KIV/ZSWI 2003/2004
Přednáška 6

Objektově-orientované metody vývoje SW

* mezi 1989 a 1994 bylo navrženo cca 40 nových objektových metodik, např. OMT (Rumbaugh et al. 1991), OOSE (Jacobson 1992), OOD (Booch 1994) atd.
* metody měly tolik společného že se tři klíčoví autoři Booch, Jacobson a Rumbaugh a rozhodli své návrhy integrovat do jedné metodiky (v rámci firmy Rational Software Corporation)
  - metodika nazvána ”Rational Unified Process”(R), dále jen RUP
  - v RUP se používá stejnými autory navržená notace UML
  - firma vydává úplný popis procesu ve formě HTML
  - tento popis je odkazován z mých stránek o ZSWI
  - dnes je to dnes jeden z nejznámějších SW procesů, ale používají se i další (OPEN Process, OOSP apod.)
  - je založen na dlouhodobých zkušenostech autorů
  - iterativní vývoj, prototypování

* další dostupný příklad - metodika GRAPPLE uvedená v knize [Schmuller: Myslíme v jazyku UML. Grada 2001] od kapitoly 15.

* většina metodik je velmi rozsáhlých, viz popis RUP
* popíšu pouze aktivity které najdete ve většině OO procesů, budu vycházet hlavně z RUP (a inspirovat se jeho bezprostředními předchůdci, tj. metodikami Booche, Jacobsona a Rumbaugha)
* nelze aplikovat mechanicky, pro vlastní použití je třeba upravit

Přehled kroků OO vývoje SW

* vývoj systému znamená vytváření modelů, model = abstrakce řešeného problému

* nejprve musíme porozumět řešenému problému (bez toho nemáme co modelovat)
* model obvykle nevytvoříme napoprvé správně, je zapotřebí více iterací
* postupně můžeme přidávat podrobnosti (atributy, metody, násobnost, průchodnost...)

* vývoj obvykle probíhá v následujících krocích:
  - sběr požadavků - zjistíme jaké požadavky na systém má uživatel; výsledkem model uživatelských požadavků, např. ve formě případů použití, rozhraní případů použití a doménových objektů (doména = skutečný svět ze kterého problém pochází)
  - analýza požadavků - strukturování systému z logického hlediska, předpokládáme ideální technologii; výsledkem analytický model = abstrakce popisující co má systém dělat (tj. zatím neříká jak to bude systém dělat)
  - návrh architektury systému - systém je rozdělen do podsystémů, provede se základní rozhodnutí týkající se komunikace mezi podsystémy, ukládání dat apod.; výstupem je architektura = popis prvků (objektů, komponent, rámců) ze kterých bude SW vytvořen a popis interakce mezi těmito prvky
  - návrh (design) systému - adaptace modelu vytvořeného v analýze pro realizaci ve skutečném světě; např. přidány třídy zapouzdřující datové struktury jako tabulky, seznamy, stromy apod. Implementace by měla být už přímočará a téměř mechanická.

Přechod mezi analýzou a návrhem systému poznáme podle toho, že se model začne týkat implementačního prostředí (začneme brát v úvahu vlastnosti cílového jazyka apod.

* při OO vývoji se pro sběr požadavků, analýzu i návrh používají stejné nástroje (v našem případě především UML diagramy), ale na různé úrovni abstrakce
  - pro jednotlivé profese SW týmu (analytici, vývojáři, management) jsou také určeny různé úrovně obecnosti modelů

Poznámka (objektové a strukturované metodiky)

Kromě objektových metodik existují i tzv. strukturované metodiky vývoje SW, které víceméně odpovídají strukturovanému programování (o nich budeme hovořit později, i když historicky OO metodikám předcházely). Výhodou strukturovaných metodik je, že počáteční modely jsou často srozumitelnější pro zákazníky.

[]

V této přednášce uvedu poměrně podrobně postup OO analýzy a návrhu, ale nebude zde uveden žádný rozsáhlejší příklad. Podrobné příklady (včetně naprogramování) můžete nalézt na WWW stránkách předmětu OOP doc. Herouta:

http://www.kiv.zcu.cz/~herout/vyuka/oop/zajimave.html

K dispozici je jednoduchá ukázková aplikace ilustrující základní vztahy mezi třídami (dědičnost, asociace) a rozsáhlejší ukázka aplikace ”Čerpací stanice”.

Sběr požadavků
....

Přehled:

* vstupem je definice problému
* provedeme doménovou analýzu a navrhneme základní doménové třídy
* provedeme sběr požadavků, nejčastěji se používají případy použití (use cases)

* na začátku by měla být stručná definice problému vytvořená zákazníkem případně vývojářským týmem s pomocí zákazníka
 - o definici problému nemůžeme předpokládat že je bez chyb, tj. v dalších krocích se bude měnit, rozšiřovat a zpřesňovat
 - pokud uděláte přesně to, o co si zákazník řekne, ale nenaplníte tím jeho skutečné potřeby, bude značně nespokojený
 - pokud definici problému vytváří zákazník, bude pravděpodobně smíšená s návrhem

Příklad (poněkud zjednodušený oproti realitě):

Vytvořte software pro síť bankomatů Naší Banky. Bankomaty budou komunikovat s centrálním počítačem banky, který transakce autorizuje a provede změny na účtu. Software centrálního počítače dodá banka. Systém vyžaduje uchovávání záznamů o činnosti a zabezpečení.

