OBSAH PŘEDNÁŠEK

KIV/ZSWI 2004/2005
Přednáška 1

Poznámky k textu přednášek
==========

Pokud bude v zápisu přednášek něco nesrozumitelné nebo pokud najdete překlepy apod., prosím dejte mi to vědět nejlépe e-mailem na adresu <luki@kiv.zcu.cz>.

Věci, které jsou uvedeny s poznámkou typu ”pro zajímavost”, nebudou vyžadovány u zkoušky, ale bylo by dobré si je alespoň přečíst.

Nejčastější zkratky, použité v tomto textu:

* DSP = dokument specifikace požadavků
* HW = hardware
* SW = software
* SWE (SW Engineering) = softwarové inženýrství

Jak a proč vzniklo SW inženýrství
==========

* první počítače programovány jednotlivci nebo malými týmy
* často VT výpočty, jazyky FORTRAN a assembler

* příchod počítačů III. generace (1965-1980) - integrované obvody (oproti počítačům sestaveným z jednotlivých tranzistorů apod.)
* o několik řádů výkonnější, větší paměť apod.
* možnosti nových aplikací (banky, pojišťovny, letecké společnosti...)
* výsledné aplikace o několik řádů rozsáhlejší než předchozí SW systémy

* výsledek - důležité systémy často léta zpoždění, cena několikanásobek původních předpokladů, chybovost atd. - termín ”softwarová krize”
* ukázalo se, že způsoby vývoje pro malé SW projekty se nedají použít pro velké projekty, zapotřebí nové techniky a metody
* termín ”softwarové inženýrství” navržen v 1968 na konferenci NATO o ”softwarové krizi”

* nejznámější případ systému OS/360 od IBM
 - jeden z prvních velmi rozsáhlých projektů operačního systému
 - na začátku 200, maximum přes 1000 lidí
 - více o něm na ZOS

* Brooks na základě zkušeností napsal knihu ”The Mythical Man Month” (Brooks 1975)
* téma - proč je obtížné vytvářet velké SW systémy

* Brooks např. tvrdí, že programátoři jsou schopni napsat jenom cca 1000 řádků odladěného kódu za rok (tj. v průměru cca 5.5 řádku za den)
* Brooks vs. odpovědi na DOTAZ - jak je to možné?

* velké projekty (= stovky programátorů) jsou úplně jiné než malé projekty - zkušenosti z malých projektů se nedají na velké přenést

* velké projekty - dlouhou dobu spotřebuje:
 - plánování jak rozdělit projekt do modulů
 - specifikace činnosti a rozhraní modulů
 - získání představy o interakci modulů
 - to všechno před tím, než začne vlastní psaní programu (”kódování”)

* následuje kódování a odladění jednotlivých modulů
* integrace (sestavení) modulů do výsledného systému
* výsledný systém je třeba otestovat (i když jednotlivé moduly odladěny, po sestavení částí obvykle nefunguje napoprvé)

* tato posloupnost se nazývá ”vodopádový model” vývoje SW:
1. Plánování
2. Kódování

```
3. Test modulů
4. Test systému
5. Nasazení
(Ještě se k tomu vrátíme; teď zmiňuji pouze pro informaci.)

* Brooks odhadl, že
 - 1/3 celkové práce je plánování
 - 1/6 kódování
 - 1/4 testování modulů
 - 1/4 testování systému

* jinými slovy
 - kódování je ta nejsnazší část
 - obtížné je:
  . rozdělit projekt do modulů
  . zajistit, aby modul A správně komunikoval s modulem B

* tj. pokud 1 programátor píše malý program, zůstává mu ta nejjednodušší část
  (jeho efektivita je také vyšší, podle současných měření cca 20 řádků/den)
* většina úloh, které jste zatím řešili, byly malé úlohy
* cílem tohoto předmětu je získat základní představu i o ostatních
  částech procesu vývoje SW

Poznámka (co se míní pojmy software, SW systém a SW produkt)

 * většina lidí si pod pojmem SW představuje pouze programy, pro praxi
   příliš omezující pohled
 * SW = programy + dokumentace + konfigurační data
 * SW systém sestává obvykle z několika programů, konfiguračních
   souborů, systémové dokumentace (popisuje strukturu systému) a
   uživatelské dokumentace (vysvětluje, jak systém používat)
 * SW produkt = SW, který se dá prodat zákazníkovi

 * existují 2 základní typy SW produktů:
  - generické produkty
   . vyvíjen např. na základě analýzy potřeb trhu
   . samostatné systémy prodávané na otevřeném trhu každému, kdo je
     schopen si je koupit (shrink-wrapped SW)
   . např. operační systémy, textové procesory, kreslící programy,
     databáze, překladače programovacích jazyků atd.
  - produkty vyvíjené na zakázku (customised products)
   . SW vyvíjený pro konkrétního zákazníka na základě jeho požadavků
   . např. informační systém pro konkrétní firmu, řídící systémy pro
     elektronická zařízení apod.
  - nejpodstatnější rozdíl - kdo určuje specifikaci

[]

Co je SW inženýrství?
.....................

* pojem software - viz poznámka výše

* pojem inženýrství (engineering) - slovníková definice je: praktická
  aplikace teorie, metod a nástrojů při návrhu strojů, mostů apod.
 - praktické řešení je ale třeba najít, i když odpovídající teorie (ještě)
   neexistuje
 - problémy je třeba řešit v rámci daných finančních a organizačních omezení

* SW inženýrství je aplikace inženýrských metod na software
 - zabývá všemi aspekty tvorby SW: specifikací, vývojem, testováním, údržbou,
   managementem, atd. především rozsáhlých SW systémů, vývojem teorie, metod
   a nástrojů pro vývoj SW
```

Definice z IEEE Standard Computer Dictionary (1990):

Aplikace systematického, disciplinovaného, měřitelného přístupu na vývoj a údržbu software; jinými slovy, aplikace inženýrských principů na software.

