KIV/ZSWI 2003/2004
Přednáška 2

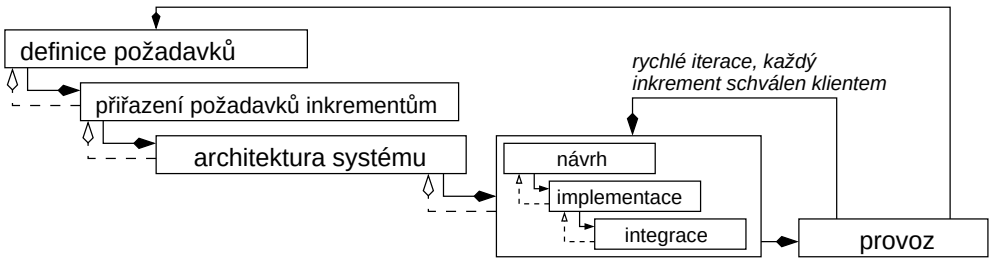
Iterativní modely vývoje SW

* všechny dříve uvedené modely SW procesu mají své výhody a nevýhody
* pro tvorbu většiny velkých systémů je potřeba používat různé přístupy pro různé části (např. podsystémy) => hybridní modely
* pokud se požadavky vyvíjejí => nutnost iterace částí procesu
* model by měl výše uvedené potřeby odrážet - proto uvedu 2 modely navržené speciálně pro iterativní procesy vývoje
  - inkrementální vývoj SW (Mills 1980)
  - spirálový model vývoje (Boehm 1988) a WinWin spirálový model (Boehm a Bose, 1994)

Inkrementální vývoj

* česky často nazýván ”přírůstkový model”
* byl navržen jako způsob, jak omezit přepracovávání částí v důsledku změn požadavků (inkrement = malý přídavek)
* někdy umožňuje zákazníkovi odložit rozhodnutí o detailních požadavcích dokud nezíská zkušenost se systémem

* proces vypadá v zásadě takto:
  - definují se přehledové požadavky
  - požadavky se přiřadí jednotlivým inkrementům
  - navrhne se architektura systému
  - opakovaně pro každý inkrement, dokud nevytvoříme finální systém: vyvineme a validujeme inkrement, integrujeme inkrement a validujeme systém.



* na začátku zákazník identifikuje přehledově služby, které od systému požaduje
* určí které služby jsou pro něj nejdůležitější a které jsou nejméně důležité

* pak je definována množina inkrementů, každý inkrement poskytuje podmnožinu celkové funkčnosti
* alokace služeb do inkrementů závisí na prioritě služby
* nejdůležitější služby mají být zákazníkovi předány jako první

* v dalším kroku jsou detailně specifikovány požadavky na první inkrement
* inkrement je vytvořen nejvhodnějším procesem; během vývoje nejsou akceptovány změny požadavků na vytvářený inkrement
* spolu s prvním inkrementem mohou být implementovány společné služby systému (někdy lze společné služby vytvářet také inkrementálně)
* paralelně s vývojem může probíhat analýza požadavků pro další inkrementy

* po dokončení inkrementu může být tento předán zákazníkovi, který ho může používat - získá tím část funkčnosti systému
* může se systémem experimentovat, což mu pomůže upřesnit požadavky na další inkrementy nebo na novější verze předaného inkrementu

* pro vývoj jednotlivých inkrementů se mohou používat různé procesy

- např. pokud má inkrement dobře definovanou specifikaci => vodopádový model
- pokud specifikace nejasná => evoluční model

Výhody inkrementálního přístupu:

* zákazníci nemusejí čekat na dokončení celého systému, aby z něj něco měli
* počáteční inkrementy jim mohou pomoci získat zkušenost pro definici dalších inkrementů
* je menší riziko celkového neúspěchu projektu; i když některé inkrementy mohou být problematické, je pravděpodobné že většina jich bude úspěšně předána zákazníkovi
* protože zákazníkovi předáváme jako první nejprioritnější inkrementy, budou nejdůležitější části systému nejlépe otestovány; jinými slovy - zákazníka pravděpodobně nepotká havárie nejdůležitější části systému

Potíže inkrementálního modelu:

* inkrementy by měly být relativně malé (do 20 000 řádek kódu), na druhou stranu by každý inkrement měl poskytovat nějakou navenek viditelnou funkčnost
  - proto může být obtížné namapovat požadavky zákazníka na inkrementy správné velikosti

* většina systémů obsahuje množinu základních služeb, které jsou vyžadovány různými částmi systému, požadavky však nejsou specifikovány detailně dokud se nedostaneme k jejich vývoji => je obtížné identifikovat služby potřebné pro všechny inkrementy
  - proto je pro větší projekty vhodné doplnit o fázi analýzy

* v současnosti nejznámější metodika - Extreme Programming (Beck 1999)
  - vývoj malých inkrementů, zatažení zákazníka do procesu, průběžné vylepšování kódu...

