Kapitola 20

Analýza a návrh softwarových systémů

Kapitola 21

Zdroje

- Materiály přednášky Software design and implementation na univerzitě v Severní Carolině
- Ian Sommerville — Software Engineering
- Podklady k přednášce Modelování a realizace programových systému Karla Richty
- Formální metody specifikace – příklady, další zdroje

Kapitola 22

Zpracování jednotlivých otázek

22.1 Algebraické specifikace, formální popis datových struktur

Příklad — množina INTů:

```
sorts: SET, INT, BOOLEAN          | vsechny vyuzivane typy
operations:                       | popisuji syntaxi daneho ADT
  - empty:            --> SET     | generator
  - insert: SET x INT --> SET     | generator
  x delete: SET x INT --> SET
  x member: SET x INT --> BOOLEAN
axioms:                           | popisuji semantiku specifikovanych operaci
  member(empty(),j) = false       | axiomy pro vsechny co nejsou generator se vsema moznyma ..
  member(insert(S,j),k) =         | generatorama na vstupu
    if (j=k) then true else member(S,k)
  delete(empty(),j) = empty()
  delete(insert(S,j),k) =
    if (j=k) then delete(S,j)
             else insert(delete(S,k),j)
```

22.2 Modelově orientované metody

Z

- Zdroj: The Z Notation: a reference manual

Formální specifikace je matematický popis (informačního) systému, který přesně popisuje, co má systém dělat, ale neříká jak.

Jazyk Z používá "matematické" datové typy popsané pomocí predikátové logiky (nezávislé na počítačové reprezentaci).

Dekomponuje specifikaci do malých částí zvaných schémata, ty popisují statické i dynamické aspekty systému.

- statické aspekty:
  - stavy
  - invarianty
- dynamické aspekty:
  - možné operace
  - vztahy mezi vstupy a výstupy
  - změny stavů

Schéma může popisovat i transformaci jednoho pohledu na systém na jiný, který přidává více detailů (při korektní implementaci původní abstrakce). Postupným zjemňováním (refinement) specifikace lze dospět až ke konkrétnímu programu.

Schéma

Každé schéma má svůj název a obsahuje:

- deklarace proměnných
- vztahy mězi proměnnými (odděleno vodorovnou čárou)

Příklad:

- (převzat ze zdroje)

Základní typy pro účely příkladu: DATE
Schéma popisující "prostor stavů" (state space): <table border style="border-collapse: collapse;"> <tr><th>BirthdayBook</th></tr> <tr><td> known: P(NAME) // množina
birthday: NAME → DATE // funkce </td></tr> <tr><td> known = dom birthday </td></tr> </table>
Schéma popisující operaci: <table border style="border-collapse: collapse;"> <tr><th>AddBirthday</th></tr> <tr><td> ΔBirthdayBook // Δ značí, že operace mění BirthdayBook (konvence)
name? : NAME // ? označuje vstup (konvence)
date? : DATE </td></tr> <tr><td> name? ∉ known
birthday' = birthday ∪ {name? → date?} // apostrof označuje novou hodnotu </td></tr> </table>

- operace zachovává invarianty předchozího schématu
- podobně jako ? označuje vstupní proměnné, ! označuje výstupní
- podobně jako Δ značí operaci modifikující stav, Ξ (Xí) značí operaci zachovávající stav

Skládání schémat

Schémata lze skládat pomocí operátoru =^ (rovnítko se stříškou) a logických operátorů (∧ a ∨).
Enum (pro účely příkladu):
REPORT ::= ok | already_known | not_known
Složené schéma:
RAddBirthday =^ (AddBirthday ∧ Success) ∨ AlreadyKnown;

- schéma Success vrací ok (typu REPORT), AlreadyKnown vrací already_known (pokud name? ∈ known)

<table border style="border-collapse: collapse;"> <tr><th>Success</th></tr> <tr><td>result! : REPORT</td></tr> <tr><td>result! = ok</td></tr> </table>

- složené schéma RAddBrithday vrací ok v případě úspěchu AddBirthday a already_known jinak
- RAddBrithday by šlo nadefinovat přímo, ale jeho definice vztahů by byly nepřehledně složité

Další syntax

- ∀ z : N • x $\leq$ z
  - znamená x je menší než všechna přirozená čísla (neboli x je nula)
- birthday' = birthday ⊕ {name? → date?}
  - předefinovává funkci birthday v bodě name? na novou hodnotu (za opertárotrem ⊕ může být i více-prvková množina)

(Více viz zroj...)

VDM

- zdroj: wen:Vienna Development Method

Skupina technik založená na specifikačním jazyce (VDM-SL).
Má variatnu VDM++, která podporuje objektově orientované a "concurrent" systémy.
Umožňuje modelování na vysoké úrovni abstrakce a převod na implementaci pomocí zjemění specifikace (refinement). Má spustitelnou podmnožinu, kterou je možné testovat.
Vydáno jako ISO standard 1996 (ten definuje ASCII syntax pro věci, co jsou jinak popsány "matematickou" syntaxí).
Jazyk je hezky popsán ve wikipedii (wen:Vienna Development Method).