* rozdíly mezi SW inženýrstvím (SWE) a informatikou (computer science, CS):
 - CS se zabývá algoritmy, způsobem práce atd. počítačů a SW systémů - existuje exaktní popis
 - SWE řeší praktické problémy tvorby SW
  . příliš složité, často nutné používat ad hoc metody
  . na rozdíl od CS se v SWE většinou nedozvíte porovnání jednotlivých metod apod. - porovnání je obtížné a drahé (dokonce čím dál dražší)
  . často poskytuje obecné koncepce, je na uživateli, aby je naplnil jednotlivinami

Ale zpátky k Brooksově knížce:

Proč Mythical Man Month
.....

* dodneška můžeme najít různá vyjádření náročnosti vývoje SW v ”člověkoměsících” (resp. ”člověkorocích”) = počet lidí*čas
* titul Brooksovy knihy vychází z tvrzení, že čas a počet lidí nejsou zaměnitelné

* pokud projekt trvá 15 lidem 2 roky, není možné, aby 15*24=360 lidem trval měsíc (a asi ani to, aby 60 lidem trval 6 měsíců)

* je to ze 3 důvodů:
1) práce není plně paralelizovatelná
  - dokud není dokončeno plánování a dokud není určeno rozdělení systému do modulů a definováno rozhraní modulů, nemůžeme začít s kódováním
  - např. pro dvouletý projekt může plánování trvat 8 měsíců
2) abychom plně využili velký počet programátorů, musíme systém rozdělit na velký počet částí (aby každý programátor měl práci)
  - každý podsystém ale může potenciálně komunikovat se všemi ostatními => počet uvažovaných interakcí mezi podsystémy roste s druhou mocninou počtu podsystémů
3) ladění a testování systému jsou obtížně paralelizovatelné
  - 10 lidí nenajde chybu 10x rychleji než 1
  - skutečně spotřebovaný čas závisí na počtu chyb a obtížnosti jejich hledání

Brooks shrnul svou zkušenost do tzv. Brooksova zákona:

”Přidáním lidí ke zpožděnému projektu projekt ještě více zpozdíme.”
(Adding manpower to a late software project makes it later.)

* proč?
 - přidaní lidé se musejí s projektem seznámit, což zabere čas i stávajícím lidem (místo aby programovali, učí nové členy týmu)
 - práce musí být přerozdělena tak, aby to odpovídalo většímu počtu členů týmu (přerozdělení práce zabere čas, také tím přijdeme o část již udělané práce)

Struktura týmu
.....

* malé skupiny programátorů (do 10 lidí) jsou obvykle organizovány celkem neformálně
 - vedoucí skupiny se spoluúčastní vývoje
 - ve skupině se může najít ”technický vedoucí” skupiny, který určuje technické směřování
 - práce je diskutována skupinou jako celkem (demokratické konsensuální rozhodování), práce je rozdělována podle schopností a zkušeností

* velké projekty - velké týmy (OS/360 - ve špičce přes 1000 lidí)
* jak tým strukturovat?

* hodně důležitá kvalita lidí

* je známo (Sackman et al 1968), že špičkoví programátoři jsou 10x
  výkonnější než špatní programátoři (ve skutečnosti měření probíhalo ve
  skupině zkušených programátorů; Sommerville uvádí rozdíl až 25x)
* problém - pokud potřebuji 200 programátorů, těžko zaměstnám 200
  špičkových - jsem nucen zaměstnat lidi různých kvalit

* ve velkých projektech je důležitá tzv. ”architekturální koherence”
* nejlépe aby jeden člověk měl pod kontrolou návrh (design) systému
* jinak může vzniknou slepenice, která se nikomu nebude líbit

* kombinace myšlenky ”někteří programátoři jsou mnohem lepší než jiní”
  a ”potřeba architekturální koherence” (Mills asi 1970)
 - organizace týmu typu ”chirurgický tým” (surgical team)
 - výkonný programátor = šéf týmu, měl by mít možnost 100% pracovat na
   návrhu a kódování
 - ostatní členové týmu mu pomáhají, aby se nemusel zabývat rutinními věcmi
 - pro 10 členný tým např. tyto role:
  . šéfprogramátor - navrhuje architekturu a píše kód
  . kopilot - pomáhá šéfprogramátorovi a slouží coby hlásná trouba
  . administrátor - management lidí, peněz, prostoru, zařízení atd.
  . redaktor - rediguje dokumentaci, kterou musí psát šéfprogramátor
  . sekretářky - administrátor i redaktor je potřebují
  . archivář (program clerk) - archivuje kód i dokumentaci
  . toolsmith - vytváří jakékoli nástroje, které šéfprogramátor
    potřebuje
  . tester - testuje kód šéfprogramátora
  . jazykový právník (language lawyer) - externista, který může
    šéfprogramátorovi poradit s programovacím jazykem (moderní programovací
    jazyky jsou jednodušší než PL/1 - v současnosti asi nebude zapotřebí
    jazykový právník, ale znalec knihoven by se mohl uplatnit)
 - dnes už je mnoho uvedených fcí automatizovatelných - je možný menší tým,
   ale základní myšlenka stále platí

* co když máme větší projekt?
 - musíme organizovat jako hierarchii
 - na nejnižší úrovni mnoho malých týmů, každý veden šéfprogramátorem
 - skupiny týmů musejí být koordinovány managerem
 - ze zkušenosti vyplývá, že každý člověk, kterého řídíte, vás stojí 10%
   času => manager na plný úvazek pro každou skupinu 10 týmů
 - tito manažeři musejí být řízeni... až 3 úrovně

Poznámka (rizika chirurgického týmu pro školní úlohy):

 Ustanovení efektivního chirurgického týmu vyžaduje talentovaného
 programátora (kterých je málo) a čas (který nemáte) a i poté má určitá
 rizika (týkají se zejména rolí ostatních členů týmu). Proto chirurgický tým
 nedoporučuji např. pro řešení zápočtových úloh.

[]

* Brooks popisuje pozorování, že výše uvedeným stromem neprocházejí dobře
  špatné zprávy (tzv. ”bad news diode”)
 - žádný šéfprogramátor ani manager nebude chtít říci nadřízenému, že má
   zpoždění 4 měsíce a deadline není možné stihnout (nositelé špatných zpráv
   obvykle nejsou vítáni)
 - výsledek - vrcholový management se obtížně dozvídá o skutečném stavu
   projektu

Role zkušenosti
....

