatření nebo je lze přenést na subdodavatele. Zde jsou uvedeny příklady minimalizace rizik pro organizaci:

- **Havarijní plány:** Jsou to plány ošetření rizikových událostí sestávající se z akčních kroků, které je nutno přijmout jestliže riziková událost nastane.
- **Alternativní strategie:** Rizikovým událostem lze někdy předejít, změníme-li strategii řešení.
- **Rezervy:** Jsou to opatření, která mají zmírnit rizika nákladů a harmonogramu. Často jsou blíže specifikována účelem, to znamená, jaká rizika mají být zmírněna (např. provozní rezerva, rezerva na nepředvídané události, časová rezerva).
- **Obstarávání:** Některé činnosti mohou být méně rizikové, budou-li provedeny externí firmou, která má s nimi vetší zkušenosti.
- **Pojištění:** Pojistné smlouvy mohou odstranit nebo zmírnit některá rizika.

Vliv dokumentace na rizika projektu

Ač to mnohdy není na první pohled vidět, ucelená a přehledná dokumentace je jedním z klíčových faktorů úspěšnosti vývojových projektů. Dokumentace využívá nástroje, které popisují strukturu a funkce aplikace v požadované podrobnosti.

Kvalitní a dobře udržovaná dokumentace (v rozumné míře podrobnosti) eliminuje rizika spojená s fluktuací členů týmu. Dále pak odhalí vnitřní rozpory v projektu, které mohou vzniknout při zapracování dodatečných požadavků zákazníka. Požadavky jsou základním vstupem každého projektu vývoje softwaru a jako takové rozhodují o tom, co bude výstupem projektu. Vývojový tým by měl spolu se zadavatelem investovat přiměřené prostředky do jednoznačného a jasného popisu nově vznikajícího systému tak, aby vývojový tým měl dostatek srozumitelných podkladů pro svoji práci a aby při ukončení projektu došlo k bezproblémovému předání a akceptaci kvalitního softwaru.

Požadavky na změny lze uchovávat ve formě textové specifikace podle potřeb projektu (aby byly zasazeny do správného kontextu), ale zároveň je třídit, vyhledávat, reportovat, definovat a sledovat jejich metriky. Požadavky je možné provázat mezi sebou i na jednotlivé elementy návrhu a tím do budoucna významně zjednodušit zapracovávání změn a následnou úpravu dokumentace, která je nezbytná.

Základním principem je využití procesu konfiguračního a změnového řízení v celém životním cyklu projektu. Určení priorit změnových požadavků a návaznost požadavků na pracovní úkoly pomáhají plánovat práci a rovnoměrné delegovat úkoly v rámci vývojového týmu. Důležitá je i dostupnost funkcí potřebných pro konfigurační a změnové řízení přímo z vývojového prostředí.

Proč tedy používat přehledné informační systémy pro správu požadavků a dokumentace?

- Jednoznačně a kompletně definované požadavky minimalizují potřebu přepracování a snižují pozdější problémy s akceptací – zákazník si ujasní své požadavky při analýze informačních potřeb.

- Priorizací a jasným plánem v oblasti správy požadavků zefektivníme i navazující činnosti (návrh, vývoj, testování). Priorizace snižuje riziko nedodržení časového harmonogramu projektu; zákazník dostává v jednotlivých iteracích vývoje k připomínkování (testování) vždy klíčové funkce vyvíjené aplikace.

- Provázanost požadavků odhalí skutečné dopady zamýšlených nebo realizovaných změn a tím sníží rizika spojená s interními kolizemi v aplikaci

- Všichni členové týmu mají jednoznačně stanovené společné zadání odsouhlasené zadavatelem – minimalizuje se riziko dodání aplikace s jinými funkcemi než zákazník požadoval. Dokumentace také eliminuje rizika způsobená fluktuací členů projektového týmu.

- Zapracované požadavky je nutné zapracovat i do dokumentace aplikace. Zanedbáváním této povinnosti dochází k degradaci dokumentace. Její následná jednorázová oprava a doplnění jsou nákladnější a vzniká nebezpečí, že nebudou doplněny všechny zapracované požadavky.