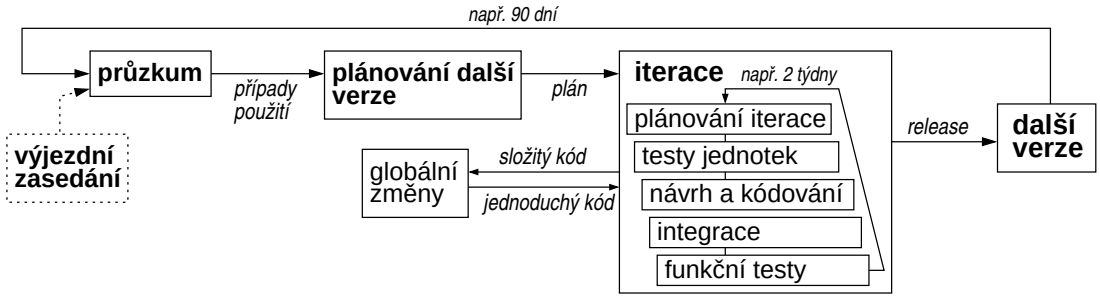
Poznámka (metodika Extreme Programming, XP)

* v současnosti nejznámější agilní metodika
  - autor jí doporučuje pro případy, kdy požadavky na SW jsou vágní nebo se rychle mění
  - použitelná pro malé týmy (2-10 lidí)

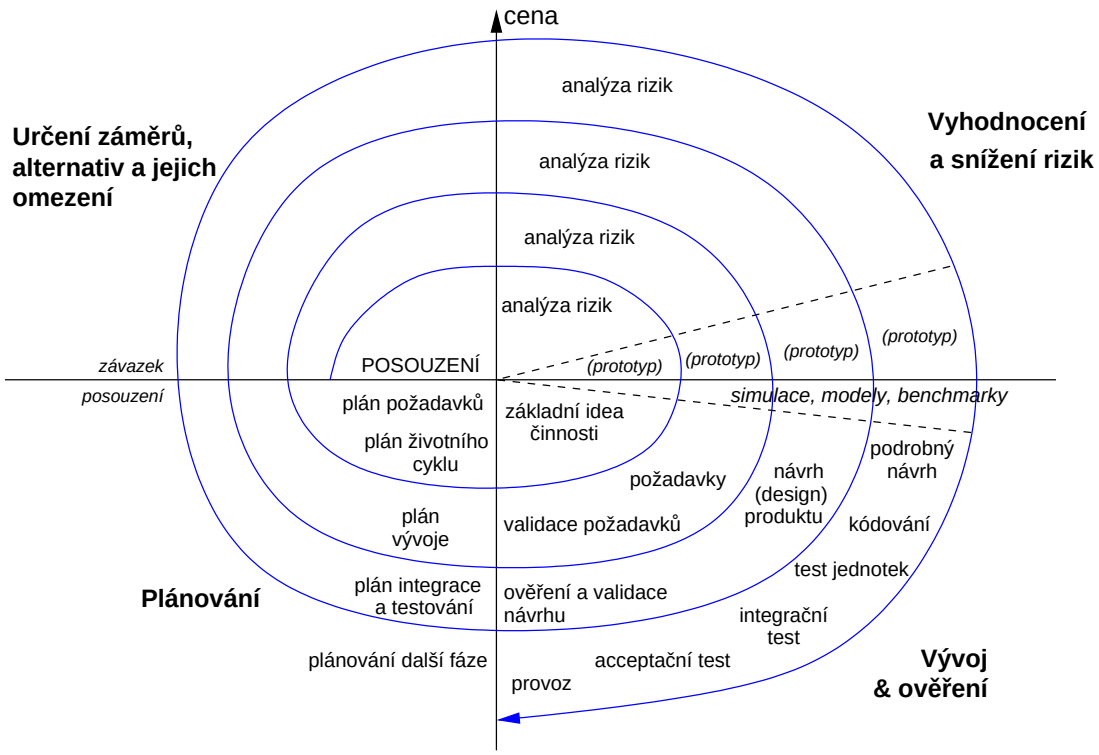
Životní cyklus v XP vypadá přibližně následovně:

