Dokument specifikace požadavků

Ve skutečnosti se můžeme setkat se 2 druhy dokumentů specifikujících požadavky:

1. dokument specifikující požadavky na systém (angl. Concept of Operations, ConOps document; používají se ještě další názvy)
 - vysokoúrovňový popis požadavků z hlediska zadavatele (musí se vyjadřovat v termínech zadavatele, protože ho budou číst také zástupci zadavatele)
 - kromě seznamu požadavků obsahuje informace o celkových záměrech systému, cílovém prostředí, omezeních, předpokladech a mimofunkčních požadavcích
 - může obsahovat modely kontextu systému, případy použití, toky informací a práce apod.
 - slouží pro validaci systémových požadavků

2. dokument specifikující požadavky na software (angl. Software Requirements Specification, SRS)
 - v češtině termín Dokument specifikace požadavků (DSP)
 - podrobná specifikace požadavků na software, odvozená z požadavků na systém
 - předpokládá se, že čtenáři už mají ponětí o SW inženýrství => jazyk může být přesnější, notace podrobnější
 - měl by obsahovat specifikaci uživatelských i systémových požadavků
  . uživatelské i systémové požadavky mohou být sdruženy v jednom popisu
  . často lepší uvést uživatelské požadavky v úvodu k systémovým požadavkům
  . pokud by byl rozsah dokumentu neúměrný, je možné vytvořit systémové požadavky jako samostatné dokumenty
 - obsahuje oficiální vyjádření o tom, co se od vyvíjeného systému očekává => pro produkty vyvíjené na zakázku může sloužit jako základ kontraktu
  . uvnitř organizace může hrát roli zákazníka např. obchodní oddělení, výzkumné oddělení apod.
  . pro generické produkty může být výhodné najmout si kvalifikovaného uživatele

Různé velké organizace definovaly vlastní standardy a doporučení pro strukturu obou typů dokumentů, např. IEEE std 1362-1998 (struktura ConOps) a IEEE std 830-1998 (struktura DSP).

Např. struktura podle IEEE/ANSI 830 (”IEEE Guide to Software Requirements Specifications”) vypadá zjednodušeně takto:

```
  Table of Contents                                  // obsah
  1. Introduction                                    // úvod
   1.1 Purpose of the requirements document          // účel DSP
   1.2 Scope of the product                          // rozsah produktu
   1.3 Definitions, acronyms and abbreviations       // definice, zkratky
   1.4 References                                    // odkazy
   1.5 Overview of the remainder of the document     // přehled zbytku DSP
  2. General description                             // obecný popis
   2.1 Product perspective (independent/part of)     // kontext produktu
   2.2 Product functions                             // funkce produktu
   2.3 User characteristics                          // charakteristiky uživatelů
   2.4 General constraints                           // obecná omezení
   2.5 Assumptions and dependencies                  // předpoklady a závislosti
  3. Specific requirements (functional, non-functional and interface
     requirements)                                   // specifické požadavky
     ....                                            // (není std. struktura)
  4. Appendices                                      // přílohy
  5. Index                                           // rejstřík
```

Ve skutečnosti bude informace v DSP záviset na vyvíjeném produktu a na modelu SW procesu, takže je nutné si obecný model přizpůsobit.

Poznámka:

Ne všechny instituce dokumentují a udržují požadavky na vyvíjený SW. Zejména v malých firmách se silnou vizí je správa požadavků často považována za zbytečnou režii. Pokud ovšem dostatečně naroste báze uživatelů a produkt se

vyvíjí, pak nutnost vyhodnocovat navrhované změny vede k potřebě zjistit původní požadavky, které které určovaly vlastnosti produktu.

[]

Doporučení: pro praxi si navrhněte si vlastní formát a používejte ho pro všechny DSP - sníží se tím pravděpodobnost, že na něco zapomenete.

Poznámka (zápočtová úloha ze ZSWI)

* na stránkách ZSWI je vystavena osnova DSP (soubor pro MS Word)
* osnova je odvozená ze standardu IEEE 830
* tento soubor můžete využít - před odevzdáním z něj zrušte nápovědu
* pro účely úlohy ze ZSWI si ho rozumným způsobem přizpůsobte
* body, které se netýkají vaší úlohy můžete odbýt poznámkou typu ”Není.”
* popis funkčních požadavků by měl vykazovat nějakou strukturu

[]

DSP by měl splňovat následující body (výběr z [Heiniger1980], [Pressmann] a [SWEBOOK2001]):

* DSP by měl specifikovat pouze externí chování systému
  - tj. snaha vyloučit z DSP návrh SW do té míry do jaké je to možné
  - někdy jsou ale požadavky a design neoddělitelné, např. systém může komunikovat s jiným systémem, z čehož vyplývají požadavky na design
* DSP by být strukturován tak, aby v něm bylo snadné provádět změny
  - popis by měl být lokalizovaný a volně vázaný
* DSP by měl specifikovat omezení implementace
* DSP by měl charakterizovat přijatelné odpovědi na nežádoucí události
* DSP by měl zaznamenat představu o životním cyklu systému