Kvalita software

Softwarové produkty jsou velice různorodé a jen obtížně se u nich hledá metoda, kterou by bylo možné plošně ověřit kvalitu dodané aplikace. Typy požadavků na jakost vyjadřují různě v závislosti na účelu a způsobu nasazení software.

Oblasti požadavků na software jsou zároveň oblastmi, které je třeba vyhodnocovat z hlediska kvality jejich implementace. Oblasti lze rozdělit například do následujících skupin (podle [wiki 1] a [wiki 2]):

- FUNKČNOST, jako schopnost produktu zabezpečit požadované funkce (důležité je, zda tyto funkce jsou zabezpečeny, nikoliv jak a za jakou cenu). Funkčnost musí být správně definována v úvodní analýze, která předchází realizaci software. Pokud tato analýza není provedena správně, může být i správně (dle požadavků a dokumentace zrealizovaný software) nefunkční.

- BEZPEČNOST – uživatelsky autorizace a autentizace, technicky odolnost proti chybným vstupům a jejich zneužití, např. k cíleným útokům.

- BEZPORUCHOVOST, jako schopnost produktu zajistit za daných podmínek požadovanou úroveň výkonu a poskytovaných služeb. Zejména s ohledem na požadavky technologické odstávky a servis aplikace.

- POUŽITELNOST, jako schopnost produktu být využíván při přiměřené míře úsilí potřebného na seznámení se s jeho možnostmi a jeho běžné provozování v daných podmínkách. Významným faktorem limitujícím použitelnost je úroveň zpracování uživatelského prostředí a způsobu ovládání. Použitelnost také významně ovlivňuje podpora pro aplikaci – dokumentace, helpdesk, poskytované školení a další služby pro uživatele.

- UDRŽOVATELNOST, jako schopnost produktu být v průběhu používání rozvíjen s cílem přizpůsobení požadavkům uživatele, které se mohou v průběhu času výrazně měnit. Na udržitelnost má významný vliv použitá technologie, tedy její možnosti nabídnout novou požadovanou funkcionalitu s rozumnými náklady a v rozumném čase.

- PŘENOSITELNOST, jako schopnost produktu spolupracovat na datové i procesní úrovni s jinými systémy, včetně těch, které pracují na jiných platformách (datových, softwarových, ale i hardwarových).

- LOKALIZOVATELNOST – snadný převod do jiných jazyků. [wiki 2]

Proč se kvalitou software zabývat?

Nekvalitně provedený software je například u podnikových informačních systémů velkou hrozbou pro společnost, která jej zavádí. Většinou se jedná o software, který podporuje základní procesy společnosti, na kterých je společnost přímo závislá. Chyby v analýze či vlastní implementaci software proto mohou být fatální. [Řepa, 1999] Existují čtyři základní důvody, proč je problematika kvality programů v současnosti tak aktuální:

- Software se stal velmi rozšířeným zbožím a uživatelé by měli být chráněni před nekvalitními produkty. Zároveň kvalitní (ověřený) software umožňuje uživateli plně důvěřovat softwarovému dílu, které si pořizuje a které se pro něj stává „black boxem" s požadovanými funkcionalitami.

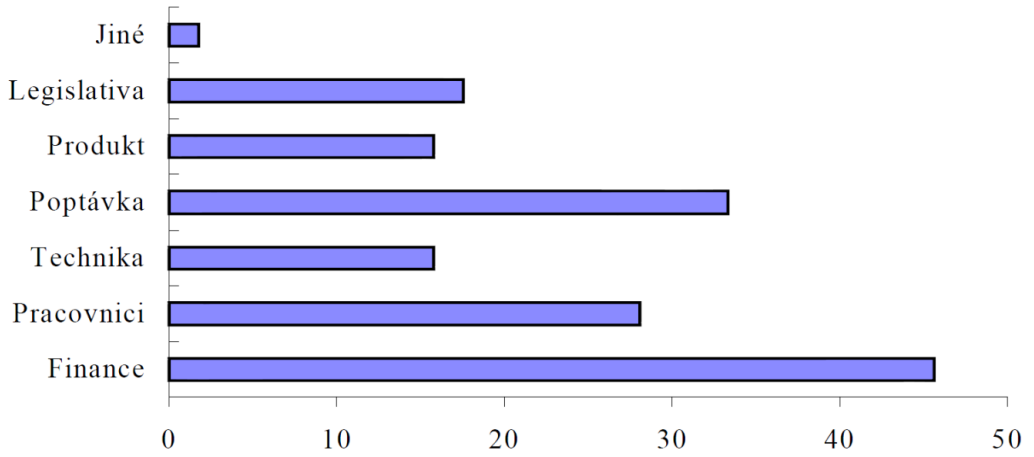
- Do software se investují obrovské finanční částky a z celospolečenského i tržního hlediska je potřeba zajistit jejich efektivní zhodnocení.