[]

* provedeme doménovou analýzu, cílem maximální porozumění doméně aplikace
 - např. seznámíme se s činností bankomatů, jejich zabezpečením apod.

* navrhneme základní doménové třídy, tj. třídy reprezentující objekty relevantní v aplikační doméně (např. organizační jednotky - katedry a fakulty, role - student, učitel, úředník atd.), je vhodné mít pro sběr požadavků
 - sledujeme podstatná jména v definici problému, věci a místa v aplikační doméně, pro každé vytvoříme předběžnou třídu (třída je zatím popsána pouze jménem); slovesa zaznamenáme aby časem mohla být operacemi
 - eliminujeme nepotřebné a chybné předběžné třídy
  . třídy které nejsou pro aplikaci relevantní zrušíme
  . pokud předběžná třída popisuje jednotlivý objekt (např. jméno, věk apod.) a nemusí existovat samostatně, prohlásíme jí za atribut
  . pokud předběžná třída popisuje činnost objektu, prohlásíme jí za operaci (např. telefonní spojení může být posloupnost akcí uvnitř telefonní sítě, aktéry jsou telefonní účastníci)
  . mezi předběžnými třídami by se neměly objevovat implementační konstrukce (např. seznam, pole, strom, tabulka apod.)

Zda bude předběžný objekt zachován závisí na typu aplikace, kterou vyvíjíme. Např. pokud vyrábíme telefony, bude ”telefonní spojení” součástí dynamického

modelu aplikace (viz výše). Pokud naopak provádíme účtování telefonních hovorů, bude totéž důležitá třída našeho modelu (s atributy jako je datum a čas, trvání spojení apod.).

V této počáteční fázi se zatím nesnažíme třídy strukturovat, protože bychom mohli podvědomě potlačit některé podstatné podrobnosti. Například pokud bychom vytvářeli systém pro katalogizaci knihovny, vznikají nám třídy pro různé typy objektů (knihy, časopisy, klasické desky, CD, ...). Strukturování do kategorií provedeme později vyhledáním podobností a rozdílů mezi základními třídami.

Příklad:

  Předběžné třídy vyplývající z definice problému: software, bankomat, centrální počítač, banka, transakce, účet, záznam o činnosti, zabezpečení.

  Předběžné třídy vyplývající z aplikační domény: klient, platební karta, stvrzenka, výplata.

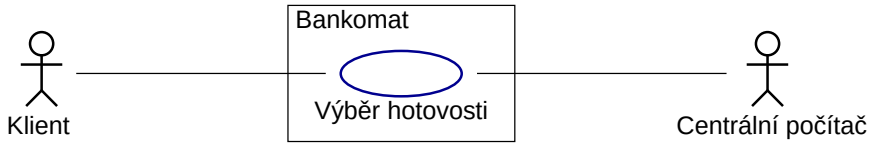
  Eliminujeme vágní třídy: software, zabezpečení.

[]

* provedeme sběr požadavků - jak budou potenciální uživatelé využívat systém, nejčastěji na základě případů použití
 - určení aktérů - tj. uživatelů a spolupracujících systémů
  . primární uživatelé, tj. ti kdo využívají hlavní fce systému
  . sekundární uživatelé, tj. ti kdo systém administrují a udržují
  . externí HW nebo SW systémy, se kterými bude navrhovaný systém komunikovat
 - identifikace případů použití (metodika viz konec 3. přednášky); souhrn případů použití by měl pokrývat všechny možné způsoby vyžití systému
 - stručný popis účelu případu použití (cca odstavec)
 - rozložení případu použití na kroky
  . základní posloupnost aktivit, tj. kroky aktéra a odpovědi systému
  . alternativní posloupnosti aktivit
 - případy použití a aktéry strukturujeme pomocí extend a include
  . hledáme posloupnosti kroků, společné pro více případů použití
 - pokud je případů použití velké množství, seskupíme je do balíčků tak, aby balíček představoval vysokoúrovňovou oblast funkčnosti systému

* případy použití, které lze snadno přehlédnout, protože se netýkají primárních funkcí systému:
 - start a ukončení systému
 - administrace systému, např. přidávání nových uživatelů, zálohování dat apod.
 - funkčnost potřebná pro modifikaci chování systému; např. z informačního systému potřebujeme vytvářet nové typy výstupů

Příklad:



 Případ použití: Výběr peněz z bankomatu.
 Aktéři: Klient, Centrální počítač
 Stručný popis: Zákazník vloží kartu a požádá o výběr určité částky. Bankomat mu po potvrzení centrálním počítačem požadovanou částku vydá.

 Popis jednotlivých kroků základního scénáře:

  1. Klient vloží kartu. Bankomat kartu přečte a zjistí její sériové číslo.
  2. Bankomat požádá uživatele o zadání PIN; uživatel zadá ”1234”.
  3. Bankomat ověří číslo karty a PIN u centrálního počítače.
  4. Bankomat požádá o zadání velikosti částky; uživatel zadá 1000 Kč
  5. Bankomat požádá centrální počítač o provedení transakce; centrální počítač transakci provede a vrátí nový zůstatek účtu.
  6. Bankomat vydá částku, vytiskne stvrzenku a vrátí kartu.