22.3 Analýza algoritmů

- **validace** = "Are we building the right product?"
  - nebo jinými slovy — jde o kontrolu, zda-li daný produkt odpovídá reálným požadavkům
- **verifikace** = "Are we building the product right?"
  - neboli — jde o kontrolu, zda-li daný produkt odpovídá výchozí specifikaci

Hoareova logika

- wen: Hoare logic
- ...v příkladech by David Stotts (UNC)
- cílem je, aby se dala formálně dokazovat korektnost programů pomocí rigorózních prostředků matematické logiky
- základem je Hoare triple, popisující jak kousek kódu změnil stav výpočtu — $\{P\}C\{Q\}$
  - kde P a Q jsou assertions a C je příkaz. P nazýváme precondition, Q postcondition. Obojí jsou formule predikátové logiky.
- Hoaerova logika (dále jen HL) obsahuje axiomy a odvozovací pravidla pro všechny konstrukty jednoduchého imperativního programovacího jazyka
- Standardní HL poskytuje pouze **partial correctness**, neboli říká, že pokud před provedením C platí P, pak po jeho provedení platí Q, **nebo** C neskončí. Existuje ale rozšíření poskytující **total correctness**.

Pravidla jsou:

- Empty statement axiom schema:

$$\frac{}{\{P\}\ \textbf{pass}\ \{P\}}$$

- Assignment axiom schema:

$$\frac{}{\{P[x/E]\}\ x := E\ \{P\}}$$

- Rule of composition:

$$\frac{\{P\}\ S\ \{Q\}\ ,\ \{Q\}\ T\ \{R\}}{\{P\}\ S;T\ \{R\}}$$

- Conditional rule:

$$\frac{\{B \wedge P\}\ S\ \{Q\}\ ,\ \{\neg B \wedge P\}\ T\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{endif}\ \{Q\}}$$

- While rule:

$$\frac{\{P \wedge B\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ \{\neg B \wedge P\}}$$

- Consequence rule:

$$\frac{P' \rightarrow\ P\ ,\ \{P\}\ S\ \{Q\}\ ,\ Q \rightarrow\ Q'}{\{P'\ \}\ S\ \{Q'\}}$$

- While rule for total correctness:

$$\frac{\{P \wedge B \wedge t{=}z\}\ S\ \{P \wedge t{<}z\}\ ,\ P \rightarrow t{\geq}0}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{done}\ \{\neg B \wedge P\}}$$

Jak se to celé používá je hezky vysvětleno tady, nebudu se to snažit zreplikovat.

Dynamická logika

- viz wen: Dynamic logic (modal logic)
- Navrhl Vaughan Pratt v roce 1974
- Je rozšířením modální logiky zaměřené na dedukci o počítačových programech (ovšem později se začla využívat i jinde)
- Modální logika je charakteristická svými dvěma modálními operátory:
  - $\Box p$ — říká že p platí vždy
  - $\Diamond p$ — říká že p může platit
    - platí pro ně $\Box p \leftrightarrow \neg\Diamond\neg p$ a $\Diamond p \leftrightarrow \neg\Box\neg p$
- DL přidává ke dvou termům logik prvního řádu (tvrzení a data) ještě třetí typ termu — akci
- Dynamická logika (dále jen DL) toto rozšiřuje přidáním dvou modálních operátorů pro akce:
  - $[a]$, kde $[a]p$ znamená, že po provedení akce a musí p platit
  - $\langle a\rangle$, kde $\langle a\rangle p$ znamená, že po provedení akce a může p platit
    - opět samozřejmě platí $\langle a\rangle p \leftrightarrow \neg[a]\neg p$ a $[a]p \leftrightarrow \neg\langle a\rangle\neg p$
- DL umožňuje skládání akcí, pro jejich zápis se dá využít notace regulárních výrazů:
  - $a|b$.. provede se a nebo b
  - $a;b$.. provede se nejprve a, pak b
  - $a*$.. a se provede 0 nebo vícekrát
- Dále jsou k dispozici konstantní akce:
  - 0.. nic neudělá a neskončí (aka BLOCK)
  - 1.. nic neudělá a skončí (aka NOP)
- Krom defaultních jsou definovány tyto axiomy:
  - A1. $[0]p$
  - A2. $[1]p \leftrightarrow p$
  - A3. $[a|b]p \leftrightarrow [a]p \wedge [b]p$
  - A4. $[a;b]p \leftrightarrow [a][b]p$
  - A5. $[a*]p \leftrightarrow p \wedge [a][a*]p$
  - A6. $p \wedge [a*](p \rightarrow [a]p) \rightarrow [a*]p$ (s akcí n:=n+1 odpovídá matematické indukci)
  - speciální axiomy
  - A7. axiom schema $[x := e]\Phi(x) \leftrightarrow \Phi(e)$, odpovídá přiřazení z Hoareovy logiky. Tedy $\Phi(e)$ odpovídá fli ve které jsou všechny výskyty x nahrazeny výrazem e.
  - A8. $[p?]q \leftrightarrow p \rightarrow q$, kde p? je speciální akce definována ke každému tvrzení taková, že p? odpovídá NOP pokud je p true, jinak je to BLOCK. If p then a else b se dá tedy zapsat jako $(p?;a)|(\neg p?;b)$
- $x :=?$ znamená přiřazení libovolné hodnoty do x, a tedy $[x :=?]$ odpovídá obecnému kvantifikátoru, zatímco $\langle x :=?\rangle$ odpovídá existenčnímu