* průzkum - v ideálním případě krátký
  - zákazník popíše případy použití vytvářeného systému (use cases, v terminologii Extreme Programming jsou nazývány ”stories”)
  - vývojáři na základě případů použití odhadnou, jak dlouho bude vývoj trvat (pokud to pro některý případ použití nelze odhadnout, je nutné ho přepsat, rozdělit, zkusit naprogramovat apod.)
  - vývojáři hledají alternativy pro architekturu systému (pomocí prototypů), identifikace rizik
* plánování další verze
  - zákazník si vybere nejmenší a pro něj nejcennější množinu případů použití, kterou bude chtít implementovat v první verzi
  - dohodnutí termínu pro dokončení první verze (mělo by být mezi 2-6 měsíci)
    . krátké cykly poskytují včas zpětnou vazbu zákazníkovi, může určit nebo změnit požadavky pro další cyklus
  - rozdělení času do 1-4 týdenních iterací, každá iterace má implementovat funkčnost pro množinu případů použití
    . kostra celého systému (architektura) musí být implementována v první iteraci => je nutné provést odpovídající výběr případů použití
* každá iterace:
  - rozdělení případů použití na úlohy, rozdělení úloh mezi programátory
  - výběr úlohy (většinou jeden programátor implementuje jeden případ použití)
  - vytvoření testů jednotek
    . v XP se napřed vytvářejí testy, pak teprve kód který je testován (např. ke každé třídě může existovat metoda ”test”, která jí otestuje)
    . testy by měly prověřit veškerou logiku budoucího kódu
    . při psaní testů se často vynořují alternativy návrhu
    . někdy se ukáže potřeba globální změny v systému - refactoring

```
    problematického kódu směrem k větší jednoduchosti
 - návrh a kódování
  . návrh i kód by měly být co nejjednodušší (pouze nyní požadované
    vlastnosti, žádné předpoklady do budoucnosti - protože požadavky
    zákazníka se budou měnit)
  . kód je považován za dokončený, pokud všechny testy proběhnou úspěšně
  . jednoduchost návrhu a testy zjednodušují budoucí přepracování kódu
 - integrace systému, testy jednotek
  . dokončená funkčnost je integrována do systému
  . spustí se automatické testy jednotek pro celý systém, tím zjistíme zda
    jsme omylem nezasáhli do funkčnosti systému
 - funkční testy
  . mají záběr do více podsystémů, někdy mají i manuální podobu (testování
    funkčnosti systému z pohledu zákazníka)
```




```
 * zahájení projektu může předcházet výjezdní zasedání (Off Site)
  - část zákazníků, managerů, programátorů a testerů se sejde (např. na
    několik dní) na jednom místě a dohaduje se o co jde (rozsah projektu...)

 * v XP je veškerý kód povinně psán ve dvojicích (párové programování)
  - jeden člen týmu kóduje, druhý může přemýšlet strategičtěji (např.
    doplnění chybějících testů, zda by se problém nevyřešil přepracováním
    části systému apod.)
  - má výhodu v kontrole (kolega si často všimne chyby kterou udělám) =>
    výsledkem kvalitnější kód
  - snadnější předávání zkušeností novým členům týmu apod.
  - nevýhoda: větší úroveň hluku (pokud jsou týmy v jedné místnosti)

* problémy XP:
 - zákazník i management musí být ochoten přistoupit na neobvyklé podmínky
 - po skončení projektu nezůstane dokumentace (pouze případy použití a
   dokumentace v kódu)

[]

Spirálový model SW procesu
..........................

* spirálový model SW procesu původně navrhl Boehm (1988)
* místo sekvence (kde se chtě nechtě někdy musíme vracet) je proces
  reprezentován spirálou

* každá otočka spirály reprezentuje fázi SW procesu, např.:
 - nejvnitřnější - všeobecný záměr, studie proveditelnosti
 - druhá - definice požadavků
 - třetí - návrh (design) systému
 - čtvrtá - vytvoření všech programů
```




```
* každá otočka spirály je rozdělena do 4 sektorů (sektory nemusejí trvat
  stejnou dobu):

* určení záměrů pro současnou fázi projektu (= otočku spirály)
 - definujeme záměry dané fáze projektu (funkčnost, výkonnost apod.)
 - určíme alternativní možnosti realizace záměrů (např. design A,
   design B, použití existující komponenty, nákup komponent)
 - identifikujeme omezení daných alternativ (cena, časový plán, přenositelnost)
 - vyhodnotíme alternativy vzhledem k záměrům a omezením - často se
   ukáže nejistota, která je zdrojem rizik
 - riziko = něco co se může nepovést
  . např. pokud se rozhodneme pro nový programovací jazyk, je riziko že
    dostupné překladače jsou nespolehlivé nebo negenerují efektivní kód
    (příklad - Corel a Java)
  . rizika mohou vyústit do problémů typu překročení deadline nebo
    překročení rozpočtu projektu
 - pokud vyplynula rizika, měli bychom se jimi v další fázi zabývat

* vyhodnocení a snížení rizik
 - pro každé z identifikovaných rizik je provedena detailní analýza a
   kroky ke snížení rizika
 - může zahrnovat prototypování, simulace, vypracování dotazníků apod.
 - např. pokud je riziko, že jsou nesprávně definovány požadavky, může být
   vytvořen prototyp části systému

* vývoj a validace
 - provádíme vývoj a ověření produktu další úrovně
 - použitý model SW procesu je určen zbývajícími riziky
  . např. pokud je hlavním rizikem integrace podsystémů, může být
    nejvhodnější vodopádový model (viz příklad na výše uvedeném obrázku)
  . pokud je hlavním rizikem nevhodné uživatelské rozhraní, může být
    nejpřiměřenějším modelem evoluční prototypování
  . pokud je hlavním problémem bezpečnost, může být zvolen vývoj
    založený na formálních transformacích

* plánování
 - naplánujeme další fázi projektu (organizace, požadavky na zdroje,
   rozdělení práce, časový plán, výstupy)
 - každá otočka je zakončena posouzením všech produktů vytvořených v
   předchozím cyklu, plánů pro další cyklus a potřebných zdrojů
  . cílem je aby byly všechny zúčastněné strany srozuměny s plány na
```