Alternativní scénáře:

A1. Uživatel vloží kartu. Bankomat kartu přečte a zjistí její sériové číslo.
A2. Bankomat požádá uživatele o zadání PIN; uživatel zadá ”9999”.
A3. Bankomat se pokusí ověřit číslo karty a PIN u centrálního počítače; centrální počítač je odmítne.
A4. Bankomat oznámí že PIN bylo chybné a vyzve uživatele, aby ho zadal znovu; uživatel zadá ”1234” což bankomat úspěšně ověří u centrálního počítače.
A5. Bankomat požádá o zadání velikosti částky; uživatel zadá 1000 Kč
A6. Bankomat požádá centrální počítač o provedení transakce; centrální počítač transakci odmítne pro nízký zůstatek na účtu.
A7. Bankomat vytiskne stvrzenku a vrátí kartu.

Jak vidíme, výsledkem je pouze soubor příkladových scénářů, který v žádném případě není úplnou specifikací systému.

[]

Analýza

Přehled kroků OO analýzy:

* vstupem je popis případů použití a doménové třídy
* případy použití zjemníme směrem k implementaci pomocí dynamických modelů, nejčastěji sekvenčních diagramů (synchronní chování) nebo pomocí stavových diagramů a diagramů spolupráce (asynchronní chování)
* vytvoříme diagram analytických tříd

* chování popsané v případech použití distribuujeme na objekty
 - rozebíráme jeden případ použití po druhém
 - popíšeme, který objekt je zodpovědný za které chování v případu použití
 - níže uvedu 2 příklady - pomocí CRC karet a pomocí sekvenčních diagramů

* v některých metodikách je oblíbená technika použití tzv. CRC karet (z angl. Class/Responsibilities/Collaborators)
 - pro třídy vytvoříme kartičky, nahoru napíšeme jméno třídy, vlevo zodpovědnost třídy, vpravo spolupracující třídy
 - při průchodu scénářem přidáváme nové odpovědnosti existujícím třídám; pokud neumíme, rozdělíme existující třídu na dvě nebo založíme novou
 - nevýhoda CRC karet - vazby mezi třídami nejsou znázorněny graficky
* pro projekt střední velikosti získáme 30 až 100 tříd

Příklad:

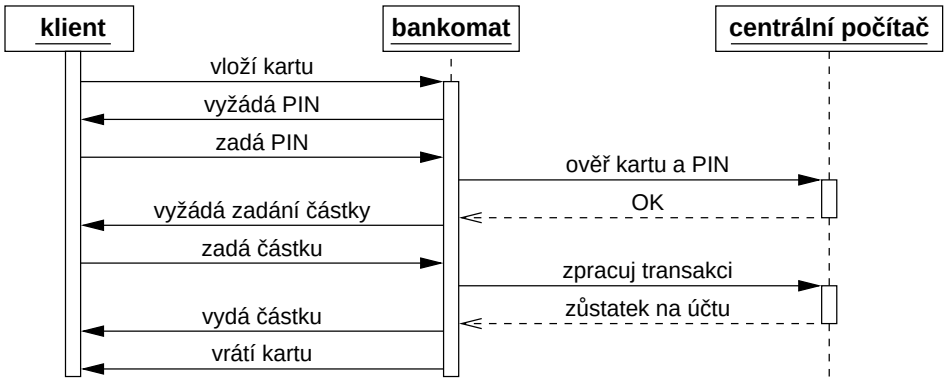
třída: Centrální počítač

odpovědnost:	spolupracuje s:
* ověří číslo karty a PIN	* bankomat
* provede transakci	
* vrátí nový zůstatek účtu	

[]

* pro podrobnější popis chování můžeme použít dynamické modely UML
 - nejprve rozebereme základní sekvenci z případu použití
 - alternativní sekvence většinou popisuje reakci na chyby, začleníme později
 - opravujeme seznam tříd

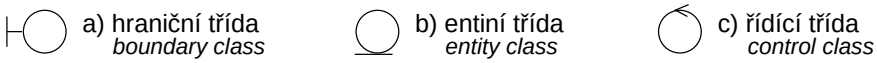
Příklad (sekvenční diagram pro případ použití ”Výběr peněz z bankomatu”)



[]

* konstrukce diagramu tříd:
 - na počátku je třída popsána pouze názvem
 - zkontrolujeme, zda není jiným názvem pro existující třídu (ponecháme ten název, který objekt popisuje nejlépe), zda není role (jméno třídy má popisovat její esenci, nikoli pouze roli kterou hraje v sekvenčním diagramu apod.)
 - ke třídě najdeme logické atributy = informace, která se nebude používat samostatně ale je silně svázána s objektem (např. jméno, věk, adresa budou atributy nějaké Osoby)

* třídy můžeme rozdělit podle následujících stereotypů:
 - všechno s čím aktéři přímo komunikují budou hraniční třídy; můžeme je zjistit např. z popisu případu použití
  . nejčastější hraniční třídy: formuláře, komunikační protokoly, rozhraní pro tiskárnu
  . v UML můžeme hraniční třídy označovat stereotypem <<boundary>> nebo níže uvedenou značkou (případně oběma způsoby zároveň)
 - informace kterou systém udržuje delší dobu skryjeme v entitních objektech; entitní objekty často odpovídají objektům reálného světa
  . typický příklad: Student, Fakulta, Předmět apod.
  . entitní objekty samy od sebe neiniciují komunikaci
  . v UML stereotyp <<entity>>
 - chování v systému koordinují řídící třídy
  . např. třídy obsahující obsahují řídící logiku, používající nebo nastavující obsah entitních tříd
  . obvykle se týkají realizace jediného případu použití
  . pokud je chování jednoduché, nemusejí být řídící třídy zapotřebí
  . v UML stereotyp <<control>>