- Stále více se používá software v aplikacích pro automatické řízení procesů, kde chyby v programovém vybavení mohou mít veliké, často katastrofální důsledky (řízení jaderných reaktorů, řízení zdravotnických zařízení, navigace letadel apod.). V poslední době je příkladem neúspěšný start sondy fobos-grunt, která se v důsledku chyby software nedostala na plánovanou oběžnou dráhu a následně na svou misi k Marsu.

- Bezpečnost software – tento aspekt softwarového díla je velice diskutovaným a citlivým tématem. Každá aplikace, která je využívána pro řízení nebo se stává strategickou aplikací, by měla projít testy zabezpečení (penetrační testy, odolnost proti útokům…)

Výzkum, který byl prováděn u nás v rámci projektu Evropské unie SPIRE [Douček, 2000], vykázal následující současné překážky ve zvyšování jakosti produkovaného software:

Obrázek 1 Největší současné problémy malých a středních firem v procentech



Zdroj: Douček, 2000

Zajímavé také je, že 14 % českých firem (dotazníky byly rozeslány 850 firmám, z nichž 108 vrátilo vyplněný dotazník) stále nepovažuje oblast řízení kvality za důležitou. Celkem 47 % firem uvedlo, že nemají dostatek informací o ISO 9001 pro zavedení systému řízení jakosti. Nezanedbatelný je i počet softwarových firem, které uvedly, že kvalita se nedá v jejich oboru aplikovat – celkem 12 %. Pouze 8 % firem uvedlo, že má certifikát svého systému řízení jakosti.

2. Stručný přehled metod vývoje

Vodopádový přístup

Vodopádový model je sekvenční vývojový proces, ve kterém je na vývoj nahlíženo jako na neustále se svažující tok (jako když teče vodopád) fázemi analýzy požadavků, návrhu, implementace, testování (validace), integrace a údržby. Jako první formální popis vodopádového modelu je často citován článek, který publikoval v roce 1970 Winston W. Royce (1929 – 1995) [Řepa, 1999]

Hlavní principy vodopádového přístupu:

- Projekt je rozdělen na fáze jdoucí postupně za sebou, přičemž některé se mohou překrývat.
- Důraz je kladen na plánování, časové rozvrhy, termíny, rozpočty a realizaci celého systému najednou.

Přísná kontrola je udržována po celou dobu životnosti projektu prostřednictvím využití rozsáhlých písemných dokumentů, jakož i prostřednictvím formálních revizí a schvalování uživatelem (signoff) na konci většiny fází a vstupy od managementu informačních technologií před začátkem další fáze.

Prototypový přístup

Vývoj pomocí vytváření prototypů je takový přístup k vývoji software, kde dochází k vývoji neúplných verzí software, tzv. prototypů. [Řepa, 1997]

Základní principy prototypového přístupu:

- Není samostatným a kompletním přístupem metodiky vývoje, ale spíše přístupem k jednotlivým částem větších tradičních metodik vývoje software (tj. přírůstková metoda, spirála, nebo RAD - Rapid application development).
- Snaha snížit nebezpečí projektových rizik rozdělením projektu na menší části a zjednodušit tak možnost změn v průběhu procesu vývoje.
- Uživatel je zapojen v celém procesu vývoje, což zvyšuje pravděpodobnost přijetí konečné implementace uživatelem.
- Malé ukázky systému jsou vyvíjeny iterativním procesem, dokud se prototyp nevyvine tak, že splňuje požadavky uživatele.
- Většina prototypů je sice vyvíjena s tím, že budou vyřazeny, ale v některých případech je možné pokročit od prototypu k funkčnímu systému.
- Aby se předešlo vývoji, který řeší jiný problém, než bylo zadáno, je třeba pochopit základní business problematiku.