další fázi
. výsledkem např.: 15 lidí na 12 měsíců + základní plán životního cyklu pro alternativu, která se vynoří
. může určit i rozdělení projektu do inkrementů nebo do nezávisle vyvíjených komponent

* výše uvedený obrázek je pouze příklad, jakým způsobem lze spirálový model implementovat
- spirálový model je chápán jako ”generátor modelů SW procesu”, s jeho pomocí vytváříme podrobnější model SW procesu (například model uvedený na obrázku, ale konkrétní obsah jednotlivých otoček může vypadat i jinak)

Rozdíly oproti předchozím modelům:

* na rozdíl od předchozích modelů explicitně uvažuje rizika projektu
- z toho plyne i hlavní problém modelu - je závislý na zkušenosti vývojářů identifikovat zdroje rizik
* v modelu nenajdeme pevné fáze jako je specifikace nebo design - model je obecnější, může zahrnovat ostatní modely SW procesu:
- v jedné otočce může být použito prototypování abychom vyřešili nejistotu v požadavcích a tím redukovali rizika
- v další otočce může být následováno klasickým vodopádovým vývojem atd.
* model je aplikovatelný i na jiné typy projektů

* model široce známý, ale méně používán než klasický vodopádový model (zákazníci jsou zvyklí na vodopádový model, je to předpoklad mnoha zadání)

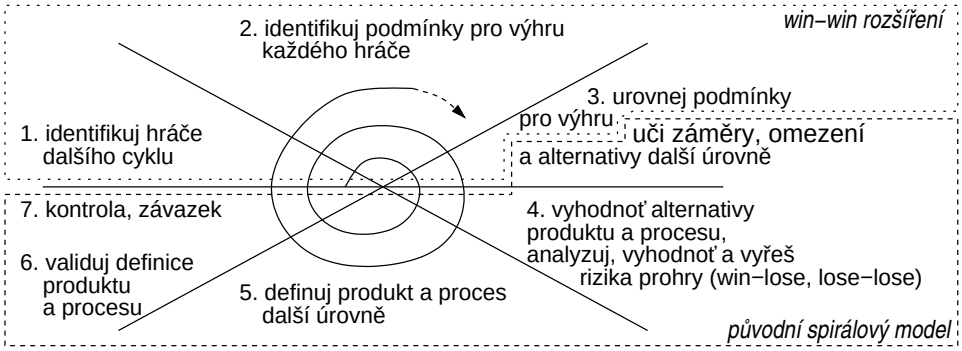
Původní článek o spirálovém modelu doporučuje následující Risk Management Plan, který lze používat i samostatně mimo spirálový model:

1. najděte 10 největších rizik projektu,
2. pro řešení každého rizika navrhněte plán,
3. seznam rizik, plány a výsledky aktualizujte každý měsíc,
4. v měsíční zprávě o průběhu projektu (plán vs. pokrok) uveďte stav rizik,
5. zahajte včas nápravné akce.

WinWin spirálový model

* původní spirálový model neposkytuje podrobnosti pro první kvadrant
- hlavně neříká, odkud se vezmou ”záměry pro současnou fázi projektu”
- proto byl původní model modifikován (Boehm a Bose, 1994)
- nový model je postaven na tzv. win-win přístupu: snaha aby všechny zúčastněné strany (zákazníci, vývojáři, vedoucí atd.) byli ”výherci”
. tj. aby tvorba produktu nedegradovala na hru s nulovým součtem (výhra-prohra, kde např. dodavatel vyhraje na úkor zákazníka nebo naopak)
. v nejhorším případě výsledek prohra-prohra, např. dodavatel prodělá a zákazník nezíská produkt, který potřebuje