Způsoby specifikace požadavků

Přirozený jazyk

* požadavky jsou obvykle popsány přirozeném jazyce - výhodou srozumitelnost pro uživatele i pro vývojáře
* přirozený jazyk má v určitých případech své nevýhody
  - nejednoznačnost popisu
  - složité koncepce (např. algoritmy) je obtížné popsat přesně
  - příliš flexibilní - stejná věc se dá říci mnoha různými způsoby; jak čtenář zjistí, že se požadavky liší a čím?
  - neexistuje jednoduchý způsob modularizace - jak zjistíte důsledek změny požadavků, tj. kterých všech dalších požadavků se změna dotkne?

* použití přirozeného jazyka je nevyhnutelné (jediné čemu rozumí všichni), proto je třeba používat jazyk jednoduše, vědomě a snažit se vyhnout běžným příčinám chybné interpretace
* příklad problematické definice:

  Pokud uživatel zadá jméno delší, než je šířka formuláře, jeho pokračování se zobrazí na následující obrazovce. (Nejednoznačné: zobrazí se na následující obrazovce pokračování jména, nebo formuláře?)

* proto je třeba se snažit vyhnout:
  - dlouhým souvětím s mnoha vedlejšími větami
  - používání termínů s několika přijatelnými významy
  - prezentaci několika požadavků jako jediného požadavku

- nekonzistenci v používání termínů, např. používání synonym.

Poznámka (měření kvality DSP)

V některých případech (velmi rozsáhlé projekty) se používají metriky, měřící kvalitu DSP. Měřit lze velikost a čitelnost textu (délka vět apod.), strukturu textu (hloubku struktury a délka částí). Slabá místa specifikace lze najít hledáním výskytu neurčitých slov nebo frází (několik, především) apod.

[]

* proto v systémových specifikacích snaha přidat strukturu, která pomůže omezit nejednoznačnosti
* uvedeme zatím pouze 2 možnosti:
 - formuláře
 - pseudokódy

Formuláře
....

* přirozený jazyk je příliš flexibilní, proto se někdy přidává (vynucuje) struktura pomocí formulářů
 - pro vyjádření požadavků definujete jeden nebo více typů formulářů
 - formulář by měl obsahovat:
  . popis specifikované funkce nebo entity
  . popis vstupů a odkud se berou
  . popis výstupů a kam putují
  . jaké další entity specifikovaná funkce nebo entita používá
  . případné vstupní a výstupní podmínky (pre-conditions a post-conditions), tj. co platí při vstupu do funkce a co při výstupu z ní
  . pokud vznikají postranní efekty, pak jejich popis

Příklad systémové specifikace funkce pro vkládání prvku do diagramu, zapsaná ve formě formuláře:

```
  ---------------------------------------------------------------------
  Funkce: Vlož prvek do diagramu.

   Popis: Vloží prvek do existujícího diagramu. Uživatel určí typ prvku
          a jeho pozici.

  Vstupy: Typ prvku, Pozice prvku, Identifikátor diagramu.

  Zdroje: Typ prvku a Pozici prvku zadá uživatel, Identifikátor diagramu
          získáme z databáze diagramů.

 Výstupy: Identifikátor diagramu.

Úložiště: Databáze diagramů. Při dokončení operace je proveden COMMIT.

Vyžaduje: Diagram odpovídající vstupnímu Identifikátoru diagramu.

 Vstupní podmínka: Diagram je otevřen a zobrazen na obrazovce uživatele.

Výstupní podmínka: Diagram je nezměněn kromě přidání prvku určeného typu
          na určenou pozici.

 Vedlejší efekty: Nejsou.
 ---------------------------------------------------------------------
```

Případy použití
....

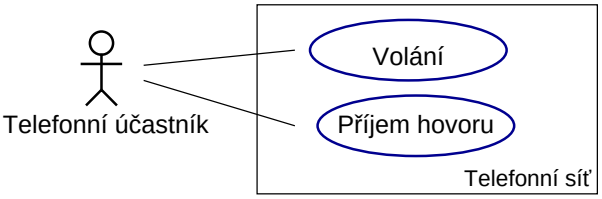
* use-cases [Jacobson at al. 1993]
* používají se zejména pro popis kontextu systému a pro popis funkčních požadavků
* základ některých metodik, např. sběr požadavků v RUP, XP apod.