Inkrementální (přírůstkový) přístup

Inkrementální přístup je vhodný pro kombinaci sekvenčních a iteračních metodik softwarového vývoje. Cílem je omezit projektová rizika rozdělením projektu na menší segmenty a zjednodušit možnost zavedení změn během procesu vývoje. [wiki 1]

Základní principy inkrementálního přístupu:

- Jsou prováděny série malých vodopádů, kde každý vodopád je prováděn pro malou část systému a je dokončen před pokračováním na další přírůstek
- Obecné požadavky jsou definovány dříve, než se přikročí k evolučnímu vývoji pomocí malých vodopádů pro jednotlivé přírůstky systému

- Prvotní koncept, analýza požadavků, design architektury a systémové jádro jsou definovány vodopádovým přístupem, následuje iterativní prototypový přístup, který vrcholí instalací konečného prototypu jako funkčního systému.

Spirálový přístup

Spirálový přístup je proces vývoje software, který kombinuje prvky designového přístupu a prototypového přístupu tak, aby zkombinoval výhody obou konceptů shora-dolů (prototypování) a zdola-nahoru (designování). [wiki 1]

Základní principy spirálového přístupu:

- Zaměřuje se na analýzu rizik a minimalizaci projektových rizik rozdělením projektu na menší segmenty a umožněním změn během procesu vývoje. V průběhu vývoje je také možné vyhodnocovat rizika a zvažovat další pokračování projektu v průběhu životního cyklu.
- Každý cyklus spirály spouští stejný sled kroků pro každou část produktu a pro každou úroveň elaborace (od konceptuálních dokumentů až po programování jednotlivých programů).
- Během každého cyklu spirály jsou tak spouštěny čtyři základní fáze (kvadranty):
  - Analýza – stanovení cílů, alternativ a rozsahu iterace
  - Vyhodnocení – vyhodnocení alternativ, identifikace a řešení rizik
  - Vývoj – vývoj produktu a kontrola očekávaných výsledků
  - Plánování – plán pro příští iteraci

V počátku každého cyklu se identifikují zainteresované subjekty a jejich podmínky kladené na úspěch iterace, na konci každého cyklu se provádí revize a předání.

Rapid Application Development (RAD) přístup

Rapid Application Development (RAD) je takový přístup k vývoji software, který zahrnuje iterativní vývoj prototypů. RAD byl původně použit k označení procesu vývoje software, který zavedl James Martin v roce 1991. [wiki 1]

Základní zásady RAD:

- Hlavním cílem rychlého vývoje a zavedení vysoce kvalitních systémů při relativně nízkých investičních nákladech.
- Snaha snížit nebezpečí projektových rizik rozdělením projektu na menší segmenty a umožněním změn během procesu vývoje.
- Klíčový důraz je kladen na naplňování business potřeb, přičemž na technologickou nebo technickou dokonalost je kladen menší důraz.
- Řízení projektu zahrnuje stanovení priorit vývoje a stanovení dodacích lhůt nebo časových rámců. Pokud se projekt zpožďuje, postupuje se spíše snižováním požadavků než posouváním termínu, aby byl termín naplněn.
- Obecně zahrnuje techniku JAD (Joint Application Development, Joint Application Design), který intenzivně zapojuje uživatele do procesu návrhu systému, a to buď prostřednictvím shody na strukturovaných seminářích nebo pomocí elektronicky vedené interakce.
- Aktivní zapojení uživatelů je nutností.
- Na rozdíl od jednorázových prototypů vzniká produkční software iterativně.
- Je vytvářena dokumentace potřebná pro snazší budoucí vývoj či údržbu.
- V rámci těchto postupů je možné použít standardní systémovou analýzu a design.