* pro projekt je důležité mít zkušené návrháře
* Brooks ukazuje, že většina chyb není v kódu, ale v návrhu
 - programátoři správně naprogramují to, co se jim řeklo, ale co se jim řeklo
   (někdy ani neřeklo), je chybně
* proto navrhuje místo klasického vodopádového modelu:
 - napsat hlavní program, který bude volat top-level procedury (které
   jsou na začátku prázdné)
 - postupně přidávat moduly do celého systému

```
 - výsledek - testování integrace systému se děje kontinuálně, chyby v
   návrhu se projeví dříve

* nějaká (malá) zkušenost týmu je nebezpečná - viz tzv. efekt druhého systému
  (second systems effect)
 - první produkt týmu obvykle bývá minimální, protože návrháři mají
   obavy, aby produkt vůbec vytvořili
 - produkt pracuje, na členy týmu to udělá dojem
 - při vytváření druhého systému zahrnou vše, co při vytváření prvního
   produktu vynechali
 => výsledek - druhý systém je nafouklý, nevýkonný atd.
 - při vytváření třetího systému už si opět dávají pozor


Příklad (v OS):

* první systém se sdílením času CTSS - minimální funkčnost
* následník MULTICS - příliš ambiciózní, verze kterou se po letech podařilo
  dokončit se příliš neujala
* třetí systém UNIX - pečlivý výběr vlastností, úspěšnější.

"No Silver Bullet"
..................

* Brooks napsal také důležitý článek nazvaný "No Silver Bullet" (1986)
  (legendy tvrdí, že vlkodlaka lze zastřelit pouze stříbrnou kulkou)
* tvrdí, že žádná technologie nebo technika managementu nezpůsobí do 10
  let řádové zlepšení produktivity vývoje SW - a měl pravdu

* různé věci, které byly považovány za "silver bullet":
 - lepší vysokoúrovňové jazyky
 - objektově orientované programování
 - umělá inteligence a expertní systémy
 - grafické programování
 - verifikace programů
 - prostředí pro tvorbu programů
* spíše se zdá, že budou postupná vylepšení

(Konec povídání o Brooksově knížce.)


Softwarový proces
=================

* SW proces = množina aktivit a (mezi)výsledků (artefaktů), jejichž výsledkem
  je SW produkt

* existují různé SW procesy
 - velká různorodost vytvářeného SW => pro různé typy systémů vhodné
   rozdílné procesy
 - do úvahy je třeba vzít různé schopnosti (znalosti, dovednosti)
   zúčastněných lidí
  . například pro Boehmův spirálový model potřebujeme lidi, kteří mají
    dostatek zkušeností pro provádění analýzy rizik

* nicméně ve všech SW procesech se objevují 4 základní aktivity:
 - specifikace SW = určení požadované funkčnosti a definice omezení
 - vývoj SW = tvorba SW splňujícího specifikaci
 - validace SW = ověření, že SW dělá, co požaduje zákazník
 - evoluce SW = přizpůsobení SW měnícím se požadavkům zákazníka

* různé SW procesy organizují tyto 4 aktivity různým způsobem (různé
  časování apod.)

Modely SW procesu
.................

* model neboli paradigma SW procesu = zobecněný popis procesu, resp. popis
  procesu z určitého pohledu
```

* nyní uvedeme nejpoužívanější obecné modely SW procesu

* vodopádový model (waterfall model)
 - výše uvedené 4 aktivity chápe jako samostatné fáze procesu
 - po splnění se fáze ukončí a přechází se na další

* evoluční vývoj
 - aktivity specifikace, vývoje a validace jsou smíšeny
 - ze specifikace je velmi rychle vyvinut prvotní systém
 - na základě požadavků zákazníka je systém dále upravován
 - pak může být systém předán nebo reimplementován strukturovanějším způsobem (pro snazší údržbu)

* formální transformace
 - na počátku je vytvořena abstraktní formální (matematická) specifikace systému
 - tato specifikace je postupně transformována do funkčního programu
 - transformace zachovávají správnost, tj. na konci víme, že program odpovídá specifikaci

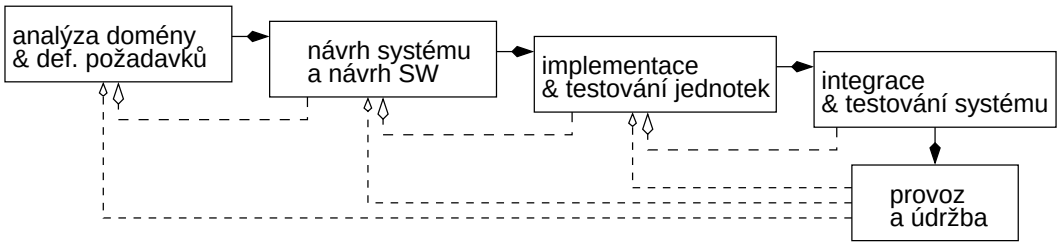
* sestavení systému ze znovupoužitelných (reusable) komponent
 - předpokládá, že části systému již existují
 - vývoj systému se zaměřuje na integraci těchto částí (místo toho, aby se vyvíjely znovu)

* iterativní modely - inkrementální vývoj a spirálové modely
 - hybridní modely, umožňují použít různé přístupy pro různé části systému

Neexistuje nějaký ”nejsprávnější” model, různé modely jsou vhodné pro různé situace. V praxi se nejčastěji používají SW procesy založené na vodopádovém modelu a evolučním vývoji. Formální metody byly v několika projektech úspěšně použity, zatím ale nejsou příliš rozšířené - rozšíření lze čekat spíše v budoucnu pro vývoj distribuovaných systémů. Většina SW procesů zatím není explicitně orientována na vyváření systémů z existujících komponent; i v tom se dá očekávat změna. Zajímavé jsou hybridní modely, protože umožňují realizovat podsystémy podle nejvhodnějšího modelu (odpovídají potřebě tvorby velmi rozsáhlých systémů).

Vodopádový model vývoje SW
.....