$\{p\}a\{q\}$ z HL se dá v DL zapsat jako $p \rightarrow [a]q$

Stejně jako HL se v ní nedají elegantně vyjádřit konkurentní chování (zatímco sekvenční se zapisuje hezky). To ovšem jde v TL (temporální).

Temporální logika

- viz wen:Temporal logic
- jestli někdo vůbec nemá tucha o TL, tak tady je to celkem hezky popsané, i když trochu delší http://www.cs.utexas.edu/user
- přináší prvek času, pravdivost tvrzení se tedy může v závislosti na čase měnit
  - například "mám hlaď" je pravdivé jenom někdy (třeba teď ;), díky temporální logice můžeme být proto přesnější a říct "budu mít hlad dokud se nenajím"
  - přínos pro formální verifikaci je zřejmý — snadno můžeme zapsat tvrzení typu "kdykoli příjde požadavek, zařízení může být zpřístupněno, ale v žádném případě nebude současně zpřístupněno dvěma žadatelům"
- je několik variací — **linear temporal logic** (AKA LTL, existuje jen jedna časová linie) a **branching logic** (více možných časových linií = nedeterminismus, zřejmě má stejnou sílu jako lineární)
- dva druhy operátorů
  - logical operators (klasika: $\neg, \vee, \wedge, \rightarrow$)
  - modal operators (operátory, které říkají například že fle musí někdy v budoucnu platit, nebo že musí platit odteď nafurt)

Modální operátory v LTL jsou následující (přeloženo z wikipedie):

Textově
N ϕ
G ϕ
F ϕ
ψ **U** ϕ
ψ **R** ϕ

Dalo by se zredukovat na dva z těchto operátorů, protože vždy platí:

- **F** ϕ = **true U** ϕ
- **G** ϕ = **false R** ϕ = $\neg$ **F** $\neg\phi$
- ψ **R** ϕ = $\neg(\neg\psi$ **U** $\neg\phi)$

22.4 Petriho sítě

- viz wen: Petri net
- Interactive Tutorials on Petri Nets
- matematická reprezentace diskrétních distribuovaných systémů
- vynalezl je Carl Adam Petri ve své Ph.D práci v roce 1962
- orientované bipartitní grafy ze dvěma druhy uzlů — místa a přechody
- nedeterministické! (není přesně definováno ktery z možných přechodů se nastartuje)

Formálně: Petriho síť je pětice (P, T, F, M_0, W), kde:

- P je seznam míst
- T je seznam přechodů
- F je seznam hran (žádná hrana nesmí spojovat dvě místa nebo dva přechody, tedy $F \subseteq (P \times T) \cup (T \times P)$)
- $M_0 : P \rightarrow \mathbb{N}$ je iniciální označkování, v každém místě $p \in P$, existuje $n_p \in \mathbb{N}$ tokenů
- $W : F \rightarrow \mathbb{N}^+$ je množina váh hran, které každé hraně $f \in F$ přiřazují $n \in \mathbb{N}^+$ které označuje kolik tokenů je při přechodu danou hranou navazujícím přechodem zkonzumováno, případně kolik tokenů výchozí přechod generuje
- stav sítě je vektor, kde v každé položce je počet tokenů aktuálně obsažený v místě s příslušným indexem