Jakost softwaru

Objektivní skutečnost je taková, že problematika kontroly tvorby a vyhodnocení kvality software je podstatně složitější problém než u většiny běžných výrobků a služeb (např. strojírenských součástek, chemických směsí apod.). Software je nehmotné zboží! Současné programové produkty jsou nesmírně komplikované. Neexistují jednoduché a efektivní metody pro vyhodnocení kvalitního software. Tuto skutečnost respektoval i proces certifikace řízení jakosti podle norem ISO [Harazim, 2008].

Softwarové inženýrství v současné době nemůže prokázat správnost programu dokazováním. Může pouze dokazovat testováním, že program určitou chybu má. Tato situace je z hlediska hodnocení kvality programu velmi nepříjemná. O to více však klade důraz a vysoké nároky na postup testování software.

Kvalita software a rychlost vývoje

Většina managerů se snaží zkrátit dobu vývoje omezením času věnovanému inspekcím návrhu kódu, testováním apod. Zejména testování bývá často obětováno, protože je na konci vývojového cyklu. Podle dostupných studiích tento chybný přístup celkovou dobu vývoje naopak prodlužuje. Jinými slovy, kvalitnější produkt bude dříve dokončen. Pokud obětujeme kvalitu, vývoj se tím prodlouží a prodraží.

Rozbory potvrzují, že kvalitu testovacího procesu ovlivňují zejména následující faktory [Harazim, 2008]:

- Množství testovacích dat
- Kvalita testovacích dat
- Použité metody testování
- Použité nástroje při testování
- Znalosti testovacích pracovníků

Proto součástí kvalitního vývoje softwaru jsou dobře zpracované plány testů, které určují Jak, Co, Kdy, Kým a Čím testovat. Proces testování se provádí nahodile a pracovníci většinou v posloupnosti testů improvizují. Zřídkakdy se přistupuje systematicky k celému procesu testování programů, a to i přesto, že špatný postup testování zvyšuje existenční rizika softwarových projektů.

3. Nástroje na řízení kvality software

Základní pojmy

- Verifikace je ověření, zda produkt dané fáze vývoje SW odpovídá konceptuálnímu modelu (např. zda kód odpovídá návrhu apod.) – tj. odpověď na otázku: vytvářím produkt správně?
- Validace je vyhodnocení SW na konci procesu vývoje SW, abychom zajistili splnění požadavků na SW – tj. odpověď na otázku: vytvářím správný produkt?
- Automatická statická analýza – používá se nejčastěji pro kontrolu zdrojových textů SW systému, případně kontrolu modelů apod.
- Inspekce – ruční kontrola artefaktů SW procesu, typicky prováděná skupinou 3 až 5 lidí.

- Testování – spouštění programu s takovými daty, abychom v programu odhalili defekty.
- Omyl (error) – chybná úvaha nebo překlep vývojáře, vede k jednomu nebo více defektům.
- Defekt (fault, bug, defect) – rozdíl mezi chybným programem a jeho správnou verzí.
- Symptom (symptom, failure, run-time fault) programu – pozorovatelné chybné chování programu. Defekt se při konkrétním běhu může projevit žádným, jedním nebo více symptomy.

Automatická statická analýza

Používají se programy pro automatickou kontrolu modelů nebo zdrojových textů. Například překladač jazyka Java obsahuje silnou typovou kontrolu. Pro slabě typované jazyky (např. C) se používají statické analyzátory (code checkers). Detekuje neinicializované proměnné, odchylky od standardů apod.

Mnoho programů používá pro detekci podezřelých míst heuristiky. Nevýhodou je, že pokud zdrojový text neodpovídá heuristikám zabudovaným v programu, mohou tyto programy produkovat falešná chybová hlášení.

Inspekce a procházení programů

Používají se při přezkoumávání kódu (inspekce kódu se provádějí před testováním programu, inspekcím by měla předcházet statická analýza). Zahrnují čtení dokumentu nebo programu týmem např. 3 nebo 4 lidí- jeden z nich autor) s cílem nalézt defekty (nikoli jejich řešení).

Výhody inspekcí

Bývají poměrně efektivní (typicky najdou 30 % až 70% defektů detailního návrhu kódu). Úsilí bývá přibližně poloviční oproti ekvivalentnímu otestování na počítači (na druhou stranu pokud máme testy již připravené, mohou běžet automaticky).