* nejstarší publikovaný model (Royce 1970)



Základní aktivity jsou:

* analýza domény, získávání (sběr) a definice požadavků
 - analýza domény = seznámení s širším (např. obchodním) kontextem systému
  . pojem ”doména” zde znamená obor, kterého se problém týká
 - získávání požadavků = konzultací s uživateli systému se zjistí cíle, požadované služby a omezení kladená na systém
 - definice požadavků = výše uvedené cíle, služby atd. jsou podrobně definovány v dokumentu specifikace požadavků (DSP), který slouží jako specifikace vytvářeného systému

* návrh systému a návrh SW (angl. system & software design)
 - návrh systému rozdělí celkové požadavky na požadavky na HW a požadavky na SW, určí celkovou architekturu systému
 - návrh SW zahrnuje identifikaci a popis základních abstrakcí a jejich vztahů (často pomocí grafické notace, např. UML)

* implementace a testování modulů
 - design je realizován jako množina modulů, tříd, případně programů
 - testování jednotek = ověření, že každý modul odpovídá specifikaci modulu

* integrace a testování systému
 - jednotlivé moduly jsou sestaveny do výsledného systému
 - úplný systém je otestován na shodu se specifikací
 - po otestování je systém předán zákazníkovi

* provoz a údržba
 - nejdelší fáze životního cyklu - systém se prakticky používá
 - údržba = oprava chyb programu a designu, které nebyly odhaleny v předchozích fázích + rozšiřování systému podle nových požadavků + zlepšování implementace

Poznámka (pojem údržba)

Pojmem ”údržba” (angl. maintenance) se v SW inženýrství neníní administrace systému, ale změny software vynucené provozem systému.

[]

Teoreticky by další fáze procesu neměla začít, dokud není předchozí fáze dokončena; v praxi:

* během designu jsou odhaleny problémy s požadavky, během kódování problémy s designem apod.
* proto SW proces není lineární sekvence - ve skutečnosti se musíme vracet (na obrázku naznačeno čerchovaně)
 - většinou se vracíme o 1 fázi
 - údržba způsobuje změny v určité fázi, dále se pokračuje podle vodopádového modelu

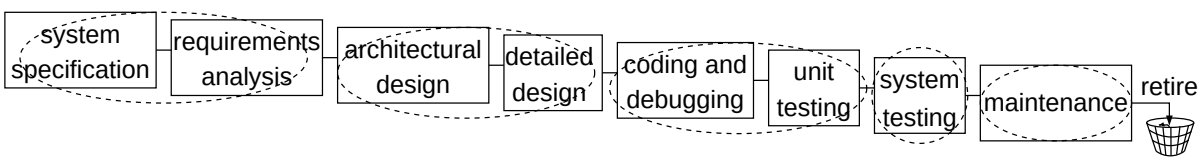
* výše zmíněné iterace jsou drahé a pracné (zahrnují přepracování a schvalování dokumentů)
* v praxi proto často dojde po několika iteracích ke zmražení příslušné části vývoje (např. zmražení specifikace) a přejde se na další fázi
 - problémy jsou tím ”odloženy na později”, čili fakticky ignorovány
 - zmražení požadavků často vede k tomu, že systém nedělá, co uživatel chce
 - zmražení designu obvykle vede ke špatně strukturovaným systémům (problémy designu se pak při implementaci obcházejí různými triky, které ztíží další vývoj - ”informační skleróza”)

Problémy vodopádového modelu:

* rozdělení projektu do fází je málo flexibilní
 - je obtížné reagovat na změny požadavků ze strany zákazníka
 - proto je vodopádový model vhodný pouze, pokud je problém dobře známý (= jsme schopni získat korektní specifikaci)
 - problémy s vodopádovým modelem vedly k formulaci alternativních modelů

Poznámka (odlišnosti v literatuře)

Protože SW inženýrství je relativně mladý obor, najdeme v různé literatuře popisující totéž téma značné odlišnosti (vyjma oblastí, které jsou již standardizovány). Např. porovnejte první zmínku o vodopádovém modelu uvedenou v souvislosti s Brooksovo knihou s výše uvedeným a s následujícím obrázkem:



Rozdělení do jednotlivých fází můžeme nazvat různě, principy ovšem zůstávají tytéž.

[]

Evoluční modely vývoje SW
.....

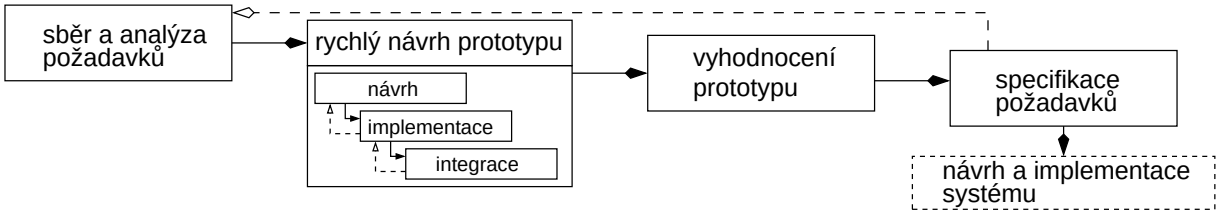
* základní myšlenka:
 - vytvořit počáteční implementaci
 - vystavit jí komentářům uživatele
 - postupně vylepšovat přes meziverze dokud nedostaneme adekvátní výsledek

* fáze specifikace, vývoje a validace neprobíhají vždy sekvenčně, ale mohou probíhat i paralelně, mezi fázemi je zpětná vazba

* existují 2 základní typy modelů evolučního vývoje SW:

1. model výzkumník (exploratory development, tj. průzkumný vývoj)
  - cílem je spolupracovat se zákazníkem na zjištění jeho požadavků, abychom mu nakonec dodali požadovaný systém
  - vývoj obvykle začíná dobře srozumitelnými částmi systému
  - systém se vyvíjí přidáváním nových vlastností navrhovaných zákazníkem

2. model prototyp (throw-away prototyping, tj. doslova tvorba prototypu, který bude zahozen)
  - cílem je lepší pochopení zákazníkových požadavků a v důsledku vytvoření lepší definice požadavků na systém
  - tvorba prototypu se zaměřuje na ty části požadavků zákazníka, kterým moc nerozumíme