- další vlastnosti
  - state-transition list
    - seznam postupně navazujících přechodů mezi možnými stavy sítě
  - state-transition matrix
  - reachability
    - otázka, zda-li je daný stav dosažitelný z výchozího. Řeší se pomocí kreslení grafu dosažitelnosti (kde každý vrchol je možný stav a hrana je přechod mezi stavy). Graf se musi budovat do šířky, protože může být nekonečný a tak bychom se při cestě do hloubky snadno mohli stratit ..
  - liveness
    - vlastnost přechodů. Pokud je každý přechod L_k *live*, pak totéž můžeme říct o síti jako celku.
    - přechod je:
      - L_0 *live* (AKA dead), právě když nemůže být fired (neexistuje firing sequence která by ho obsahovala)
      - L_1 *live*, právě když může být fired
      - L_2 *live*, právě když pro každé celé kladné k může být fired k-krát
      - L_3 *live*, právě když existuje firing sequence kde je přechod fired nekonečně
      - L_4 *live* (AKA live), právě když je přechod v každém dosažitelném stavu L_1 *live*
  - boundedness
    - síť je k-omezená, pakliže v každém dosažitelném stavu má v každém místě vždy nejvýše k tokenů. Petriho síť je ohraničená, pakliže má její graf dosažitelnosti konečný počet vrcholů.
- možná rozšíření
  - barevné petriho sítě (tokeny je možno rozlišovat)
  - prioritizované petriho sítě (přechody mají priority)
- hierarchie petriho sítí
  - **state machine** (každý přechod má 1 in a 1 out hranu => může vzniknout konflikt)
  - **marked graph** (každé místo má 1 in a 1 out hranu => může vzniknout konkurence)
  - **free choice** (hrana je buď jediná která vystupuje z místa, nebo jediná která vstupuje do přechodu => může být konflikt i konkurence, ale ne najednou)
  - **extended free choice** (taková, která se dá transformovat na FC)
  - **asymetric choice** (konkurence i konflikt povoleny, ale ne asymetricky)
  - **petri net** (všecko povoleno)

22.5 Vyjadřovací prostředky a metody návrhu IS

Celý název tématu: Vyjadřovací prostředky a metody (datové modelování, procesní modelování — funkční a dynamické) strukturované analýzy a návrhu informačních systémů

Pro velkou provázanost zde popsáno spolu s konceptuálním modelováním, E-R schámaty a 3NF.

Společné zdroje:

- Konceptuální modelování a návrh databáze (slajdy VUT Brno; pdf)

Datové modelování

- wen: Data modeling

Zabývá se vytvářením datového modelu `informančního` sytému (na základě požadavků zadavatele či klienta). Entity datového modelu reprezentují objekty reálného světa. Datový model může být různého typu, např. objektový nebo relační. Model se tá popisovat různými formalismy (např. pomocí teorie množin a lambda kalkulu), dnes se typicky používá UML (model tříd).

Konkrétnímu modelu nějakého typu se říká instance modelu. Instance datového modelu může být tří typů (úrovní). (Jde o mírně historický pohled – ANSI, 1975):

- konceptuální model – popisuje doménu problému, vyjmenovává třídy entit a relace mezi nimi
- logický model – přidává sémantiku (dle použité technologie, např. definuje sloupce tabulek nebo třídy OO modelu)

- fyzický model – přidává detaily fyzického uložení (diskové oddíly, tablespaces, ...)

Idea rozdělení je, že úrovně jsou relativně nezávislé (např. konceptuální model může mít podobu ER-diagramu a logický může být OO modelem), v ranné fázi modelování lze pracovat jen s konceptuálním modelem, pak zpřesňovat.

Viz také:

- #E-R schémata a jejich transformace do relačního modelu
- wen:Relational model
- UML (diagram tříd)

Procesní modelování (funkční a dynamické)

Procesní modelování má využití v různých případech. Modelujeme stávající procesy organizace, která poptává IS. V tomto případě jde o deskriptivní procesní model, slouží pro analýzu požadavků. Užitečný je i model popisující procesy po nasazení IS (preskriptivní model). Další využití procesního modelování je v optimalizaci obchodních procesů (příp. pouze k jejich dokumentaci).

Procesní model zachycuje procesy, je dobré aby alespoň hlavní proces měl definován cíl. Jednotlivé procesy zahrují:

- začátek (iniciální aktivita; v aktivity diagramu UML plný puntík)
- aktivity (v UML zobrazeny jako obdelníčky se se zakulacenými rohy)
- vstupy/výstupy (signály; obdélníček s levou/pravou stranou ve tvaru šipky doprava)
- rozhodování (kosočtvereček)
- (aktéři) (panáček)
- konec (konečná aktivita, může jich být víc; zakroužkovaný puntík)

(V předmětu Vedení DB aplikací a jazyk UML jsme kreslili aktivity (i samotné procesy) jako obdélníčky s oběma bočními stranami ve tvaru šipek.)

Používané notace:

- wen:Business Process Modeling Notation (BPMN; by OMG)
- UML (activity diagram)

TODO: funkční vs. dynamické ??

Konceptuální modelování, databázové modelování, implementace

- Viz #Datové modelování.