Cena opravy defektu bývá nižší než při testování na počítači (protože je známá přesná příčina defektu, zatímco testování na počítači najde pouze symptomy).

Nalézá jiné typy defektů než klasické testování, tj. je s ním komplementární (je vhodné provádět obojí).

Nevýhody:

Pro maximální efektivitu je třeba, aby s nimi tým získal zkušenost.

Testování

Spouštění programu se záměrem najít v něm defekty (tj. snažíme se, aby se projevily symptomy případných defektů). [wiki 2]

Existují dva základní způsoby testování – **„black box“** a **„white box“** testování.

Black box testování

Používají se také názvy: functional, data-driven, input-output driven testing.

Tester na program pohlíží jako na černou skříňku s danou specifikací, vnitřní struktura a vnitřní funkce programu ho nezajímají. Hledá případy, ve kterých se program nechová podle specifikace. Pro nalezení všech defektů by bylo nutné otestovat program se všemi možnými vstupy (platnými i neplatnými), což je prakticky nemožné (např. překladač jazyka C bychom museli otestovat se všemi platnými i neplatnými programy).

Víme, že úplné otestování programu je nemožné. Jak ale maximalizovat počet defektů nalezený konečným počtem testovacích případů? K programu už nemůžeme přistupovat čistě jako k černé skříňce, ale musíme učinit nějaké rozumné předpoklady o jeho vnitřním chování.

White box testování

Také: glass-box, clear-box, logic-driven testing. Testovací data se odvozují z programové logiky. Pro úplné otestování programu bychom potřebovali pomocí testovacích případů otestovat všechny možné logické cesty v programu (analogie otestování programu se všemi možnými vstupy).

Má dva zásadní problémy. Počet logických cest je i v malých programech příliš velký. I po otestování všech logických cest mohou v programu zůstat nenalezené defekty, protože některé logické cesty mohou chybět a protože nemusejí být nalezeny defekty citlivé na data.

Shrnutí metod

Při black-box i white-box testování se budou testovací případy skládat z popisu vstupních dat a z popisu správného výstupu pro daná vstupní data. Program nebo jeho část spustíme se vstupními daty, porovnáme předpoklad se skutečným výstupem (nejlépe automaticky). Testovací případy mají obsahovat platné i neplatné vstupy. Testovací případy je třeba uchovávat, protože je můžete znovu potřebovat (např. pro otestování programu po změně). Je nutné také zkontrolovat, zda program neprovádí nechtěné vedlejší efekty (zápisy do databáze apod.).

Návrh testovacích případů

Už jsme uvedli, že úplné otestování programu není možné, proto je pro testování velmi podstatný návrh efektivních testovacích případů. Klademe si otázku, jaká podmnožina všech testovacích případů má největší pravděpodobnost nalézt většinu defektů?

Náhodný výběr testovacích dat

První nápad – náhodně vybraná podmnožina všech možných vstupů. Pravděpodobně jedna z nejhorších možností, protože má malou pravděpodobnost být optimální podmnožinou nebo být alespoň blízká optimální podmnožině.

Použitelné metody budou kombinací myšlenek black box a white box testování. Existuje několik metodik, každá má své silné a slabé stránky – tj. každá detekuje/přehlédne jiné typy defektů, proto je dobré připravovat testovací případy pomocí více metod.

Rozdělení vstupů do ekvivalentních tříd

Dobrý testovací případ bude mít dvě vlastnosti. Bude vyvolávat co nejvíc vstupních podmínek a tím omezí celkový počet potřebných testovacích případů. Testovací případ by měl pokrývat určitou množinu vstupních hodnot. Množinu vstupů bychom měli rozdělit do tříd ekvivalence tak, abychom mohli rozumně předpokládat, že test nějaké reprezentativní hodnoty v dané třídě je ekvivalentní testu kterékoli další hodnoty.

Pokrytí kódu testovacími případy

Při white-box testování nás zajímá, do jaké míry testovací případy pokrývají zdrojový text programu. Jak už jsme si uvedli, otestovat všechny cesty v programu je obvykle neproveditelné, proto se o to nebudeme pokoušet. Praktické metody by měly testovat pouze rozumnou podmnožinu cest v programu.