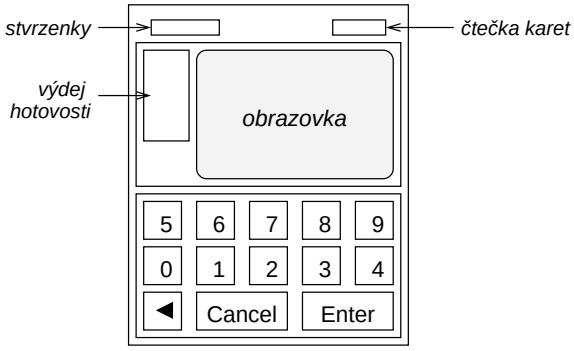


Příklad (opět bankomat):

 Entitními objekty budou v našem příkladu Klient a Účet.

 Výběr peněz musí být ověřen centrálním počítačem, který vystupuje jako aktér. Komunikace s tímto aktérem probíhá prostřednictvím tzv. ATM sítě, jejíž rozhraní (ATM Network Interface) bude hraničním objektem.

 Hraničními objekty uživatelského rozhraní budou klávesnice, obrazovka, čtečka platebních karet, výdejní automat bankomatu, tiskárna stvrzenek apod. Pro hraniční objekty představující uživatelské rozhraní bychom v této fázi měli vytvořit náčrtek nebo prototyp, který by si uživatel mohl vyzkoušet.



Pozor: zatím pouze získáváme požadavky na uživatelské rozhraní, neděláme návrh.

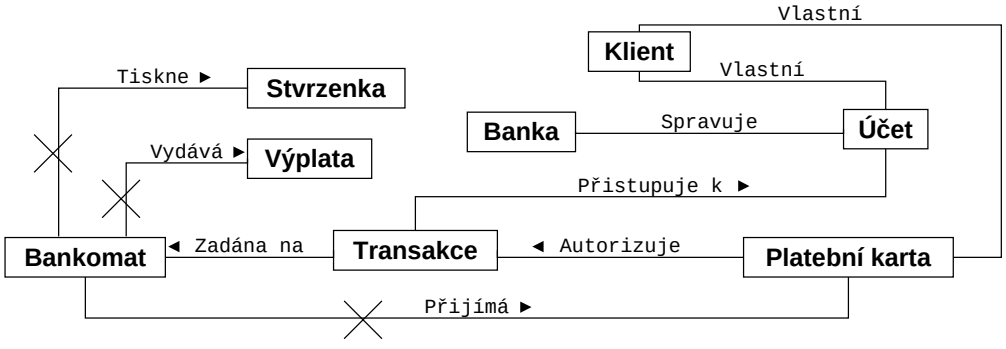
[]

* Constantine a Lockwood (1999) říkají, že prototyp uživatelského rozhraní má být abstraktní
 - příliš konkrétní (”hezky vypadající”) prototypy odvádějí pozornost od principiálních problémů
 - abstraktní modely dovolují najít rychleji dobré řešení, protože nás nezatěžují podrobnostmi

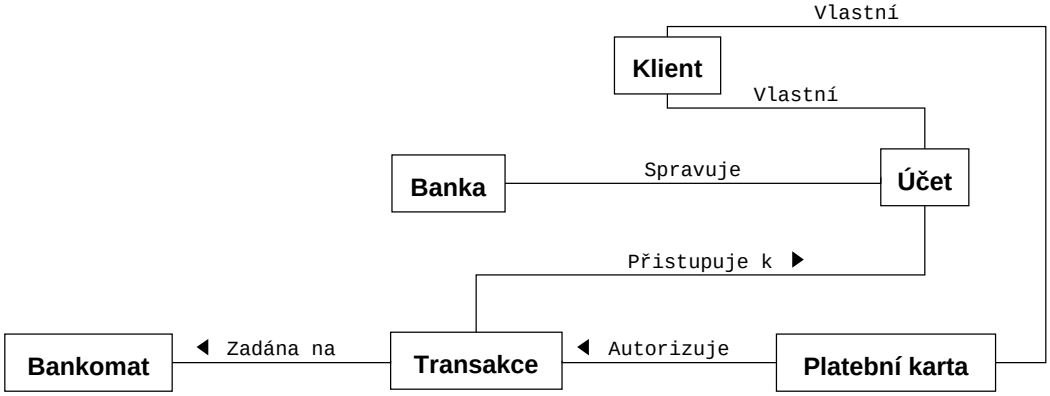
* vytváříme předběžné asociace mezi třídami
 - představují vztah mezi instancemi, který nějakou dobu přetrvává (např. Osoba pracuje pro Firmu); objekty o sobě navzájem ”vědí”
 - asociace mohou odpovídat fyzickému umístění nebo vztahu vlastnictví (má spojení s, je součástí, je obsažen v, patří), řízení a komunikaci (řídí, komunikuje s)
 - asociace měly by být pojmenovány účelem asociace
 - na počátku se nesnažíme rozlišit asociace a agregace, neuvádíme násobnosti ani průchodnost