E-R schémata a jejich transformace do relačního modelu

- wen:Entity-relationship model

Transformace

- odstranění (rozepsání) složených a vícehodnotových atributů (1NF)
- převod silných entit na tabulky (relace)
- převod relací:
  - entity v relaci 1:1 lze reprezentovat jednou tabulkou
  - relaci 1:N reprezentujeme pomocí cizího klíče
  - relaci M:N pomocí relační tabulky
- reprazentace slabých entit pomocí tabulky a cizího klíče (jako 1:N)
- možnosti řešení dedičnosti:
  - tabulky pro nadtyp i podtypy
  - tabulky jen pro podtypy s opakováním sloupců
  - vše v jedné tabulce (+ rozlišovací flag)

Návrh relačních schémat v 3NF

- hezky a stručně popsané tady: Third Normal Form (3NF) (na James Cook University)
- a nebo jako obvykle — wen: Third normal form
- a nebo taky tady: Schema Refinement and Normalization (na Římské Univerzitě)
- Armstrongova pravidla
  - X je částí Y pak X -> Y
  - X -> Y pak XZ -> YZ pro každé z
  - X -> Y a Y -> Z pak X->Z

1NF

- Všechny atributy jsou jednoduchého typu
- (Takže třeba atribut datum, který se skládá ze dne měšíce a roku to nesplňuje).

2NF

- Pokud je v 1NF a pro kterýkoli wcs:kandidátní klíč je každý neklíčový atribut závislý na celém tomto klíči (nikoli na jeho části)
- (neboli žádný neklíčový atribut není závislý na části klíče; implikuje, že pokud v tabulce složené klíče nejsou, je tabulka ve 2NF)
- To nám zaručuje, že v tabulce nebudou pohromadě nesouvisející věci.
- příklad:
  - (student_id, student_name, class_id, class_name) se závislostmi student_id->student_name a class_id->class_name není v 2NF (protože klíč je [student_id, class_id] a například student_name není na tomto klíči plně závislé).

3NF

- Pokud je 2NF a každý neklíčový atribut je netranzitivně závislý na každém klíči (neboli mezi neklíčovými atributy nesmí existovat jiná než triviální závislost)
- Relace která není v 3NF se bude patrně týkat více věcí a hrozí tedy aktualizační anomálie.
- příklad:
  - (class_id, instructor, office) se závislostmi class_id->instructor, instructor->office není v 3NF (protože office je na klíči class_id závislá tranzitivně .. když bude tedy instruktor přiřazen více předmětům, přestěhuje-li se, musíme v této tabulce změnit všchny záznamy office s ním spojené)
- často v praxi stačí
- na rozdíl od BCNF se už nezabývá závislostmi mezi klíči

BCNF

- wen: BCNF
- Pokud platí X->A a A není v X, pak X je klíčem.
- příklad:
  - (s_id, s_name, c_id, c_name, date) s klíčí [s_id, c_id], [s_id, c_name], [s_name, c_id], [s_name, c_name] a existují závislosti s_id->s_name, c_id->c_name není BCNF (protože např. klíčový atribut s_name je funkčně závislý na klíčovém atributu s_id z jiného klíče).
- indikuje, že některé atributy se netýkají celku který je identifikován klíči (a tyto by měly být jinde)

TODO (?):

- **Dekompozice**
- **Syntéza**

22.6 Modely životního cyklu softwarových systémů

- hlavní zdroj — Ian Sommerville — Software Engineering (konkrétně 6tá edice, link vede na 8mou)
- **Waterfall model**
  - postupně se prochází přesně definoványmi fázemi:

  1. analýza a definice požadavků
  2. design systému a softwaru
  3. implementace a unit testing
  4. integrace a testování systému
  5. provoz a údržba

  - nevýhody
    - neflexibilní (teoreticky by se nemělo zpětně zasahovat do ukončených fází)
  - výhody
    - v každé fázi je dobře definováno co se bude dělat a z čeho vycházet
    - snazší pro management (uz na začátku se dá rozumně plánovat, ví se co se bude dít)
    - dá se očekávat robustní návrh systému
- **Evolutionary development**
  - idea je udělat jednoduchou verzi systému, která bude následně zpřesňována spolu s požadavky uživatele
  - 2 typy
    - exploratory development — snahou je spolu s uživatelem postupně prozkoumávat požadavky a postupovat směrem k cílovému systému (vždy se udělá to čemu všichni dobře rozumí)
    - throw-away prototyping — narozdíl od předchozího se tady zaměřujeme spíš na špatně specifikované části systému a ty se snažíme pomocí prototypů lépe definovat
  - nevýhody
    - špatně se kontroluje průběh procesu
    - často vznikne špatně strukturovaný systém (časté opravy a zásahy do předchozích částí, aby mohly být ty nově definované dodělány)
    - často jsou potřeba speciální nástroje
  - výhody
    - vyplatí se zejména u menších projektů (míň než 100kloc)
- **Formal systems development**

  1. požadavky a specifikace jsou detailně popsány v některém formálním jazyku
  2. další kroky jsou nahrazeny postupnými transformacemi až do podoby spustitelného kódu (postupně jsou tedy přidávány detaily které zpřesňují specifikaci a zároveň neporušují její predikáty)

  - výhody
    - jednotlivé malé kroky jsou snadno vystopovatelné
    - "snadno" se dá dokázat správnost programu
  - nevýhody
    - potřeba speciálních skillů (rozhodnot o následující transformaci je složité)
- **Re-use oriented development**