* v novém modelu je zahrnuta snaha vyrovnat se s konfliktními zájmy
- např. zákazník: mít produkt co nejdříve, zaplatit za něj co nejméně
- vývojáři: odborný růst, pracovní a platové podmínky, preference pro používané nástroje, odložit psaní dokumentace apod.
- nadřízení: zisk, nepřekročení rozpočtu, žádná překvapení



* při pozornosti vůči zájmům a očekáváním jednotlivých hráčů můžeme často
  (ale ne vždy!) vytvářet situace typu výhra-výhra:

1. identifikace klíčových hráčů - do úvah musíme zahrnout všechny hráče
   podstatné pro daný cyklus projektu (např. nadřízené uživatelů apod.)
2. pochopení zájmů každého hráče (motivační analýza, obvykle nejobtížnější
   část - motivace jiných lidí budou totiž odlišné od našich)
3. urovnání podmínek pro výhru, řešení konfliktů
 - např. zákazníci většinou neodhadnou, co je snadné nebo obtížné
   naprogramovat => mají nerealistická očekávání; v tomto případě:
  . vhodné použít dobře kalibrovaný model pro odhad ceny a trvání vývoje
  . nabídnout novou alternativu typu výhra-výhra, např. ”všechny požadované
    funkce sice není možné vyvinout za 12 měsíců, ale můžeme pracovat podle
    přírůstkového modelu a za 12 měsíců splnit vaše nejdůležitější požadavky”

Ostatní části jsou stejné jako u původního spirálového modelu.

* uvedli jsme základní modely SW procesu (vodopádový až spirálové)
 - vodopádový model - základní model, konzistentní se strukturovaným
   programováním shora dolů; vhodný pokud jsou známé požadavky (např.:
   operační systémy, překladače apod.)
 - evoluční vývoj - pokud části požadavků nejsou zřejmé, např.
   uživatelské rozhraní (”nevím co chci, ale poznám to až to uvidím”)
 - formální metody - pokud musím dokazatelně splnit specifikaci, mám
   podpůrné nástroje a tým seznámený s formálními metodami
 - komponentově orientovaný vývoj - máme-li vhodné komponenty
 - inkrementální vývoj - potřebujeme omezit přepracovávání, dodáváme
   systém po částech
 - spirálový model - vhodné pro složité inovativní projekty vytvářené uvnitř
   organizace, zahrnuje předchozí modely jako speciální případy (nemusí být SW)
 - win-win spirálový - jako spirálový, ale vhodné pro projekty na zakázku.

* hlavním účelem modelů je určit správné pořadí kroků při vývoji SW a
  určit kritéria pro přechod k dalšímu kroku
* tj. model SW procesu odpovídá na otázky:
 - co budeme dělat příště?
 - jak dlouho to máme dělat?

[Poznámka WISCA.]

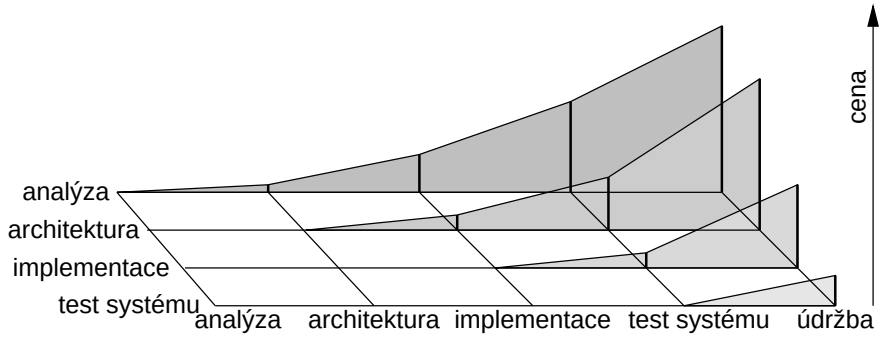
* v dalším povídání se seznámíme s některými metodologiemi
 - metodologie = jak provést příslušnou fázi
 - už na začátku bylo řečeno, že základní aktivity SW procesu jsou
   1. analýza domény a definice požadavků, 2. design systému a design SW, 3.
   implementace a testování modulů, 4. integrace a testování systému
 - jako první metodologii pro (1)

Analýza domény a definice požadavků

* v této fázi definujeme:
 - jaké služby jsou od systému požadovány
 - jaká jsou omezení vývoje a výsledného produktu
* chyby v této fázi SW procesu nevyhnutelně vedou k problémům při
  designu a implementaci!

Poznámka (cena změn):

 Studie potvrzují intuitivně zřejmou věc, že změny v počátečních
 stadiích projektu jsou podstatně levnější (10x až 200x) než později.
 Tj. čím později defekt detekujeme, tím více času i peněz nás bude stát
 jeho oprava (viz obrázek).