* případy použití jsou součástí grafické notace UML (Unified Modeling Language) pro popis objektově orientovaných modelů systému
 - stejný typ diagramu je možné použít pro popis chování systému, podsystému nebo třídy
 - podpora v moderních CASE nástrojích

* případ použití reprezentuje vnější pohled na systém, modeluje zamýšlené fce systému a jeho vztah k okolí
* uživatelé a další systémy které mohou se SW systémem interagovat jsou nazývání aktéři (angl. actors), česky někdy aktoři nebo herci
 - v diagramu jsou aktéři reprezentováni jako panáčci, název aktéra má být uveden pod figurkou
 - aktér definuje koherentní množinu rolí, kterou uživatelé systému mohou hrát při interakci se SW systémem
 - aktérem nemusí být člověk, může to být i jiný systém
* třídy interakce nazýváme ”případy použití” (angl. use cases)
 - v diagramu jsou zakresleny jako pojmenovaná elipsa, název může být umístěn v elipse nebo pod ní
 - navenek se projeví posloupností zpráv vyměněných mezi systémem a jedním nebo více aktéry
* účast aktéra na případu použití se nazývá ”asociace” a značí se čarou spojující aktéra s případem použití

* aktéři reprezentují uživatele systému, jejich znalost nám pomůže určit hranici systému a co má systém dělat
* na základě potřeb aktérů vytvoříme případy použití (uvidíme dále jak)
* tím zajistíme, že vytvářený systém bude splňovat potřeby uživatelů

* použití pro modelování kontextu systému
 - máme-li jakýkoli systém, některé věci jsou uvnitř a některé vně
 - např. telefonní síť
  . uvnitř telefony, dráty, ústředny, účtování apod.
  . vně uživatelé sítě
 - kontext systému tvoří vše co je vně systému a se systémem interaguje
 - zakreslíme ohraničením celého systému čarou a určením aktérů kteří s ním interagují



* použití pro modelování požadavků na systém
 - diagram případů použití může být startovací bod pro uživatele i vývojáře
 - na začátku potřebujeme vědět co všechno má systém dělat, později k tomu přidáváme podrobnosti
 - notace umožňuje rozlišit případy použití na podpřípady používané ostatními případy použití a vytvářet varianty, zavádět vztahy zobecnění a specializace aktérů (později uvedu podrobněji).

Pseudokódy
....

* někdy je funkce specifikována jako posloupnost jednodušších akcí,
  pořadí je podstatné
* v přirozeném jazyce bychom obtížně vyjadřovali např. vnořené podmínky
  nebo smyčky
* pak může být vhodné doplnit specifikaci popisem v pseudokódu

* pseukód
 - jazyk s abstraktními konstrukcemi, které právě potřebujeme
   (sekvence jednoduchých příkazů, iterace, větvení):
 - vnoření konstrukcí je vyjádřeno odsazením
 - vyhýbáme se syntaktickým konstrukcím cílového programovacího jazyka
   (které by nás zbytečně omezovaly a sváděly k programování)
 - popisujeme požadovaný záměr, nikoli jak bude v cílovém jazyce
   implementován
 - na druhou stranu musí umožňovat téměř automatickou konverzi do kódu
   (pokud příliš vysoká úroveň, nemusíme postřehnout problémy ve
   specifikaci)

Příklad (část popisu činnosti bankomatu):
```
    ----------------------------------------------------------------
    Přečti kartu
    Vypiš výzvu: ”Prosím zadejte PIN”
    Přečti zadané_PIN
    Opakuj nejvýše 3x
        Přečti zadané_PIN
        Jestliže zadané_PIN je PIN_karty pak opusť smyčku
    Jestliže zadané_PIN není PIN_karty pak ...
    ----------------------------------------------------------------
```

Poznámka (další použití pseudokódů)

 Podrobnější příklady pseudokódů uvedu při popisu strukturované analýzy, kde
 se pseudokódy používají pro specifikaci tzv. procesů; s pseudokódy se
 potkáme také v tématu kódování.

[]

* formuláře vs. pseudokód
 - formuláře - celková specifikace systému
 - pseudokód - řídící sekvence, rozhraní

Specifikace rozhraní
....

* pokud musí nový systém komunikovat s dalšími systémy, musí být přesně
  specifikováno softwarové nebo komunikační rozhraní
* specifikace rozhraní měla by být součástí DSP, pokud je rozsáhlá tak
  v příloze; měla by být vytvořena brzy

* 2 typy rozhraní, které musejí být definovány:
 - procedurální rozhraní - existující podsystémy poskytují množinu
   služeb, které přístupné prostřednictvím volání procedur rozhraní
 - popis předávaných dat
  . popis struktury dat - pro popis se používají datové modely,
    nejznámější jsou ERA diagramy
  . popis reprezentace dat, např. ”datum je reprezentován jako řetězec ...”