Dobrým kritériem je pokrytí všech rozhodovacích příkazů tak, aby byly vykonány všechny jejich větve. Musíme vytvořit tolik testovacích případů, aby se v každém příkazu „if“ vykonala alespoň jednou větev při podmínce „false“ a alespoň jednou větev při podmínce „true“ atd.

Pro složitější podprogramy se vyplatí modelovat cestu podprogramem pomocí orientovaného grafu popisujícího možný tok řízení v podprogramu. Uzly reprezentují příkaz nebo část příkazu (přiřazovací příkazy, volání podprogramů, případně podmínky rozhodovacích příkazů). Každý pár uzlů, pro který je možný přenos řízení, je spojen hranou.

Analyzátory programu

Při překladu je ke každému příkazu připojen kód, který počítá, kolikrát byl daný příkaz vykonán (tzv. instrumentace). Po běhu můžeme zjistit, které části programu nebyly pokryty příslušným testovacím případem. Příklad nástroje: GCT (Generic Coverage Tool) volně šířený nástroj pro jazyk C.

Strategie testování

Testování by mělo být předem naplánováno spolu s celým SW procesem. Pro zvýšení efektivity testování je vhodné provádět také inspekce kódu. Testování by mělo začínat na úrovni jednotek (procedur, tříd; funkce každé jednotky se ověří samostatně) a postupovat směrem k větším celkům (podsystémům, celému systému).

Testování jednotek provádí obvykle ten, kdo danou část napsal; testování větších celků provádí nebo alespoň řídí specialista – tester (Rozsáhlejší software testuje nezávislá testovací skupina.)

Průběh testování v návaznosti na vývoj

Testování mělo probíhat postupně současně s implementací systému v následujících krocích:

- Testování jednotek (unit testing) – testujeme nejmenší jednotky návrhu, např. procedury nebo funkce; pro testování můžeme používat white-box techniky.
- Integrační testování (integration testing) – tím testujeme defekty týkající se rozhraní.
- Validační testování (validation testing) – funkce viditelné uživatelem.

- Testování systému (system testing) – pokud je SW pouze jednou součástí většího celku, účelem je otestovat celek; např. zátěžové testování, zotavení po závadě apod.

4. Oblasti testování

Testování jednotek

Pojmem „jednotka“ se v případě konvenčně napsaného SW obvykle myslí procedura, funkce, nebo nejmenší samostatně přeložitelná jednotka zdrojového textu (čili neexistuje všeobecně přijímaná definice). Jednotka se testuje samostatně, okolní jednotky jsou nahrazeny ovladači testů (řídí testovanou jednotku) nebo testovacími maketami (nahrazují jednotky volané z testované jednotky). Používají se již probrané white-box techniky.

Pro objektově-orientovaný software se za jednotku považuje třída – třídy jako samostatné komponenty jsou obvykle rozsáhlejší než samostatné podprogramy.

Integrační testování

Po otestování individuálních komponent musíme komponenty integrovat – sestavit – do částečného nebo úplného systému. Výsledek musíme otestovat na problémy, které vznikají interakcí komponent.

Jeden možný přístup je tzv.“big bang“ (velký třesk). Po otestování jednotlivých modulů je z nich v jediném kroku sestavena aplikace použitelná pouze pro malé programy. Pro větší systémy se jedná o nejméně efektivní způsob integrace.

Testování rozhraní

Cílem testování rozhraní je detekovat defekty, které mohou vzniknout chybnou interakcí mezi moduly nebo podsystémy nebo chybným předpokladem o rozhraní. Testování rozhraní je obtížné, protože některé typy defektů se projeví pouze za neobvyklých podmínek.

Validační testování

Začíná tam, kde končí integrační testování. Testujeme, zda SW splňuje požadavky uživatele. Akceptační testování – zadavatel určí, zda produkt splňuje zadání. Testuje se na reálných datech.

Pro generické produkty není většinou možné vykonat akceptační testování u každého zákazníka, proto probíhá alfa a beta testování.