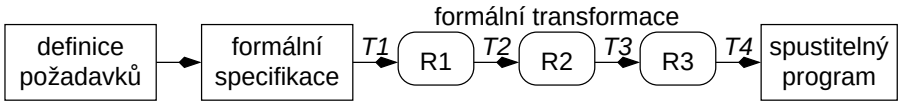
* výhody evolučního vývoje SW
 - často vede efektivněji než vodopádový model k systému splňujícímu bezprostřední požadavky zákazníka
 - specifikace může být vytvářena postupně
 - zákazník vidí, že se něco děje (na rozdíl od vodopádu, kde produkt vidí až na konci)

* problémy:
 - proces není viditelný: manažeři nemohou měřit postup vývoje (na rozdíl např. od vodopádového modelu)
 - systémy vyvářené evolučně jsou často špatně strukturované
  . neustálé změny vedou k porušení struktury systému
  . vývoj dalších verzí se postupně stává obtížnější = dražší
 - prototypování může vyžadovat speciální nástroje a techniky
  . výstup může být problematický, např. nekompatibilní s jinými nástroji a technikami

* tj. vhodné buď pro malé systémy (do 100 000 řádků kódu), nebo pro středně velké systémy (do 500 000 řádků) s krátkou dobou života
* ve velkých systémech by evoluční vývoj působil zřetelné problémy, ale je možné použít smíšený přístup:
 - špatně srozumitelné části - throw-away prototyping => zpřesnění DSP
 - dobře srozumitelné části - vodopádový model
 - uživatelské rozhraní - model výzkumník

Formální modely vývoje SW

* určitým způsobem podobné vodopádovému modelu
* na počátku je vytvořena specifikace požadavků
* ta je zpřesněna do podrobné formální specifikace systému (pojmem formální se zde míní ”matematická”)
* postupné zpřesňování specifikace formálními transformacemi
* konečným výsledkem má být spustitelný program



* v každém kroku se přidávají podrobnosti, ale můžeme dokázat, že odpovídá reprezentaci systému z předchozího kroku
* pokud neuděláme chybu, pak výsledek (dokazatelně) odpovídá specifikaci
* proces designu, implementace a testování jednotek je tedy nahrazen formálními transformacemi

* dokázat korespondenci transformací je obtížné a lze při tom udělat chybu
* proto jsou v praxi mnohem oblíbenější formální metody, pro které existuje podpůrný SW - model checkers, theorem provers (”dokazovače hypotéz”)

* příklady metod formálního vývoje SW: Cleanroom (IBM cca 1987), metody založené na jazycích B a Z

* příklad části jednoduché formální specifikace v modulárním specifikačním jazyce založeném na temporálních logikách:

module *PříkladČítače*

EXTENDS *Naturals*
VARIABLES *cnt*, Čítač.
retval Návratová hodnota.

$TypeInvariant_C \triangleq cnt \in Nat$

$Init_C \triangleq cnt = 0$ Nový čítač se nastaví na nulu.

INTERFACE

cntget $\triangleq$ Zvětší čítač a vrací jeho původní hodnotu.
$\wedge\ cnt' = cnt + 1$ Zvětšíme čítač
$\wedge\ retval' = cnt$ a nastavíme návratovou hodnotu.

THEOREM $Spec_C \Rightarrow \Box TypeInvariant_C$ Typová správnost specifikace.

* využití zejména tam, kde je třeba dokázat zákazníkovi bezpečnost, spolehlivost systému apod.
* cena vývoje srovnatelná s jinými přístupy

Nevýhody:

* vyžaduje specializované znalosti - výběr transformace vyžaduje určitou zkušenost, která se liší od klasického programování
* někdy vyžaduje transformaci do speciálního programovacího jazyka (často čím nižší programovací jazyk, tím obtížnější převod)
* z uvedených důvodů se v praxi příliš nepoužívá

Vývoj orientovaný na znovupoužitelné komponenty

* ve většině SW projektů se stane, že vývojáři znají design nebo kód podobný tomu, který se požaduje

* vezmou, upraví a začlení do svého systému
* děje se nezávisle na modelu vývoje SW

* v posledních letech se objevuje nový způsob vývoje - component-based SWE
* vývoj se spoléhá na velkou množinu SW komponent a nějaký rámec pro jejich integraci
 - komponentou může být např. knihovní fce, třída, podsystém, aplikace (např. Mozilla pro prohlížení nápovědy)
 - rámec - např. příkazové jazyky (Tcl/tk), distribuované objektové architektury (CORBA, JavaBeans) apod.

* komponentově orientovaný proces vývoje postupně:
 - specifikace požadavků
 - analýza komponent
 - modifikace požadavků
 - design systému s využitím komponent
 - vývoj a integrace
 - validace systému

* zatímco první a poslední fáze obdobná jako v ostatních procesech, ostatní fáze se liší

* analýza komponent
 - podle specifikace požadavků vyhledáváme komponenty, které specifikaci splňují
 - obvykle jí ale nesplňují přesně, resp. poskytují pouze část požadovaných funkcí
 - výsledkem je informace o komponentách

* modifikace požadavků
 - analyzujeme požadavky a modifikujeme je tak, aby odrážely dostupné komponenty
 - pokud modifikace není možná, vrátíme se k analýze komponent se snahou najít alternativní řešení

* design systému s využitím komponent
 - během této fáze je navržen rámec systému nebo se využije už existující rámec
 - pokud neexistuje požadovaná komponenta, musí se navrhnout

* vývoj a integrace
 - vytvoříme komponenty, které nemáme (a nemůžeme koupit atd.)
 - komponenty jsou integrovány do systému

* tento model vývoje SW omezuje množství SW, který musíme vyvinout => snížení ceny a omezení nebezpečí, že nastanou potíže
* obvykle také rychleji získáme výsledný SW