* klasickým příkladem specifikace procedurálního rozhraní je popis knihovních
  procedur nebo tříd programovacího jazyka
 - v klasických jazycích např. prototyp procedury nebo fce, popis
   vstupních/výstupních parametrů, popis činnosti, návratová hodnota
 - objektových jazycích např. obecný popis třídy, popis konstruktorů,
   popis metod

Zjednodušený příklad - definice rozhraní s tiskovým serverem:

```
  ---------------------------------------------------------------------
  interface PrintServer {
  // definuje rozhraní s tiskovým serverem
        void initialize (Printer p);            // inicializace tiskárny
        void print (Printer p, PrintDoc d);     // tisk dokumentu
        PrintQueue queryPrintQueue (Printer p); // obsah tiskové fronty
        void cancelPrintJob (Printer p, PrintDoc d); // zrušení tisku dokumentu
  } //PrintServer
  ---------------------------------------------------------------------
```

* kromě přirozeného jazyka, formulářů, případů použití a pseudokódů se pro specifikaci požadavků ještě používají
 - grafické notace - např. ostatní diagramy UML (zatím jsme viděli případy použití)
 - formální (matematická) specifikace (hodně jednoduchý příklad: konečný automat; ten lze znázornit také v UML)

Proces specifikace požadavků

Na minulé přednášce jsme si uváděli obecný model fáze specifikace požadavků. Za nejdůležitější fáze můžeme považovat:

* sběr požadavků - od koho požadavky získat a jak
* klasifikace požadavků, detekce a řešení konfliktů (někdy se nazývá ”analýza požadavků”, ale tento název se také používá v trochu jiném významu, proto se mu vyhýbám)
* specifikace požadavků (ConOps a DSP)
* validace požadavků (validace = ověření, kontrola).

Ve skutečnosti to nemusí proces podle vodopádového modelu, lze použít i spirálový model. Jednotlivé aktivity se opakují, dokud nevznikne přijatelný DSP.

[Dokreslit obrázek: spirálový model podle SWEBOOK2001, p. 16]

Poznámka (následné fáze SW procesu)

V malých systémech můžeme z DSP pokračovat přímo tvorbou návrhu systému. Pokud je systém rozsáhlý, jsou někdy zapotřebí další (např. 2-3) cykly analýzy, které přidávají další podrobnosti a interpretují doménové požadavky pro vývojáře (aby byli schopni doménové požadavky správně interpretovat). Poté je vývoj předán návrhářům.

[]

Sběr požadavků

* tato fáze se zabývá zdroji požadavků a způsobem jejich získávání
* zdroje požadavků - je zapotřebí je identifikovat a vyhodnotit jejich vliv na systém (příklady zdrojů požadavků: celková motivace systému, doménové znalosti, zadavatelé systému, provozní prostředí, prostředí organizace)

* pokud byly identifikovány zdroje požadavků, analytik nebo analytici zjišťuji požadavky na systém od zástupců zadavatele
 - je zapotřebí počítat s tím, že uživatelé mohou mít potíže své požadavky vyjádřit, opomenout důležité informace, nebo nemusí chtít spolupracovat
 - je obtížné i když jsou zadavatelé dostupní a ochotní spolupracovat

* existuje mnoho technik, z nich nejdůležitější:
 - interview = předem připravený rozhovor
 - případy použití - definice případů použití se scénáři jejich průběhu
 - tvorba prototypů - od papírových modelů obrazovek po SW prototypy; pomůže uživateli lépe pochopit, jaká informace se od něj požaduje
 - pozorování prací u zákazníka případně účast analytiků dodavatele na

```
   pracích u zákazníka (relativně drahé)
 - analýza existujícího SW systému

Dále stručně popíšeme interview a podrobněji sběr požadavků na základě
případů použití.

Interview
.........

* nejčastěji forma získávání požadavků
 - předem připravený rozhovor, který vede moderátor (klade otázky, dává
   slovo)
* nedoporučuje se trvání delší než 2 hodiny na setkání
* doporučuje se předem si připravit scénář - které okruhy se budou probírat,
  v jakém pořadí, scénář se snažit nenásilně dodržovat

* např. cílené interview má strukturu:
 - moderátor shrne účel interview a jeho strukturu
 - posloupnost otázek k jednotlivým bodům
 - závěrečné shrnutí + ověření že informace byly správně pochopeny

* otázky by měly být otevřené
 - příklady otevřených otázek:
  . Co má systém řešit? Jak to řešíte nyní?
  . Kdo bude uživatelem systému?
  . Je něco dalšího na co bych se vás měl zeptat?
 - příklady uzavřených otázek:
  . Je 50 položek přiměřené množství?
  . Potřebujete větší formulář?
* čemu je vhodné se vyhnout:
 - není dobré chtít po uživatelích popis příliš složitých aktivit (lidé
   dělají spoustu věcí, které neumějí popsat, např. zavazují si tkaničky)
 - je dobré se vyhnout otázkám začínajícím ”Proč”, mohou vyprovokovat obranný
   postoj


Sběr požadavků na základě případů použití (use-cases)
.....................................................

* pro většinu lidí je jednodušší zacházet s případy z reálného života
  než s abstraktním popisem
 - např. dokáží pochopit a okomentovat scénář toho, jak budou interagovat se
   SW systémem
 - při získávání požadavků toho můžeme využít pro definici skutečných
   požadavků

* přehled vývoje modelu případů použití:
 - na základě požadavků zákazníka najdeme aktéry a případy použití,
   stručně je popíšeme
 - model by měl být přezkoumán zákazníkem
  . zda jsme našli všechny aktéry a případy použití
  . zda dohromady poskytuje co zákazník chce
  . v iterativním procesu vývoje pak můžeme určit priority případů použití a
    v každé iteraci vybrat množinu případů použití pro podrobnější
    specifikaci
 - specifikovat podrobně posloupnost událostí pro každý případ použití
 - model můžeme strukturovat
 - úplný model je přezkoumán, slouží jako základ dohody mezi vývojáři a
   zákazníkem o tom, co má systém dělat

Hledání aktérů a případů použití:

* dále uvedu konkrétní metodiku, pocházející z RUP (Rational Unified Process)
 - vhodná pro střední a velké systémy
 - často se používá, protože notace případů použití je součástí UML a má
   podporu v CASE nástrojích

* svoláme pracovní setkání pro hledání případů použití
* je to brainstorming, potřebujeme lidi různých znalostí a zkušeností
```