Vyhodnocování samotného testování

Pro ohodnocení úspěšnosti naplňování těchto kritérií je vhodné mít kritéria nejen se dvěma hodnotami „splněno/nesplněno“, ale používat více hodnot, ideálně spojitou škálu ohodnocení. Pro tyto účely se z metodik managementu přejímají Vyvážené ukazatele výkonnosti (BSC, balanced score card). Cílem je nejen ohodnotit, že projekt naplňuje zadání na 150 % funkčnosti a 38 % podporovatelnosti, ale také na tato hodnocení navázat sledování nákladů, např. člověkohodin, počty odhalených chyb, statistiky vývojových úkolů a další.

Závěr

Nesystémový přístup k řízení vývoje software může mít fatální dopady na celý projekt – tedy na úspěšnost nasazení výsledného softwarového díla.

Opomenutí nebo přehlížení základních zásad kontroly kvality produkovaných aplikací, tedy nedostatečné testování software nebo jeho neúplné testování sice umožní rychlejší nasazení software, ale za cenu následných nákladů spojených s odstraňováním vad za provozu. Systémový a systematický přístup, na jehož základě firma navrhne a zavede dobrý systém řízení jakosti, přináší firmě značný přínos a nezanedbatelnou konkurenční výhodu, což platí i pro softwarové firmy. Tlak na kvalitu software stále poroste díky sílící konkurenci na poli vývojářských firem. Analýza současného stavu a odstranění dosavadních bariér pro dosahování vyšší kvality jsou nutným předpokladem k vytvoření dostatečně účinných procesů zajišťujících vysokou kvalitu dodávaného software. Správnou cestou, jak tohoto cíle dosáhnout, je optimální plánování fází testování v průběhu vývoje software. Každý z výše uvedených způsobů vývoje software má svá specifika, která je při testování nutné zohlednit. Nicméně všechny moderní, dnes používané metody vývoje, mají pevně zakotveny fáze testování a sledování kvality produkovaného kódu. Zde platí: čím dříve je chyba odhalena, tím jsou náklady na její odstranění nižší.

Kontrola kvality produkovaného software musí být jedním ze základních bodů strategie úspěšné softwarové společnosti. Pokud tomu tak není, nemá společnost jeden ze základních předpokladů pro udržitelný rozvoj. Chyby v dříve vyprodukovaném software, respektive jejich odstraňování, začne neúnosně zatěžovat zdroje.

Literatura

[1] Mannová, B. Ing., Ph.D. Katedra počítačů na Elektrotechnické fakultě ČVUT v Praze, Řízení SW projektů – Přednášky 2006

[2] Leština, P.: Vývoj aplikací: Co se nevyplácí podcenit, Computerworld 10/2009

[3] Procházka, J.: Riziky řízený vývoj software, SYSTÉMOVÁ INTEGRACE 1/2009

[4] Weinberger, J.: Řízení projektových rizik. Business World, prosinec 2005, str. 28-31.

[5] Doucek, P., Lea Nedomová, L.: Řízení kvality v malých a středních firmách1, [online]. Dostupné na: http://si.vse.cz/archive/proceedings/2000/rizeni-kvality-v-malych-a-strednich-firmach.pdf

[6] Wikipedie, Otevřená encyklopedie: Testování softwaru - [online]. Dostupné na: http:// http://cs.wikipedia.org/wiki/Testov%C3%A1n%C3%AD_softwaru

[7] Řepa, V. Analýza a návrh informačních systému. Ekopress, Praha 1999, ISBN: 80-86119-13-0

[8] Řepa, V. Chlapek, D. Materiály ke strukturované analýze. Vysoká škola ekonomická, Praha 1997, ISBN 80-7079-260-4.

[9] Wikipedie, Otevřená encyklopedie: Metodologie vývoje softwaru - [online]. Dostupné na: http:// http://cs.wikipedia.org/wiki/Metodologie_v%C3%BDvoje_softwaru

[10] Harazim, K., Michna, R.: Trendy v testování a verifikace programů, [online]. Dostupné na: http://axpsu.fpf.slu.cz/~sos10um/trendy/1-verifikace.doc

Ing. Martin Vonka, České vysoké učení technické v Praze