Příklad (opět bankomat)

Předběžné asociace mezi vytvořenými třídami:

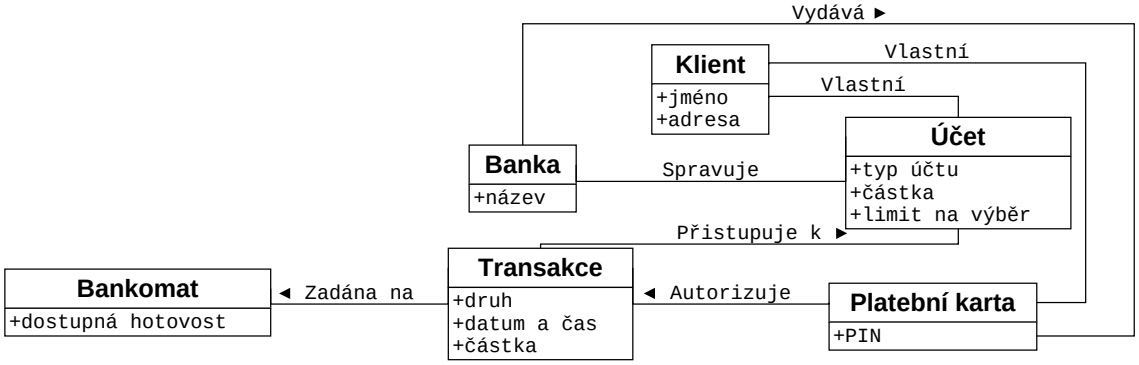


* zrušíme nepotřebné nebo nesprávné asociace
 - v této fázi zrušíme asociace, které nejsou podstatné pro řešený problém, jsou redundantní nebo které představují popis implementace
  . například asociace ”je prarodičem” lze popsat pomocí dvou asociací ”je rodič” (někdy ale můžou být i odvozené asociace užitečné!)
 - zrušíme předběžné asociace, představující jednorázové akce; např. bankomat sice přijímá platební karty, ale mezi bankomatem a platební kartou neexistuje trvalý (strukturální) vztah; tj. ve výše uvedeném příkladu zrušíme předběžné asociace (v obrázku jsou zrušené asociace škrtnuty)
 - vyhýbáme se asociacím mezi dvěma řídícími objekty a mezi hraničním a řídícím objektem, protože oba typy vztahů trvají krátkou dobu
  . vztahy které nepřetrvávají můžeme modelovat jako závislosti
 - vyhýbáme se ternárním a vyšším asociacím, většinou je lze přestrukturovat na binární asociace nebo případně popsat asociační třídou
  . například Firma vyplácí Plat Osobě lze přestrukturovat: Firma zaměstnává Osobu, Plat může být atributem asociace ”zaměstnává”

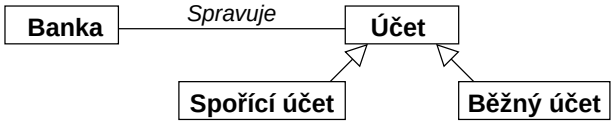


* vytváříme agregace a kompozice
 - agregace pokud objekt je součástí jiného nebo objekt je podřízený jinému, např. pokud se některé operace nad celkem provedou nad všemi částmi
 - kompozice pokud je silné vlastnění (strom součástí, součásti nelze sdílet) a stejná doba života

* hledáme primární atributy objektů a asociací
 - hledáme zatím pouze nejdůležitější logické atributy, relevantní pro aplikaci
 - jsou to navenek viditelné vlastnosti jednotlivých objektů, jako je jméno, rychlost, barva apod.



* vytváříme hierarchii dědičnosti, můžeme postupovat dvěma směry
 - zdola nahoru, tj. zobecněním - hledáme třídy se společnými vlastnostmi, které vyjmeme do nadtřídy (např. Běžný účet a Spořící účet)
 - shora dolů, tj. specializací - existující třídy zjemníme pomocí podtříd
 - vytvořená hierarchie by neměla být zbytečně hluboká (zvláště v případě, že cílový jazyk nebude OO)
 - pokud chceme využít polymorfismus, vedeme asociaci k rodičovské třídě



* testujeme dostupnost vůči dotazům - je pro třídu možné získat hodnotu nebo hodnoty, které potřebuje?
 - pokud chybí cesta, je pravděpodobně zapotřebí přidat asociaci

* výsledné diagramy specifikují strukturu systému, ale nikoli důvody, které nás k ní vedly; ty bychom měli také zdokumentovat

Cílem analýzy je dostatečně popsat problém a aplikační doménu bez závislosti na konkrétní implementaci (i když v praxi není možné tento cíl vždycky splnit).