* nevýhody:
 - kompromisy vůči požadavkům jsou nevyhnutelné => systém nakonec nemusí splňovat požadavky zákazníka
 - vývoj není zcela v našich rukách, protože nové verze komponent vyvíjí někdo jiný než my (vývoj může být ukončen, nové verze komponenty nemusejí být kompatibilní se starými verzemi) => údržba systému se může prodražit
 - rámce a knihovny komponent bývají tak rozsáhlé, že seznámení s nimi může vyžadovat několik měsíců (ve velkých organizacích bývají určeni specializovaní pracovníci)

❇

KIV/ZSWI 2004/2005
Přednáška 2

Důvody proč (většinou) nefunguje vodopádový model (Parnas a Clemens, 1986):

* uživatelé systému zřídka vědí co přesně chtějí a co vědí neumějí vyjádřit,
* i kdybychom mohli vyjádřit všechny požadavky, na některé datily přijdeme až ve fázi implementace,
* i kdybychom znali všechny detaily, jako lidé neumíme s tak složitými věcmi nakládat,
* i kdybychom s nimi uměli nakládat, vnější síly vedou ke změnám požadavků, některé z nich vedou ke zrušení platnosti dřívějších rozhodnutí.

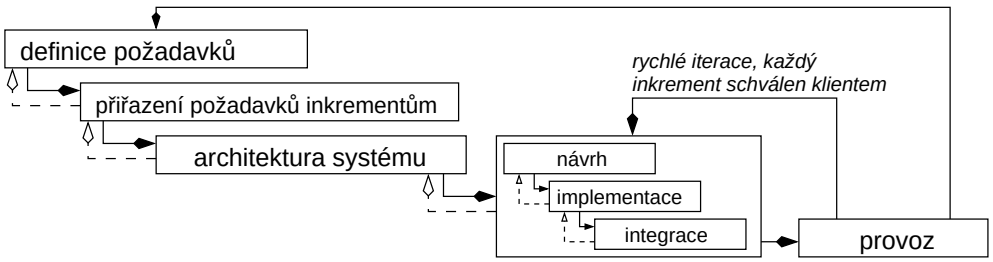
Iterativní modely vývoje SW

* všechny dříve uvedené modely SW procesu mají své výhody a nevýhody
* pro tvorbu většiny velkých systémů je potřeba používat různé přístupy pro různé části (např. podsystémy) => hybridní modely
* pokud se požadavky vyvíjejí => nutnost iterace částí procesu
* model by měl výše uvedené potřeby odrážet - proto uvedu 2 modely navržené speciálně pro iterativní procesy vývoje
  - inkrementální vývoj SW (Lehman 1969, Mills 1980)
  - spirálový model vývoje (Boehm 1988) a WinWin spirálový model (Boehm a Bose, 1994)

Inkrementální vývoj

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

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



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

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

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

* paralelně s vývojem může probíhat analýza požadavků pro další inkrementy

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

* pro vývoj jednotlivých inkrementů se mohou používat různé procesy
 - např. pokud má inkrement dobře definovanou specifikaci => vodopádový model
 - pokud specifikace nejasná => evoluční model

Výhody inkrementálního přístupu:

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

Potíže inkrementálního modelu:

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

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

* v současnosti nejznámější metodika - Extreme Programming (Beck 1999)
 - metodika uzrála v rámci systému pro výplaty ve firmě Chrysler (1996)
 - vývoj malých inkrementů, zatažení zákazníka do procesu, průběžné vylepšování kódu...

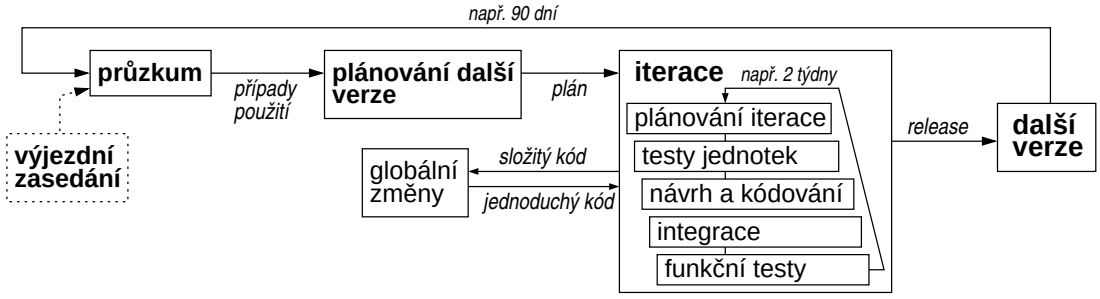
Poznámka (metodika Extreme Programming, XP)
[zdroje: Beck 1999, Benda 2004]

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

 Životní cyklus v XP vypadá přibližně následovně:

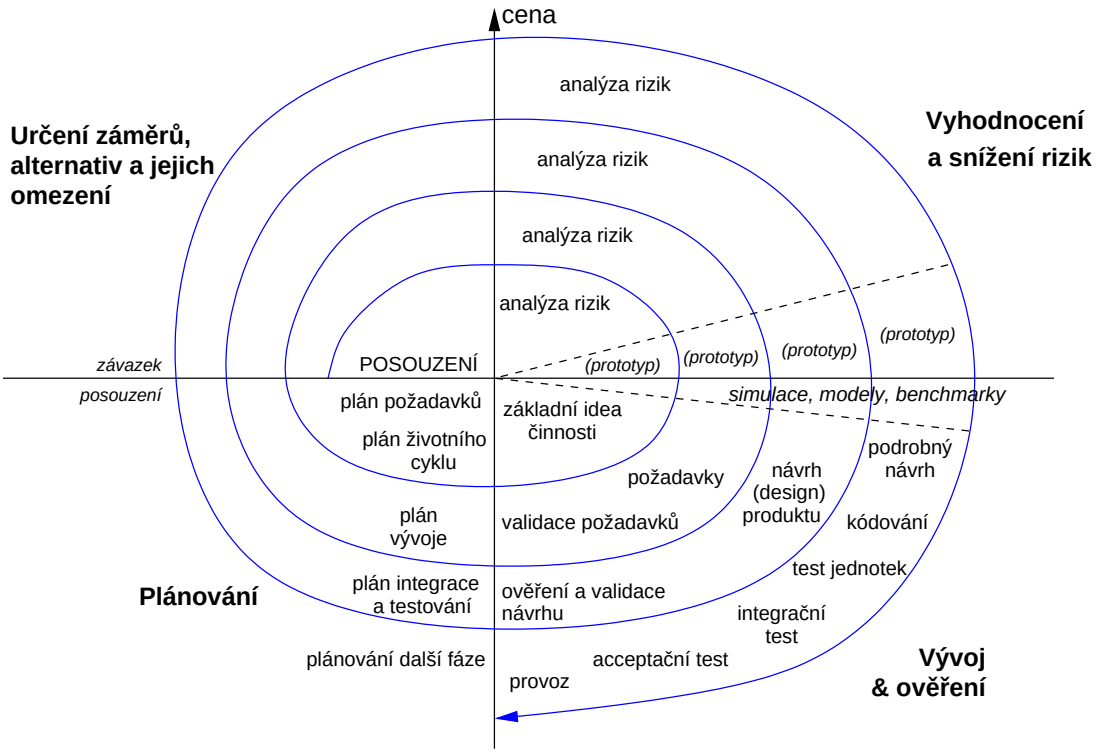
 * průzkum - v ideálním případě krátký
  - zákazník popíše případy použití vytvářeného systému (use cases, v terminologii Extreme Programming jsou nazývány ”stories”)
  - vývojáři na základě případů použití odhadnou, jak dlouho bude vývoj trvat (pokud to pro některý případ použití nelze odhadnout, je nutné ho přepsat, rozdělit, zkusit naprogramovat apod.)
  - vývojáři hledají alternativy pro architekturu systému (pomocí prototypů), identifikace rizik
 * plánování další verze
   - zákazník si vybere nejmenší a pro něj nejcennější množinu případů použití, kterou bude chtít implementovat v první verzi
   - dohodnutí termínu pro dokončení první verze (mělo by být mezi 2-6 měsíci)
    . krátké cykly poskytují včas zpětnou vazbu zákazníkovi, může určit nebo změnit požadavky pro další cyklus
   - rozdělení času do 1-4 týdenních iterací, každá iterace má implementovat funkčnost pro množinu případů použití

```
    . kostra celého systému (architektura) musí být implementována v první
      iteraci => je nutné provést odpovídající výběr případů použití
* každá iterace:
 - rozdělení případů použití na úlohy, rozdělení úloh mezi programátory
 - výběr úlohy (většinou jeden programátor implementuje jeden případ použití)
 - vytvoření testů jednotek
  . v XP se napřed vytvářejí testy, pak teprve kód, který je testován (např.
    ke každé třídě může existovat metoda "test", která jí otestuje)
  . testy by měly prověřit veškerou logiku budoucího kódu
  . při psaní testů se často vynořují alternativy návrhu
  . někdy se ukáže potřeba globální změny v systému - refactoring
    problematického kódu směrem k větší jednoduchosti
 - návrh a kódování
  . návrh i kód by měly být co nejjednodušší (pouze nyní požadované
    vlastnosti, žádné předpoklady do budoucnosti - protože požadavky
    zákazníka se budou měnit)
  . kód je považován za dokončený, pokud všechny testy proběhnou úspěšně
  . jednoduchost návrhu a testy zjednodušují budoucí přepracování kódu
 - integrace systému, testy jednotek
  . dokončená funkčnost je integrována do systému
  . spustí se automatické testy jednotek pro celý systém, tím zjistíme zda
    jsme omylem nezasáhli do funkčnosti systému
 - funkční testy
  . mají záběr do více podsystémů, někdy mají i manuální podobu (testování
    funkčnosti systému z pohledu zákazníka)
```




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

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

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

[]

Další agilní metodiky jsou např. DSDM, Scrum, FDD (viz
http://www.agilealliance.org).


Spirálový model SW procesu
..........................
[zdroj: Boehm 1988]

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

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



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

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

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

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

* plánování
 - naplánujeme další fázi projektu (organizace, požadavky na zdroje, rozdělení práce, časový plán, výstupy)
 - každá otočka je zakončena posouzením všech produktů vytvořených v předchozím cyklu, plánů pro další cyklus a potřebných zdrojů
  . cílem je aby byly všechny zúčastněné strany srozuměny s plány na další fázi
  . výsledkem např.: 15 lidí na 12 měsíců + základní plán životního cyklu pro alternativu, která se vynoří
  . může určit i rozdělení projektu do inkrementů nebo do nezávisle vyvíjených komponent

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

Rozdíly oproti předchozím modelům:

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

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

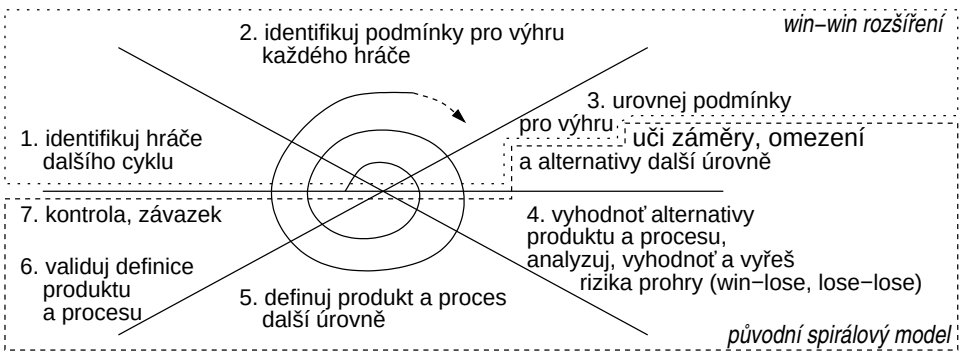
Původní článek o spirálovém modelu doporučuje následující Risk Management Plan, který lze používat i samostatně mimo spirálový model:

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

WinWin spirálový model

[zdroje: Boehm a Ross 1989, Boehm a Bose, 1994]

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



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

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

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

Ostatní části jsou stejné jako u původního spirálového modelu.