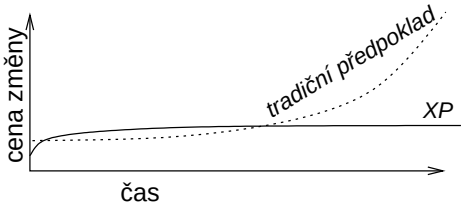
Např. defekt analýzy požadavků jehož oprava by nás stála 1000 Kč během fáze analýzy požadavků nás může stát 25 000 Kč během závěrečného testování systému (více u velkých systémů, méně u malých systémů).

Proto je zdravá snaha vyřešit problémy tak brzy, jak to jen jde.

[]

Poznámka pro zajímavost (cena změn a metodika Extreme Programming)

Autor metodiky XP tvrdí, že pečlivým dodržováním pravidel XP je možné cenu změn snížit takto:



Naneštěstí je to pouze jeho odhad - zatím nevím o žádných měřeních, které by to potvrzovaly.

[]

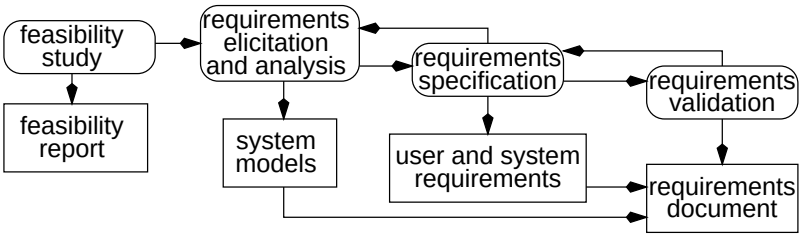
Při specifikaci požadavků na rozsáhlé systémy obvykle rozlišujeme 4 fáze:

* studie proveditelnosti (feasibility study)
 - odhad zda požadavky zákazníka mohou být splněny za pomocí existujících HW a SW technologií v rámci daných rozpočtových omezení
 - studie proveditelnosti má být rychlá a levná, výsledkem zda přikročíme k podrobnější analýze

* zjištění a analýza požadavků
 - zjišťujeme požadavky na nový systém pozorováním existujících systémů, diskusí s potenciálními uživateli a se zástupci zadavatele
 - může zahrnovat vývoj modelů a prototypů

* specifikace požadavků
 - výsledkem je dokument specifikace požadavků (DSP)
 - požadavky obvykle uvedeny ve dvou úrovních podrobnosti:
  . zákazník potřebuje vysokoúrovňový popis svých požadavků (”user requirements”)
  . vývojáři potřebují podrobnější specifikaci systému (”system requirements”)

* validace požadavků
 - kontrola požadavků - zda jsou realistické, konzistentní a úplné
 - během této fáze nacházíme chyby v DSP => DSP musíme opravit
 - v průběhu této i předchozích fází se mohou objevit nové požadavky => dtto



Studie proveditelnosti

* pro všechny nové systémy by SW proces měl začínat studií proveditelnosti
* snaží se zodpovědět následující otázky:
  - přispěje systém k celkovým záměrům organizace?
  - lze systém implementovat za pomocí existujících HW a SW technologií?
  - bude možné systém integrovat s již existujícími systémy?
  - bude vývoj systému finančně efektivní (tj. neproděláme na tom?), resp. lze systém realizovat v rámci daných rozpočtových omezení?

* vstupem je obecný popis + jakým způsobem má být v organizaci využíván
* výstupem je zpráva která doporučuje nebo nedoporučuje pokračovat ve vývoji
* obecné fáze - identifikace potřebných informací, získání informací, vytvoření zprávy

Musíme najít informační zdroje - manažery, koncové uživatele, lidi kteří znají podobné systémy apod., budeme se jich ptát:

* jaké jsou problémy se současným procesem zpracování informací, jak je má nový systém pomoci řešit?
* lze přenášet informace z a do jiných informačních systémů, které organizace již provozuje?
* co musí být v systému podporováno a co nemusí?
* je zapotřebí technologie, která ještě nebyla použita?

Výsledná studie by:

* měla obsahovat doporučení zda pokračovat ve vývoji
* může navrhnout změny rozsahu, rozpočtu a časového plánu systému
* může navrhnout další vysokoúrovňové požadavky na systém

Analýza domény a zjištění požadavků

* za pomocí zadavatele se seznamujeme s aplikační doménou a zjišťujeme, jaké služby má systém poskytovat
  - aplikační doména = obor, ze kterého problém pochází