Analýza není v žádném případě jednoduchá sekvence akcí s jasným koncem, ale

iterativní proces, ke kterému se musíme vracet po většinu doby vývoje systému. Často bývá citován výrok:

> ... analysis is frustrating, full of complex interpersonal relationships, indefinite, and difficult. In a word, it is fascinating.
>
> Once hooked, the old easy pleasures of system building are never again enough to satisfy you. [Tom DeMarco, 1978]

Návrh architektury systému

* architektura SW systému = vysokoúrovňový design SW
  - v angličtině se používají názvy system architecture, design, high-level design, top-level design, system design apod.
  - architektura slouží jako rámec pro podrobnější návrh rozsáhlého systému
  - popisuje organizaci systému do podsystémů, alokaci podsystémů na HW a SW komponenty
  - architektura systému se navrhuje buď po analýze (tj. před podrobným návrhem) nebo se její návrh překrývá s podrobným designem
    . pokud následuje po analýze, umožní rozdělit navrhovaný systém do podsystémů, jejichž návrh pak může probíhat nezávisle
  - jak zdůrazňuje mnoho autorů, dobře navržená architektura je podmínkou pro včasné odladění a pro udržovatelnost produktu

Návrh architektury SW systému obvykle probíhá v těchto krocích:

* rozdělení systému do podsystémů
* rozdělení do vrstev a oddílů (partitions)
* návrh topologie systému
* identifikace paralelismu, alokace na uzly a volba komunikace
* volba způsobu řízení atd.

Rozdělení systému do podsystémů

* rozdělení systému do podsystémů provádíme pro všechny větší aplikace
  - podsystém bude obsahovat aspekty systému s nějakými podobnými vlastnostmi (podobná funkčnost, stejné fyzické umístění apod.)
  - například kosmická loď bude obsahovat podsystémy pro podporu životních funkcí, pro navigaci, pro řízení motorů, pro řízení experimentů apod.
  - nebo operační systém bude obsahovat podsystémy pro plánování procesů, správu paměti, systém souborů apod.
  - podsystémů by nemělo být moc (i pro velké systémy cca do 20)

* podsystém můžeme nejsnáze identifikovat pomocí služeb které poskytuje
  - služba = množina fcí které mají stejný základní účel
  - např. souborový systém poskytuje množinu příbuzných služeb, jejichž základním účelem je poskytnout přístup k souborům
  - hranice podsystému bychom měli volit tak, aby většina komunikace probíhala uvnitř podsystému (mezi množinou objektů nebo modulů tvořících podsystém)

* vztah mezi dvěma podsystémy může být klient-poskytovatel (client-supplier) nebo peer-to-peer
  - vztah klient-poskytovatel - klient volá poskytovatele, který vykoná nějakou službu a pošle odpověď; poskytovatel nemusí znát rozhraní klienta
  - vztah peer-to-peer - každý podsystém může volat druhý, tj. oba musejí znát rozhraní toho druhého
    . vztah peer-to-peer je komplikovanější, mohou v něm vznikat obtížně srozumitelné komunikační cykly => snažíme se o vztah klient-poskytovatel kdekoli je to možné

* dekompozice systému do podsystémů - základní rozdělení do horizontálních vrstev nebo vertikálních oddílů

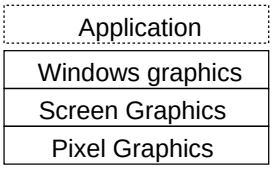
Rozdělení do vrstev
....

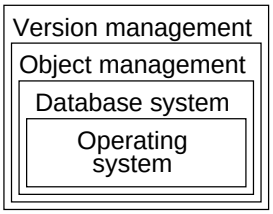
* vrstvené systémy = uspořádaná množina virtuálních světů
 - každý svět je vystavěn z prvků nižšího světa a poskytuje stavební prvky vyššímu světu
 - mezi vrstvami vztah klient-poskytovatel - nižší vrstvy (poskytovatelé) poskytují služby vyšším vrstvám (klientům)
 - znalost je jednosměrná, tj. podsystém zná jednu nebo více vrstev pod sebou, ale nezná vrstvy nad sebou

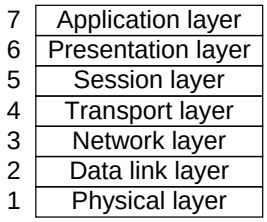
* objekty ve stejné vrstvě mohou být nezávislé, ale mezi objekty v různých vrstvách obvykle existuje korespondence (např. poskytují obdobné služby na různých úrovních abstrakce)

Příklad (interaktivní grafický systém)

 * v interaktivním grafickém systému
  - aplikace pracuje s okny
  - okna jsou implementována pomocí grafických operací typu ”nakresli čáru” nebo ”vybarvi obdélník”
  - grafické operace jsou implementovány pomocí operací nad jednotlivými pixely
 * každá vrstva může mít svou vlastní množinu tříd a operací, je implementována pomocí tříd a operací nižší vrstvy







[]

vrstvené architektury dvou typů, a to uzavřené a otevřené
 - uzavřené (striktně vrstvené) - vrstva je implementována pouze pomocí prostředků nejbližší nižší vrstvy
  . omezuje závislosti mezi vrstvami = princip skrývání informací
  . dovoluje snadnější změny rozhraní - změna ovlivní jen nejbližší vyšší vrstvu
  . například síťové modely, jako je sedmivrstvý ISO/OSI model, jsou uzavřené
 - otevřené - může používat prostředky kterékoli nižší vrstvy
  . omezuje potřebu definovat obdobné operace v každé vrstvě, kód může být kompaktnější a efektivnější
  . změna podsystému může ovlivnit kteroukoli vyšší vrstvu - obtížná údržba
  . například grafické systémy (změnu pixelu lze vyvolat z kterékoli vrstvy)
 - oba typy architektury jsou užitečné, při návrhu je nutné volit mezi modularitou a efektivitou