  1. analýza komponent
  2. úprava požadavků podle definovaných komponent
  3. design systému s využitím komponent
  4. vývoj ostatních potřebných částí a jejich integrace

  - výhody
    - redukce kódu který se bude psát (=snižování rizika chyby)
  - nevýhody

    * výsledek nemusí 100% odpovídat výchozím požadavkům
    * v případě využití COTS (Commercial Off-The-Shelf) komponent vzniká závislost na dalším vendorovi

- **Incremental development**
  - myšlenka je umožnit uživateli učinit některá potřebná rozhodnutí až v době kdy bude mít jasno

  1. definice požadovaných služeb systému, přidělení priorit
  2. definice počtu jednotlivých inkrementů, rozdělení jednotlivých služeb k inkrementům které se budou dělat
  3. v každém inkrementu se pak detailně dospecifikují služby které se budou dělat a proběhne jejich vývoj, výsledek je nasazen k uživateli

  - * v každém inkrementu se může na danou část použít jiná metodika vývoje

- 1. v dalších fázích jsou pak nově vzniklé inkrementy integrovány do stávajícího systému

  - z této metody vychází extreme programming (kde jsou dané myšlenky dotažené do extrému ;)
  - výhody
    * zákazník nemusí čekat až do konce, než bude moct aspoň něco používat
    * zákazník může uplatnit u pozdějších inkrementů experience, kterou nabyde používáním těch předchozích
    * menší riziko celkového selhání
    * nejdůležitější části systému jsou nejlépe otestované (byly vytvořené v ranných inkrementecha a s každým přírůstkem testovány znovu a znovu)
  - nevýhody
    * inkrementy mají být malé a může být složité na ně namapovat požadavky customera
    * některé komponenty budou využívány mnoha částmi systému a zpočátku nemusí být přesně vidět jejich všechny požadované vlastnosti

- **Spiral development**
  - místo popisu procesu jako sekvence ativit s backtrackingem zpět je využito spirály
  - každá jedna otočka spirály se skládá ze čtyř sektorů:

  1. stanovení cílů aktuální smyčky — identifikace omezení a rizik, stanovení plánů
  2. ohodnocení a redukce rizik (definovaných dříve)
  3. vývoj a validace — výběr metody v závislostí na rizicích (pokud je riziko třeba špatné UI, budeme prototypovat)
  4. plánování — revize otočky spirály a plány co dál

  - na rozdíl od ostatních se v tomto modelu explicitně pracuje s riziky

22.7 Plánování a řízení projektů

- hlavní zdroj — Ian Sommerville — Software Engineering (konkrétně 6tá edice, link vede na 8mou)
  - další odkazy:
    * wen: Program Evaluation and Review Technique
- project management je iterativní proces, který končí až končí vlastní projekt (!)

vše by se dalo popsat následujícím pseudokódem:

```
Establish the project constraints
Make initial assesments of the project parameters
Define project milestones and deliverables
while project has not been completed or cancelled loop
  Draw up project schedule
  Initiate activities according to schedule
  Wait ( for a while )
  Review project progress
  Revise estimates of project parameters
  Update the project schedule
```

```
  Renegotiate project constraints and deliverables
  if ( problems arise ) then
    Initiate technical review and possible revisions
  end if
end loop
```

- **Plán projektu**
  - úvod — cíle projektu a omezení
  - organizace projektu — popis organizace týmu
  - analýza rizik
  - definice hw a sw požadavků
  - dekompozice práce
  - harmonogram projektu
  - popis monitorování a reportingu
- milestone — ukončení nějaké podfáze projektu, vznikají při něm výstupy pro management
- deliverable — výstup který se předává zákazníkovi (specifikace, design, ..)
- deliverable je milestone, ale milestone nemusí být deliverable (u milestone totiž může jít o nějaké interní dokumenty)
- **Rozvrhování projektu**
- první rozvrh je vždycky moc optimistický, je proto třeba ho průběžně aktualizovat
- jde hlavně o rozpad jednotlivých prací na aktivity a jejich rozvrhnutí tak, aby ty co mohou běželi paralelně a naopak navazovaly ty které jsou závislé
- ale důležitý není jenom čas, je potřeba zajistit taky dostatek prostředků (ať už jde o lidi, nebo třeba místo na disku)
- nástroje
  - activity graph — graf návaznosti aktivit (task = čtvereček, milestone = kolečko, závislost = hrana)
    - kritická cesta — nejdelší cesta ze startu do cíle, její natáhnutí znamená natáhnutí projektu
  - gannt chart (aka activity bar chart) — horizontální osa ukazuje čas, sloupečky pak jednotlivé tasky
    - popisují stejnou informaci jako acivity graph
    - dají se v nich líp znázornit rezervy tasků
- **Řízení rizik**
- riziko = pravděpodobnost že nastane některá nežádoucí událost
- základní rozdělení rizik
  - project risks — rizika ohrožující plán či zdroje projektu
  - product risks — rizika ohrožující výkon nebo kvality vyvíjeného produktu
  - business risks — rizika ohrožující organizaci nebo vlastní vznik softwaru
- risk management je opět iterativní proces, který se musí dít v průběhu projektu, nejen na začátku (!)
- základní kroky risk managementu
  - risk identification — identifikace možných rizik (jako: technology risks, people risks, organisational risks, tools risks, req. risks, estimation risks)
  - risk analysis — zhodnocena pravděpodobnost a dopad jednotlivých rizik
    - nejde o přesná čísla, ale spíš rozsah psti (jako: very low .. $< 10\%$, low .. 10 — 25%, moderate .. 25 — 50%, high .. 50 — 75%, very high .. $> 75\%$)
    - u dopadu podobně (jako: catastrophic, serious, tolerable nebo insignificant)
    - výsledkem je prioritizovaná (a průběžně aktualizovaná) tabulka rizik
  - risk planning — výběr strategie řízení každého z rizik
    - minimalizace psti.
    - minimalizace dopadu
    - přijmutí rizika (když nastane, budu se s tím umět vyrovnat)
    - přesun rizika na někoho jiného (pojištění)
  - risk monitoring