* je to obtížné, protože:
  - zadavatel obvykle ve skutečnosti neví přesně, co od systému požaduje, resp. je pro něj obtížné to vyjádřit; může také mít nerealistické požadavky, protože nezná cenu jejich realizace
  - zadavatel vyjadřuje požadavky ve vlastních termínech a s implicitní znalostí vlastní práce (aplikační domény); vy musíte pochopit požadavky i bez této zkušenosti
  - různí zástupci zadavatele mají různé požadavky, které vyjadřují různým způsobem; vy musíte určit všechny zdroje požadavků, najít společné části a části které jsou v konfliktu
  - požadavky mohou ovlivňovat politické faktory; například konkrétní manažeři mohou mít specifické požadavky, které by pomohly posílit jejich vliv v organizaci
  - prostředí ve kterém se provádí analýza se průběžně mění - důležitost jednotlivých požadavků se může změnit; nové požadavky mohou přicházet od zástupců zadavatele, kteří do toho předtím neměli co mluvit.

Obecný model analýzy požadavků vypadá takto:

* porozumění aplikační doméně - analytik musí pochopit aplikační doménu
  - např. pokud řeší informační systém supermarketu, musí vědět jak

fungují supermarkety
* sběr požadavků - analytik zjišťuje požadavky na systém od zástupců zadavatele
* klasifikace požadavků - nestrukturovanou množinu požadavků se snažíme logicky uspořádat
* řešení konfliktů - pokud je více zadavatelů, některé požadavky budou vzájemně v rozporu; rozpory požadavků musíme vyhledat a vyřešit
* určení priorit - konzultací se zadavatelem bychom měli určit nejdůležitější požadavky
* kontrola požadavků - musíme zjistit, zda jsou požadavky úplné, konzistentní a zda odpovídají tomu, co zadavatel od systému chce

Ve skutečnosti je to opět iterativní proces, kde se musíme vracet.

Specifikace požadavků

Nejdříve si uvedeme základní klasifikaci typů požadavků (funkční, mimofunkční a doménové); poté se budeme zabývat uživatelskými a systémovými požadavky a způsobem jejich specifikace. Nakonec uvedeme jak by měl vypadat dokument specifikace požadavků.

Typy požadavků

Požadavky na systém lze rozdělit do následujících tříd:

* funkční požadavky
* mimofunkční požadavky
* doménové požadavky

* funkční požadavky (functional requirements)
 - popisují funkce nebo služby, které jsou od systému očekávány
 - například požadavky na univerzitní knihovní systém:
  . uživatelé by měli mít možnost prohledávat počáteční množinu databází nebo její podmnožinu
  . systém by měl poskytovat uživatelům vhodné prohlížeče pro čtení dokumentů v úložišti dokumentů

* mimofunkční požadavky (non-functional requirements)
 - netýkají se funkcí systému, ale vlastností jako je spolehlivost, čas odpovědi a obsazené místo na disku nebo v paměti
 - často kritičtější než jednotlivé funkční požadavky, např. pokud je řídící systém letadla nespolehlivý, je nepoužitelný
 - někdy se mimofunkční požadavky týkají SW procesu, např. v procesu musí být použit určitý standard pro řízení jakosti (ISO 9000), design musí být vytvořen určitým CASE nástrojem, pro implementaci musí být použit daný programovací jazyk, produkt musí být dodán do určitého data apod.
 - někdy jsou požadavky dané vnějšími faktory, např. legislativní požadavky (systém musí být navržen v souladu se zákonem na ochranu osobních informací apod.)

 - například následující části specifikace definují mimofunkční požadavky:
  . 3.6.6 Veškerá komunikace mezi uživatelem a systémem by měla být vyjádřitelná ve znakové sadě ISO 8859-2.
    (Definuje omezení návrhu uživatelského rozhraní, tj. není funkce ale omezení => mimofunkční požadavek)

  . 3.6.8 Proces vývoje systému a všechny dokumenty mají odpovídat softwarovému procesu definovanému ve standardu XYZ.
    (Mimofunkční požadavek týkající se SW procesu.)

  . 3.6.9 Systém nemá operátorům systému poskytovat žádné osobní informace o zákaznících kromě jména a čísla zákazníka.
    (Požadavek daný vnějšími faktory, např. přijatelností systému pro veřejnost.)

```
* časté problémy mimofunkčních požadavků
 - obecně definované mimofunkční požadavky jdou obtížně ověřitelné,
   takže mohou působit neshody po dodání systému zákazníkovi
  . příklad obecné definice: Systém má být snadno použitelný a
    nebezpečí chyb uživatelů by mělo být minimalizováno.
  . příklad ověřitelné definice: Zkušené účetní by měly být schopny
    používat všechny funkce systému po třídenním zaškolení. Počet chyb
    po zaškolení by neměl přesáhnout dvě za den.
  . mnoho požadavků je obtížné definovat měřitelným způsobem, někdy to
    ani nejde (např. udržovatelnost systému)
 - mimofunkční požadavky mohou být v konfliktu, např.:
  . systém pro řízení sondy má být vytvořen v jazyce Ada
  . maximální velikost systému má být 4 MB (bude instalován v ROM)
  . je zapotřebí vyřešit - určit jiný jazyk nebo zvětšit paměť
```