* je dobré aby skupina byla malá (< 10 lidí), polovina vývojáři a
  polovina zástupci zákazníka
* prostředníkem mezi nimi moderátor - jako katalyzátor pro myšlenky a přání

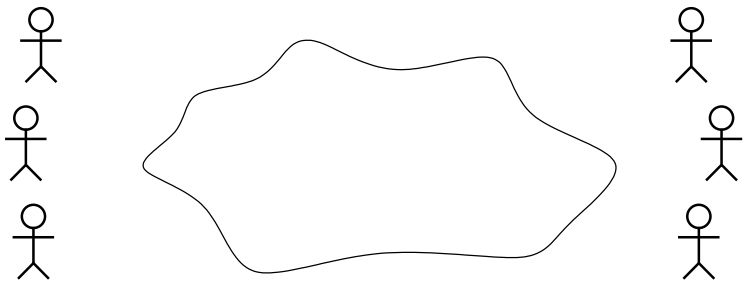
Postup:

* identifikace aktérů
* identifikace případů použití
* vytvoření popisu pro každý případ použití
* popis toku událostí pro každý případ použití
* strukturování případů použití
* identifikace analytických tříd atd.

Postupně probereme:

* identifikace aktérů
 - pokusíme se identifikovat kdo nebo co bude používat systém
 - začneme konkrétními lidmi, pak zkusíme identifikovat roli kterou
   hrají při interakci se systémem (postup od konkrétního k abstraktnímu)
 - tím získáme jména aktérů
 - vždy zaznamenat popis - body zachycující roli vzhledem k systému,
   odpovědnost
 - ”aktéři” jsou i další systémy se kterými náš systém komunikuje
   (ikonka panáčka pro aktéra je zde poněkud mimo)
 - této fázi se nemusíme snažit se aktéra nějak omezovat nebo strukturovat

 - na co se ptát:
  . kdo bude systém používat?
  . z jakých dalších systémů bude náš systém přijímat informace?
  . do jakých dalších systémů bude náš systém dodávat informace?
  . kdo systém spouští?
  . kdo udržuje informace o uživatelích?



 - nakreslete obláček
 - po obou stranách sloupec panáčků, u každého panáčka jméno role

Poznámka:

 * mnoho aktérů může mít svou pevně danou pozici, např. ředitel
 * někdy může pozice odpovídat více rolím, např. sekretářka na KIV může
   mít zodpovědnost za evidenci DP, za přidělování přístupu do laboratoří apod.
   => můžou být dva aktéři systému
 * někdy můžeme dostat návrh ”Měla by tam být taky Helenka”
  - pak je třeba určit její roli nebo role - jméno aktéra by měla být role
  - ptáme se: co je role Helenky? kdo ještě může zastávat tuto roli?
              proč má Helenka tuto roli?

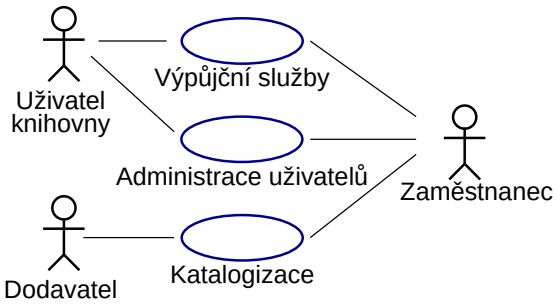
[]

Praktické triky:

* ptáme se zda něco chybí
* navrhujeme špatná řešení, zbytek týmu vás může opravit a vysvětlit
  jaké jsou skutečné role v systému
* vždycky přijměte všechny návrhy - je to brainstorming, tj. kritika ho
  spolehlivě zničí (neaktéry, příliš obecné aktéry jako ”uživatel” a
  vícenásobné výskyty aktérů můžeme vyrušit později)

Pokud se množina aktérů jeví jako úplná, je čas začít s případy použití.