Alokace zdrojů

- DOKONČIT
- když máme připraven rozpad práce na tasky, je potřeba přiřadit zdroje nutné k vykonání daného tasku
  - tzn. třeba i lidi (i když označovat je za zdroje není moc hezké ;)
- ve velkých společnostech může být problém s pracovníky kteří jsou specialisti — prodloužení jeho práce na jiném projektu může zasáhnout i do našich plánů
- ke znázornění se hodí výše popsané activity graphs a gant charts

Použití metrik

- cílem je získat pro nějaký atribut sw produktu numerickou hodnotu
  - porovnávaním takových hodnot se dá sledovat např. vývoj kvality
- problém je že neexistují žádné obecně uznávané standardy a tedy ani rozšířené tooly
- příklady:
  - počet řádek kódu
  - počet reportovaných chyb ve výsledném produktu
  - počet člověko-dní potřebných k vývoji komponenty
- wen: COCOMO = COnstructive COst MOdel — slouží k odhadu počtu člověko-měsíců, které bude vývoj sw. produktu trvat
- **control metrics** — metriky asociované s procesem
  - např. jak dlouho trvá odstranění nalezeného defektu
- **predictor metrics** — metriky asociované s produktem
  - např. počet atributů a operací třídy, nebo průměrná délka identifikátorů
- extrení vlastnosti software se nedají přímo měřit
  - jde třeba o komplexnost či pochopitlenost
- proto je snaha najít vztah mezi vnitřními (velikost software) a vnějšími metrikami
- k tomu je třeba:
  - přesné měření interního atributu
  - existence vztahu mezi tím co umíme změřit a ext. atributem
  - pochopení daného vztahu a jeho vyjádření v podobě vzorce nebo modelu
- **proces měření**
- fáze:
  - výběr měření která se budou provádět — je třeba vědět co chci změřit
  - výběr komponent k posouzení
  - měření charakteristik komponent
  - identifikace měření vybočujících z řady
  - analýza vybočujících komponent
- klíčové ovšem je naměřená data zaznamenávat a mít k dispozici při dalších měřeních
- **metriky produktů**
- dají se rozdělit na:
  - dynamic metrics — měřené na běžícím programu
  - statické metriky — měření nějaké reprezentace systému (program, dokumentace)
- příklady:

  - fan-in — počet volání fce X (hodně znamená že je fce úzce svázaná se zbytkem)
  - fan-out — počet fcí které jsou z fce X volané
  - lenght of code — větší kód = komplexnější = náchylnější k chybám
  - cyclomatic complexity — měří komplexnost programu
  - lenght of identifiers — delší = víc samovysvětlující = vyšší čitelnost
  - depth of conditional nesting — větší hloubka = horší srozumitelnost
  - fog index — čím delší slova a věty v dokumentu, tím je méně srozumitelný (třeba pro dokumentace)