Poznámka (příklad mimofunkčních požadavků a jejich metrik)

rychlost	průchodnost, např. počet transakcí za sekundu čas odpovědi na požadavek uživatele nebo na událost (průměrný, maximální) čas překreslení obrazovky
velikost	velikost v paměti, na disku, např. v KB
snadnost použití	doba potřebná pro zaškolení doba trvání typické úlohy rozsah nápovědy soulad se standardem pro GUI
spolehlivost	počet havárií za časovou jednotku průměrná doba mezi haváriemi pravděpodobnost nedostupnosti počet chyb/1000 řádek kódu
robustnost	doba zotavení po havárii doba opravy po havárii pravděpodobnost poškození dat při havárii přijatelné chování při degradaci systému
přenositelnost	velikost kódu závislého na platformě (např. v %) počet cílových platforem
kapacita	např. počet zákazníků který systém zvládne

[]

Poznámka (pojem ”robustní”)

Pojem ”robustní” = schopný rozumně zvládnout chybové stavy (fčnosti, mezní zátěže apod.

[]

```
* doménové požadavky (domain requirements)
 - vyplývají z aplikační domény, nikoli ze specifických požadavků zadavatele
 - mohou být funkční nebo mimofunkční
 - např. část specifikace ze systému pro automatické zastavení vlaku
   přejede-li červenou:
```

Zpomalení vlaku bude vypočteno jako $D = D_c + D_\Delta$, kde $D_\Delta = 9.81\,\mathrm{ms}^2 \cdot \frac{\mathrm{gradient}}{\alpha}$; hodnota α je známá pro různé typy vlaku.

```
 - problémy s doménovými požadavky:
  . specialisté rozumí doméně natolik dobře, že je nenapadne doménové
    požadavky specifikovat explicitně
  . vývojáři o nich nemusejí vědět nebo jim rozumět
```

```
Další rozdělení vyplývá z rozdílné úrovně popisu - pro různé čtenáře:

* uživatelská specifikace požadavků (user requirements specification)
 - vysokoúrovňový popis funkčních a mimofunkčních požadavků zákazníka
 - musí být srozumitelné pro uživatele, kteří nemají technické znalosti
   (manažery klienta a dodavatele, koncové uživatele)
* systémová specifikace požadavků (system requirements specification)
 - podrobnější specifikace uživatelských požadavků pro vývojáře dodavatele
 - má být přesná
  . může sloužit jako základ kontraktu mezi zákazníkem a dodavatelem
  . slouží jako výchozí bod pro design systému

Příklad (uživatelská specifikace požadavků):

 1. Systém musí poskytnout způsob reprezentace externích dokumentů a
    možnost jejich prohlížení.

Příklad (systémová specifikace požadavků):

 1.1 Uživateli bude poskytnuta možnost definovat typy externích dokumentů.
 1.2 Každý typ externího dokumentu bude na obrazovce reprezentován určitou
     ikonou.
 1.3 Uživateli bude poskytnuta možnost definovat pro typ externího
     dokumentu vlastní ikonu.
 1.4 Uživateli bude poskytnuta možnost sdružit typ externího dokumentu
     s prohlížečem.
 1.5 Pokud uživatel vybere ikonu reprezentující externí dokument, výsledkem
     bude spuštění prohlížeče sdruženého s typem externího dokumentu pro
     dokument reprezentovaný vybranou ikonou.
 Zdůvodnění: Je pravděpodobné, že nové dokumenty budou distribuovány ve
     formátech, které zatím nejsou známé.
[]


Doporučení:

* zavést si standardní formát a dodržovat ho; např. zvýraznit počáteční
  požadavek (1.), k požadavku připojit zdůvodnění
 - zdůvodnění je důležité ve chvíli, kdy je navržena změna požadavku -
   známe důvod původní verze
* používat jazyk konzistentním způsobem, hlavně rozlišit nutné
  požadavky od preferencí (např. pojem ”bude” bude vždy znamenat nutný
  požadavek, zatímco pojem ”měl by” bude vždy znamenat žádoucí chování)
* nepoužívat žargon

Příklad:
```

1. Systém musí poskytnout způsob reprezentace externích dokumentů a možnost jejich prohlížení.

1.1 Uživateli bude poskytnuta možnost definovat typy externích dokumentů.

1.2 Každý typ externího dokumentu bude na obrazovce reprezentován určitou ikonou.

1.3 . . .

Zdůvodnění: Je pravděpodobné, že nové dokumenty budou distribuovány ve formátech, které zatím nejsou známé.

❇