* identifikace případů použití
 - smažte z tabule velký obláček a začněte s definicí případů použití
 - pro každý návrh nakreslete elipsu, popište jí a udělejte čáry nebo
   šipky k aktérům (představují předávané zprávy)
 - popiskou případu použití může být celá věta (zkrátit se může později)
 - možno vycházet z ručního postupu zpracování entit, jako např. jak se
   zpracovává objednávka (abychom lépe vizualizovali, můžeme si
   někam jinam nakreslit schéma/plánek zákazníkova obchodu apod.)
 - zatím se nesnažte strukturovat, i když případy použití budou mít
   společné části (zatím toho o interní struktuře případů použití víme
   příliš málo)



 - po definici revize
  . podívejme se na každého aktéra - co je jeho úlohou v systému?
  . podívejme se na každý případ použití - je zřejmé co aktér dosáhne
    případem použití?
  . je případ použití úplný nebo je to jenom větší část něčeho jiného?
 - každý aktér by měl mít alespoň jeden případ použití
  . pokud ne, může být aktér další výskyt jiného aktéra, tj. duplikát
  . nebo není přímým uživatelem systému
  . pokud diskuse neukáže nutnost zachování aktéra, zrušíme ho

* vytvoření popisu pro každý případ použití
 - zpracujte případ použití jeden po druhém, nejlépe na samostatné čtvrtky
 - nakreslete elipsu a popisku pro případ použití
 - požádejte skupinu o pomoc s vytvořením stručného popisu případ použití
   (1-3 věty)
 - někdy může být užitečné nakreslit aktéry spojené s případem použití
 - spodní polovinu papíru ponechte prázdnou pro další krok

Poznámka:

 Zde se většinou ukáže, že věci které se zdály být jasné ve skutečnosti
 vůbec jasné nejsou - mohou se objevit nové případy použití a některé
 staré mohou zaniknout.

[]

* popis toku událostí pro každý případ použití
 - opět probíráme případ použití jeden po druhém
 - budeme hledat 5 až 10 základních kroků (tím říkáme úroveň podrobnosti)
 - zápis kroků v pořadí, číslujeme 1, 2, 3, ...
 - pak kroky projdeme, identifikujeme alternativní kroky, číslujeme
   např. A1, A2..., nebo a) b) ...
 - nesnažíme se řešit jak bude vypadat kód (smyčky apod.), ale poznamenáme si
   všechny nejasnosti (musíme je vyřešit před vytvořením DSP)

* vytvořte doplňkovou specifikaci
 - pro funkční požadavky které se netýkají žádného případu použití
 - pro mimofunkční požadavky - mohou se týkat vlastnosti konkrétního
   případu použití nebo obecné vlastnosti systému

* pro další zacházení je důležité, co chceme dosáhnout:
 - některé projekty používají případy použití pouze neformálně pro sběr

požadavků, požadavky ale zapisují a uchovávají jiným způsobem
- některé projekty mohou pokračovat dalšími fázemi:
. vytvoříme podrobnější specifikaci případů použití
. případy použití a aktéry strukturujeme (do případu použití můžeme doplnit sekvenci akcí z jiného případu použití apod.)
. z chování případů použití už můžeme identifikovat tzv. analytické třídy atd.
- konkrétní volba závisí na velikosti projektu, dostupných nástrojích atd.

Klasifikace požadavků, detekce a řešení konfliktů

V této fázi se zabýváme následujícími činnostmi:

* nestrukturovanou množinu požadavků se snažíme logicky uspořádat
* rozlišíme funkční, mimofunkční a doménové požadavky, uživatelské a systémové - potřebujeme je oddělit v DSP
* detekujeme a řešíme konflikty mezi požadavky.

Klasifikace a uspořádání požadavků
...

Požadavky mohou být klasifikovány podle různých kritérií, např. na:

* požadavky na funkce a mimofunkční požadavky (viz minulá přednáška)
* podle priority, např. na nutné, žádoucí, vhodné, volitelné (často je zapotřebí vzít v úvahu také cenu vývoje)
* podle rozsahu; např. některé mimofunkční požadavky jsou globální, zatímco některé požadavky na funkce je možné změnit aniž by měly vliv na ostatní fce systému
* podle pravděpodobnosti změny na trvalé a nestálé požadavky; pro vývojáře může být snazší reagovat na změnu, pokud v DSP označíme požadavek jako nestálý

Dále uvedu několik příkladů, jakým způsobem je možné nestrukturované požadavky uspořádat. Nejprve příklad nestrukturovaného požadavku:

2.2.15 Zobrazování mřížky.
2.2.15.1 Aby bylo možné umisťovat entity do diagramu, uživatel může zapnout zobrazování mřížky buď v centimetrech nebo v palcích, a to pomocí volby na řídícím panelu. Na počátku se mřížka nezobrazuje. Mřížka může být vypnuta a zapnuta kdykoli během relace a kdykoli může být přepnuta mezi centimetry nebo palci. Možnost zobrazovat mřížku bude i pro zmenšené pohledy, ale počet řádků mřížky bude omezen tak, aby mřížka při prohlížení zmenšeného obrázku nerušila.