Řízení kvality

- **quality assurance** — zřízení frameworku organizačních procedur, které vedou k vysoké kvalitě
- **quality planning** — výběr patřičných procedur z daného frameworku a jejich adaptace pro specifický projekt
- **quality control** — definice a ustanovení procesů, které zaručují, že jsou procedury a standardy dodržovány vývojovými teamy
- o quality management by se měl starat nezávislý team
- quality management by měl být oddělen od project managementu
  - neměl by být tedy závislý na nějakých schedulech a budgetech
- ke kvalitě existuje sada standardů ISO 9000 (ISO 9000-3 se vztahuje přímo k sw. vývoji)
  - ISO 9000 -> jeho instancí je Organisation quality process -> jeho instancí je Prject quality management
- dva typy QA standardů
  - product standards
  - process standards
- časte jsou standardy považovány za zbytečnou byrokracii (a každý se snaží najít důvod proč nejsou v tom kterém projektu potřeba)
- jak lidi přesvědčit?
  - zapojit je do vývoje standardů — musí rozumět motivaci
  - průběžně standardy aktualizovat podle aktuálních potřeb
  - poskytnout tooly pro podporu
- standardy dokumentace
- jsou důležité, protože dokumentace je jediná hmotná součást vyvíjeného software
- jde o:
  - document process standards — proces v němž dokument vzniká
  - document standards — definují strukturu dokumentů
  - document interchange standards — zaručují kompatibilitu dokumentů
- quality planning
- musí definovat co přesně vysoká kvalita znamená (což je u sw někdy složité — viz požadavek na dobrou udržovatelnost)
  - problém je, že výroba sw. je spíš kreativní proces, takže hodně záleží na tom kdo to dělá
- quality plan by měl být as short as possible (jinak ho nikdo nebude číst)
- je hodně parametrů které je možné optimalizovat (jako: safety, robustness, modularity, portability, usability, efficiency, lernability)
- quality control
- je potřeba kontrolovat dodržování stanovených standardů
  - quality reviews — QA team kontroluje vybraný proces či produkt
  - automated software assessment — automatická kontrola metrik

Stupně zralosti sw. týmů (CMM)

- process improvement = porozumění stávajicímu procesu a jeho změna k lepšímu
- W. E. Demming — po 2. světové válce pracoval na zlepšování japonského průmyslu — aplikoval statistickou kontrolu procesů v průmyslu .. a byl hodně úspěšný.
- proces má podobně jako produkt charakteristiky, např:
  - understandability
  - visibility
  - reliability
  - maintainability
- process improvement má následující klíčové fáze:
  - process analysis
  - improvement identification
  - proces change introduction
  - process change training
  - change tuning
- při snaze o zlepšování procesu je klíčové zjistit, co a jak měřit — GQM paradigm:
  - Goals
  - Questions
  - Metrics

Capability Maturity Model

- vyvinut by Software Engineering Institute (na Carnegie-Mellon University, financuje to americké ministerstvo obrany)
- původní záměr bylo mít prostředky k zjištění schopností potenciálního contractora, který bude něco dělat pro US DoD
- klasifikace sw. procesů na 5 úrovní:
  - **Initial level** — neexistují efektivní manažerské postupy a plány. Možná jsou zformalizovány postupy, ale není nijak kontrolováno jejich dodržování.
  - **Repeatable level** — existují formální manažerské, QA i configuration control procesy. Organizace je schopna úspěšně zopakovat projekty stejného typu jaké byly vykonány dříve. Výsledek projektu ale závisí spíš na konkrétních lidech, manažerech.
    - key process areas: sw. configuration management, sw. QA, sw. subcontract management, sw. project tracking and oversight, sw. project planning, requiremnts anagement
  - **Defined level** — organizace má proces dobře definovaný, takže má základ pro kvalitativní zlepšování. Existují formální procedury ke kontrole dodržování procesu.
    - key process areas: peer reviews, intergroup coordination, sw. product engineering, integrated sw. management, training programme, organization process definition, organizatio process focus
  - **Managed level** — je formálně definován nejen proces, ale i program sběru informací o probíhajících procesech. Data o procesech a produktech jsou sbíráná a na jejich základě probíhá zlepšování.
    - key process areas: sw. qualty management, quantitative process management
  - **Optimising level** — organizace provádí průběžný process improvement. Zlepšování procesů ma plán i budget a je implicitní součástí vnitřních procesů.
    - key process areas: process change management, technology change mgmt, defect prevention
- dá se dobře aplikovat na velkých organizacích (pro malé organizace může být už příliš byrokratický)
- tento model má ale i své nevýhody:
  - zaměřuje se výhradně na project management, ne na product development (nezohledňuje možnosti jako prototyping, formální metody ..)
  - vůbec se nezabývá analýzou rizik

  - není dobře popsáno v kterých typech organizací se dá CMM využít a kde už moc ne
  - organizace která splňuje 80% levelu 2 a 70% levelu 3 dostane pořád level 1 rating (= k postupu je potřeba splnit 100%)

- při srovnání s ISO 9000 se dá říct, že organizace na levelu 2 až 3 jej splňují, ale dá se najít i organizace na levelu 1 která ISu odpovídá. Přesto mnoho key areas s ISem koresponduje.

22.8 CASE systémy

- What is a CASE Environment? (na CMU)

22.9 Třívrstvá struktura informačních systémů, klinet/server

- wen: Client-server architecture

22.10 XML a značkovací jazyky

- zdroje:
  - wen: Markup language
  - Slajdy z přednášky o XML (podle mne by měl stačit obsah 1. přednášky)

22.11 Objektová analýza a návrh (UML)

22.12 Informační bezpečnost