* specifikace problému obvykle definuje pouze svrchní vrstvu (= požadovaný systém)
* spodní vrstva je dána dostupnými zdroji (HW, OS, knihovny)
* pokud je velký rozdíl, je při návrhu zapotřebí přidat mezilehlé vrstvy pro přemostění případné konceptuální mezery
 - malé systémy cca 3 vrstvy, pro velké systémy obvykle postačuje 5-7 vrstev (i pro nejsložitější systémy je podezřelé více než 10 vrstev)

Poznámka (doporučení z RUP)

```
   pokud máme 0  - 10 tříd    pak vrstvení není zapotřebí
              10 - 50 tříd        2 vrstvy
             25 - 150 tříd        3 vrstvy
             100-1000 tříd        4 vrstvy
```

[]

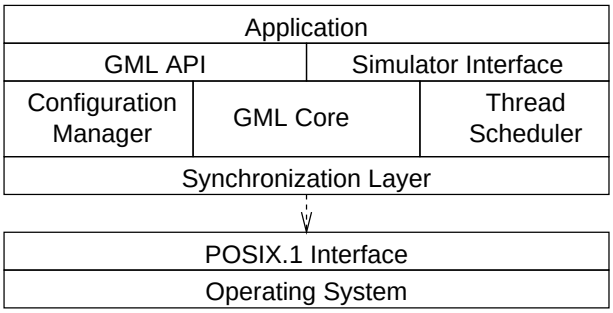
* není-li prostředí přenositelné, považuje se za vhodné vytvořit alespoň
  jednu vrstvu mezi aplikací a službami poskytovanými OS nebo HW (vrstva
  poskytuje logické služby a mapuje je na fyzické)
 - přepsáním této vrstvy můžeme systém přenést na jinou platformu

Rozdělení do oddílů
....

* oddíly (partitions) rozdělují systém vertikálně na nezávislé nebo slabě
  svázané podsystémy, každý z nich poskytuje jiný typ služeb

 - například operační systém obsahuje ovladače pro jednotlivé typy zařízení
 - podsystémy mohou o sobě navzájem něco vědět, ale protože tato znalost není
   velká, nevznikají podstatné závislosti mezi oddíly
 - např. v operačním systému podsystém pro plánování procesů a správu paměti,
   plánování procesů někdy potřebuje vědět kolik je dostupné paměti apod.

* systém může být postupně dekomponován do podsystémů pomocí vrstev a oddílů
  (vrstvy můžeme dělit na oddíly a naopak oddíly do vrstev)
 - ve většině velkých systémů směs vrstev a oddílů
 - příklad:



Topologie systému
....

* poté co byly identifikovány základní podsystémy, měly bychom určit toky dat
 - někdy mohou data téci mezi všemi podsystémy, v praxi jen zřídka
 - většinou jednodušší uspořádání (jednoduchá roura - např. překladač, hvězda
   např. hlavní systém řídí podřízené) apod.
 - konkrétní příklady ukážeme později (v kapitole o architektonických stylech)

Identifikace paralelismu
....

* dalším úkolem je identifikovat které podsystémy mají pracovat paralelně a
  u kterých se paralelní běh vylučuje
 - paralelní podsystémy mohou být implementovány různými HW jednotkami nebo
   různými procesy nebo vlákny OS (často závisí na zadání, např. bankomaty
   musí běžet navzájem nezávisle => samostatné HW jednotky)
 - podsystémy kde není možný paralelní běh mohou být např. součástí stejného
   procesu

* identifikace inherentního paralelismu
 - jako vodítko se používá dynamický model
 - dva objekty jsou inherentně paralelní pokud mohou přijímat události ve
   stejném čase bez vzájemné komunikace
 - inherentně paralelní objekty nemohou být součástí stejného vlákna řízení
 - například řízení křídel letadla a řízení motoru musí probíhat paralelně
   nebo případně nezávisle (lze implementovat nezávislým HW, cena vzájemné
   komunikace bude malá)

* definice paralelních úloh
 - vlákno řízení - množina objektů si navzájem předává řízení tak, že v jednom čase je aktivní pouze jeden objekt; řízení zůstává objektu dokud ten nepošle zprávu jinému objektu
 - různá vlákna řízení mohou běžet navzájem paralelně

Alokace podsystémů na počítače nebo procesory

* alokace podsystémů na počítače nebo procesory vyžaduje následující kroky:
 - odhad požadavků na HW zdroje
 - rozhodnutí zda budeme podsystém implementovat jako HW nebo jako SW
 - alokace na počítače nebo procesory (jednotky)
 - určení propojení fyzických jednotek

* provedeme odhad požadavků na HW zdroje
 - hrubý odhad potřebné výpočetní síly můžeme provést např. na základě požadovaného počtu transakcí za sekundu a doby zpracování jedné transakce (většinou odhad na základě experimentů)
 - pokud je zapotřebí větší výkonnost než může poskytnout jeden CPU, přidáme další procesory, případně implementujeme HW

* rozhodnutí zda podsystém bude implementován HW nebo SW
 - HW můžeme považovat za ”zatuhlou” optimalizovanou formu SW
 - k implementaci v HW vedou dva hlavní důvody:
  . existuje HW který poskytuje přesně požadovanou funkčnost
  . HW implementace poskytne vyšší výkonnost než SW implementace na obecném CPU (např. na signálovém procesoru poběží algoritmus rychleji)
 - nevýhoda - HW řešení není flexibilní