[]

První věta směšuje 3 typy požadavků:

1. Koncepční funkční požadavek - editor má poskytovat mřížku.
2. Mimofunkční požadavek - jednotky mřížky.
3. Mimofunkční požadavek na UI - mřížku bude zapínat a vypínat uživatel.

Výše uvedená specifikace má navíc následující potíže:

* je neúplná - chybí inicializační informace - jakou jednotku bude mřížka používat po zapnutí?
* směšuje uživatelské a systémové požadavky
* zdůvodnění požadavku není vyděleno

Příklad uživatelské specifikace - přepsání s vynecháním detailů:

2.2.15 Mřížka editoru.

2.2.15.1 Editor bude poskytovat možnost zobrazit mřížku, tj. matrici horizontálních a vertikálních čar zobrazených na pozadí

```
          editoru. Mřížka bude pasivní a určování pozice jednotlivých
          elementů bude záležitostí uživatele.

          Zdůvodnění: Mřížka uživateli pomůže uživateli lépe rozmisťovat
          elementy diagramu. I když aktivní mřížka (tj. taková kde se
          elementy "přilepují" k čarám mřížky) může být užitečná, je
          takové umisťování nepřesné. Umístění entit určí nejlépe uživatel.
[]

* důležité uvádět zdůvodnění - pokud dojde ke změně požadavků, můžeme
  určit co vedlo k původnímu požadavku

Další příklad - podrobnější uživatelská specifikace:

2.2.456 Přidání prvku do diagramu.

2.2.456.1 Editor bude poskytovat možnost vložit do diagramu prvek
          určeného typu.
2.2.456.2 Pro vložení prvku bude použita následující posloupnost akcí:
          1. Uživatel vybere typ prvku, který má být vložen.
          2. Uživatel přesune kurzor přibližně na pozici, kam má být
             prvek vložen a signalizuje, že prvek má být na pozici vložen.
          3. Uživatel by měl prvek posunout na jeho konečnou pozici.
          Zdůvodnění: Umístění prvků určí nejlépe uživatel.
[]

* popis obsahuje seznam akcí uživatele (někdy nezbytné), neobsahuje ale
  implementační detaily (např. jak se posouvají symboly).

Detekce a řešení konfliktů
..........................

* pokud zjistíme konflikt (viz také validace požadavků), musíme řešit
 - konflikt může nastat např. pokud dva uživatelé požadují vzájemně
   neslučitelné vlastnosti nebo pokud mezi požadovanými schopnostmi a danými
   omezeními
 - ve většině případů není vhodné, aby rozhodli vývojáři
 - často je důležité, aby konkrétní rozhodnutí bylo možné vysledovat zpět ke
   konkrétnímu zástupci zadavatele (viz také dále - správa požadavků)


Specifikace požadavků
---------------------

Po uspořádání požadavky specifikujeme v DSP (viz výše). Po specifikaci
následuje validace (ověření) požadavků.


Validace požadavků
------------------

* vstupem úplný DSP
* musíme zjistit, zda jsou požadavky úplné, konzistentní a zda
  odpovídají tomu, co zadavatel od systému chce

* co všechno je třeba kontrolovat:
 - platnost požadavků
  . uživatel si myslí, že systém má poskytovat určité funkce, ale
    podrobnější analýza může ukázat něco jiného
  . různí uživatelé mají různé požadavky, bude kompromis platný?
 - konzistenci - požadavky v dokumentu nesmějí být v konfliktu, tj. nesmějí
   být různé popisy nebo různá omezení stejné fce
 - úplnosti požadavků - definovány by měly být všechny fce a omezení systému
 - kontrola realizovatelnosti - zda může být systém implementován, zda může
   být implementován s danými prostředky a v daném čase
 - ověřitelnost - systémové požadavky by měly být napsány tak, aby byly
   ověřitelné (ušetříme si dohadování se zákazníkem při předávání produktu)
 - sledovatelnost původu požadavku - abychom dokázali odhadnout dopad změny
```

Použitelné metody:

* přezkoumání (reviews)
* prototypování
* tvorba testů
* automatická analýza konzistence.

* přezkoumání (reviews) - požadavky jsou systematicky zkontrolovány týmem
 - manuální proces, více čtenářů od zákazníka i od kontraktora
 - může být formální nebo neformální
  . nejčastěji formální přezkoumání DSP = vývojový tým provází klienta systémovými požadavky, vysvětluje důsledky každého požadavku, přitom kontroluje konzistenci, úplnost atd. (konkrétní technika - viz ”Inspekce kódu” v přednášce o testování)
  . neformální = diskuse požadavků s tolika zástupci zákazníka, s kolika je to možné

* prototypování - zákazníkovi předvedeme spustitelný model systému, může zjistit zda odpovídá požadavkům
 - např. chování uživatelského rozhraní zákazník nejlépe pochopí pomocí prototypu
 - nevýhodou prototypů - pozornost uživatele budou odvádět kosmetické záležitosti nebo omezení prototypu
 - proto je v podobných případech doporučováno se SW prototypům vyhnout a použít např. papírový model obrazovek

* generování testovacích případů - pokud vytvoříme testy požadavků, často odhalíme problémy; pokud je obtížné vytvořit test, bude požadavek obtížně implementovatelný

* automatická analýza konzistence - pokud byly požadavky specifikovány jako model ve formální nebo strukturované notaci, můžeme konzistenci modelu zkontrolovat automaticky.