* alokace úloh na fyzické jednotky (počítače nebo procesory)
 - vyžaduje-li úloha větší výkon, poskytneme jí více CPU
 - určité úlohy vyžadují konkrétní fyzické umístění, např. každý bankomat má vlastní SW, aby mohl (omezeně) pracovat i když je síť nefunkční
 - podsystémy které nejvíce komunikují by měly být přiřazeny stejné jednotce

* po určení druhu a relativního počtu fyzických jednotek určujeme propojení mezi jednotkami
 - vybereme topologii (propojení mohou odpovídat asociacím v objektovém modelu, případně (pro strukturované metody návrhu) toku dat v DFD)
 - určíme obecné požadavky na mechanismy a komunikační protokoly (například průchodnost nebo spolehlivost)

Datová úložiště

* interní a externí úložiště dat mají dobře definované rozhraní, proto mohou sloužit jako čistá hranice oddělující podsystémy
 - například účetní systém může používat relační databázi pro komunikaci mezi podsystémy
 - nebo aplikace pro zpracování obrazu může používat soubor - matici pixelů

* soubory
 - jsou levné, jednoduché a permanentní, ale mají nízkou úroveň abstrakce, tj. je v aplikaci je nutný další kód pro práci s nimi
 - soubory jsou vhodné pro data která jsou objemná ale zároveň obtížně strukturovatelná v termínech DB systémů
 - také pro data která mají malou informační hustotu nebo data která jsou uchovávána krátkou dobu

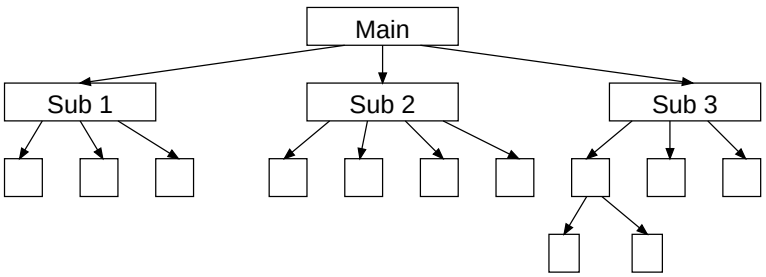
* databáze - silnější prostředek
 - společné rozhraní pro množinu aplikací pomocí standardního přístupového jazyka (SQL)
 - výhodné pro data ke kterým bude přistupovat více uživatelů, případně více programů, a pro data která mohou být efektivně spravována příkazy DB jazyka
 - poskytují další vlastnosti jako podporu transakcí, zotavení po havárii apod.

- nevýhody:
  . vyšší režie
  . nedostatečná podpora pro složitější datové struktury (relační databáze předpokládají velké množství dat s relativně jednoduchou strukturou)
  . často není možná čistá integrace s programovacím jazykem (SQL je neprocedurální, zatímco aplikaci vytváříme v procedurálním nebo OO jazyce)

Výběr mechanismu řízení
.....

* tři způsoby řízení v souvislosti s externími událostmi, a to sekvenční systém řízený procedurálně nebo řízený událostmi, a paralelní systém
 - doplňuje strukturální modely architektury

* systémy řízené procedurálně
 - běh je řízen programovým kódem
 - procedura žádá o vstup např. z klávesnice a pozastaví se (blokuje se)
 - po příchodu běh pokračuje v proceduře, která o vstup žádala
 - například téměř všechny znakově orientované aplikace v MS DOSu a v Linuxu
 - výhoda - jednoduchá implementace
 - nevýhoda - obtížné zpracování asynchronních událostí (program musí požádat o vstup)

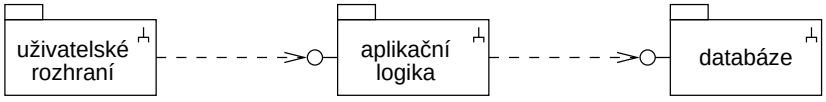


* systémy řízené událostmi
 - běh systému řídí dispečer poskytovaný podsystémem, programovacím jazykem nebo OS
 - s jednotlivými událostmi jsou svázány procedury aplikace
 - systém někdy poskytuje frontu událostí, nově příchozí události se řadí do fronty ze které dispečer vybere následující událost a zavolá odpovídající proceduru (”callback”)
 - procedura po skončení obsluhy události vrací řízení dispečeru
 - ve skutečnosti simuluje spolupracující vlákna uvnitř úlohy, ale na rozdíl od skutečného paralelismu dlouho trvající procedura zablokuje celou aplikaci
 - například téměř všechny GUI (MS Windows, X Window), simulace apod.
 - jednoduchá obsluha nových typů událostí
 - obtíže implementace - procedury se vracejí, proto nemůžeme stav systému uchovávat v lokálních proměnných (musíme použít globální proměnné)

* paralelní systémy
 - řízení několik nezávisle běžících objektů
 - události přicházejí objektům jako zprávy
 - objekt může čekat na vstup, zatímco ostatní pokračují v činnosti

Příklad architektury (třívrstvá architektura)

Mnoho současných aplikací je strukturováno tak, aby od sebe byly odděleny databáze, vlastní aplikace a uživatelské rozhraní:



[]

V praxi je důležité, aby pro každý projekt existovala osoba, která je zodpovědná za architekturu systému (šéf-architekt). Většinou to bývá vedoucí týmu.

❇