Správa požadavků

* požadavky na velký systém se neustále mění (důvody byly uváděny průběžně)
* správa čili management požadavků je proces řízení změn systémových požadavků

* z hlediska vývoje se požadavky dělí na trvalé a nestálé požadavky
 - trvalé požadavky
  . relativně stabilní, jsou odvozeny ze základní funkce organizace nebo z aplikační domény
  . např. v nemocnici budeme mít vždy pacienty, lékaře a sestry; v bance budeme mít vždy klienty, účty apod.
 - nestálé požadavky
  . pravděpodobně se změní během vývoje nebo po uvedení systému do provozu
  . např. nemocnice - pravděpodobně se změní systém plateb od zdravotních pojišťoven
  . nebo banka - mění se podmínky pro získání úvěru apod.

* management požadavků by měl začít plánováním, v něm se rozhodne
 - způsob identifikace požadavků - každý požadavek by měl mít jedinečný identifikátor, např. číslo, abychom ho mohli odkazovat (křížové reference apod.)
 - proces změny požadavků - definujeme proces, abychom se ke změnám požadavků chovali konzistentním způsobem - viz níže
 - sledovatelnost - které vztahy mezi požadavky navzájem atd. budeme uchovávat a jak (čím více, tím dražší)
  . zdroj požadavku - kdo požadavek navrhl, důvod; abychom se mohli ”zdroje” zeptat na podrobnosti
  . vztahy mezi požadavky v DSP; pro určení kolika požadavků se změna dotkne
  . vztahy mezi požadavky a designem systému; pro určení dopadu změny na systémový design a implementaci
 - jaké nástroje se použijí pro uchovávání informací o požadavcích (malé projekty - postačují obvyklé prostředky jako textové a

tabulkové procesory, databáze; velké projekty - CASE nástroje)

Sledovatelnost požadavků
.....

Sledovatelnost požadavků (traceability) znamená možnost sledovat sledovat požadavky zpět k jejich zdroji (např. z DSP zpět k požadavku v ConOps dokumentu, ze kterého vyplývá), dopředu k návrhu nebo nebo SW artefaktu který ho implementuje (např. z DSP k digramu tříd nebo ke komponentě), nebo případně závislosti požadavků mezi sebou navzájem.

* jedna možnost - matice závislostí požadavků - mapuje např. vzájemné závislosti požadavků
 - prvky - jak závisí požadavek v řádku na požadavcích daných sloupcem
 - U (Uses) - požadavek v řádku používá možnosti dané požadavkem
 - R (Relates) - nějaký slabší vztah, např. oba části stejného podsystému

Id pož.	1.1	1.2	1.3	1.4	1.5
1.1	.	U	R	.	.
1.2	.	.	U	.	.
1.3	R	.	.	.	.

* matice závislosti lze používat při malém množství požadavků, ale pokud je požadavků mnoho, byla by její údržba drahá
* pak by závislosti měla zachycovat přímo databáze požadavků - součást CASE nástrojů pro správu požadavků

Poznámka (nástroje pro správu požadavků)

Moderní nástroje pro správu požadavků často umí vygenerovat matici nebo graf závislostí, graficky znázornit propagaci změn, generovat zprávy o stavu požadavků, generovat DSP podle zvoleného standardu apod.

[]

Proces změny požadavků
.....

* všechny navrhované změny požadavků by měly podléhat procesu o třech základních krocích:
 - analýza problému a specifikace změny
 - analýza změny a určení její ceny
 - implementace změny

* analýza problému a specifikace změny
 - identifikujeme problém některého požadavku nebo dostaneme návrh změny
 - zjišťujeme zda je problém nebo změna požadavku platná
 - výsledkem může být podrobnější návrh změny požadavku

* analýza změny a určení její ceny
 - zjistíme důsledky změny (k tomu se nám hodí mít informace o závislosti požadavků atd.)
 - určíme jakou změnu DSP, případně i designu a implementace by bylo třeba provést
 - odhadneme cenu změny, případně odhad nového termínu dokončení
 - po dokončení analýzy padne rozhodnutí, zda budeme pokračovat realizací změny (Přijdeme za zákazníkem: Stálo by to X, budete to chtít teď nebo později?)

* implementace změny
 - modifikujeme DSP, případně design a implementaci

Praktická poznámka:

Pokud je požadavek na změnu urgentní, je tlak na to provést změnu nejdříve v implementaci a pak zpětně modifikovat DSP. To nevyhnutelně vede k tomu, že se DSP a implementace rozejdou: na začlenění změny do DSP se buď zapomene nebo DSP je změněn nekonzistentně se změnou implementace.

❇