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

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

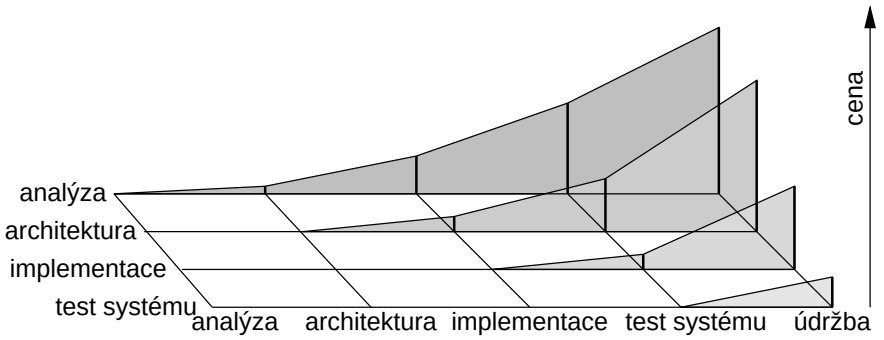
* v dalším povídání se seznámíme s některými metodologiemi
 - metodologie = jak provést příslušnou fázi
 - už na začátku bylo řečeno, že základní aktivity SW procesu jsou 1. analýza domény a definice požadavků, 2. design systému a design SW, 3. implementace a testování modulů, 4. integrace a testování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

Analýza domény a definice požadavků

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

Poznámka (cena změn):

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



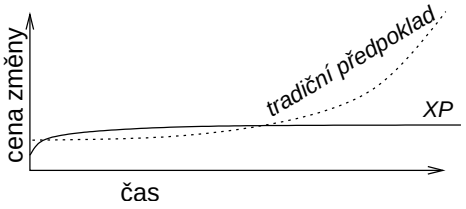
Např. defekt analýzy požadavků, jehož oprava by nás stála 1000 Kč během fáze analýzy požadavků, nás může stát 25 000 Kč během závěrečného testování systému (více u velkých systémů, méně u malých systémů).

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

[]

Poznámka pro zajímavost (cena změn a metodika Extreme Programming)

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



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

[]

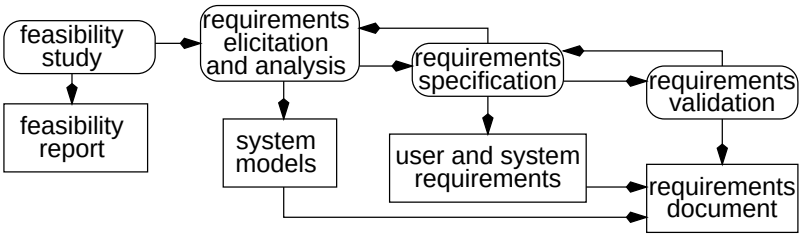
Při specifikaci požadavků na rozsáhlé systémy obvykle rozlišujeme 4 fáze:

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

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

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

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



Studie proveditelnosti

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

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

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

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

Výsledná studie by:

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

Analýza domény a zjištění požadavků

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

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

Obecný model analýzy požadavků vypadá takto:

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

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

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


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

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

Typy požadavků
..............

Požadavky na systém lze rozdělit do následujících tříd:

* funkční požadavky
* mimofunkční požadavky
* doménové požadavky

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

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

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

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

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

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

Poznámka (příklad mimofunkčních požadavků a jejich metrik)

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

[]

Poznámka (pojem ”robustní”)

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

[]

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

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

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

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

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

Příklad (uživatelská specifikace požadavků):

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

Příklad (systémová specifikace požadavků):

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


Doporučení:

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

Příklad:
```

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

1.1 Uživateli bude poskytnuta možnost definovat typy externích dokumentů.

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

1.3 . . .

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

❇

KIV/ZSWI 2004/2005
Přednáška 3

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 určovaly vlastnosti produktu.

[]

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

Poznámka (osnova DSP na stránkách předmětu 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

[]

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

Popíšeme si zatím:

* přirozený jazyk (a několik doporučení)
* formuláře
* případy použití
* pseudokódy a specifikace rozhraní.

Přirozený jazyk

* požadavky jsou obvykle popsány přirozeným jazykem - 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
....

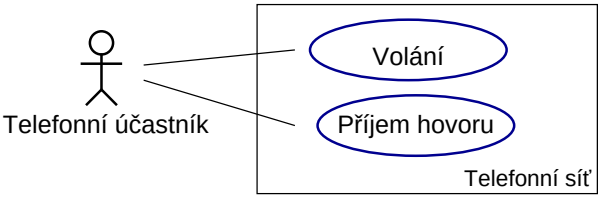
* use-cases [Jacobson at al. 1993]
* používají se zejména pro popis kontextu systému a pro popis funkčních požadavků
* základ některých metodik, např. sběr požadavků v RUP, XP apod.

* 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, se nazývají 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 pro zajímavost (pseudokódy nižší úrovně)

* někteří autoři (např. Sommerville) doporučují používat pseudokód odvozený z konkrétního programovacího jazyka, např. z Javy nebo Pascalu
  - výsledkem často zbytečně detailní specifikace, místo popisu záměru určuje implementaci (= omezuje jí)
  - notace je srozumitelná pouze lidem se znalostí programovacího jazyka (uživatel by měl specifikaci požadavků pokud možno rozumět)
  - často méně přehledná

Příklad (pseudokód odvozený z Javy):

```
class Bankomat {

    public static void main(String args[]) throws NeplatnáKarta {

        try
        {
            tatoKarta.přečti();
            obrazovka.výzva(”Prosím zadejte PIN”);
            PIN = klávesnice.přečtiPIN();
            pokusů = 1;
            while (!tatoKarta.PIN.rovno(PIN) & pokusů < 3)
            {
                obrazovka.výzva(”Chybné PIN - prosím zadejte znovu”);
                pokusů = pokusů + 1;
                PIN = klávesnice.přečtiPIN();
            }

            if (!tatoKarta.PIN.rovno(PIN))
                throw new NeplatnáKarta(”Chybné PIN”);
            // Zde bychom již měli mít platné PIN.
            ...
```

[]

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 jsou 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 být 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 zjišťují požadavky na systém od zástupců zadavatele
 - je zapotřebí počítat s tím, že uživatelé mohou mít potíže své požadavky vyjádřit, mohou 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:

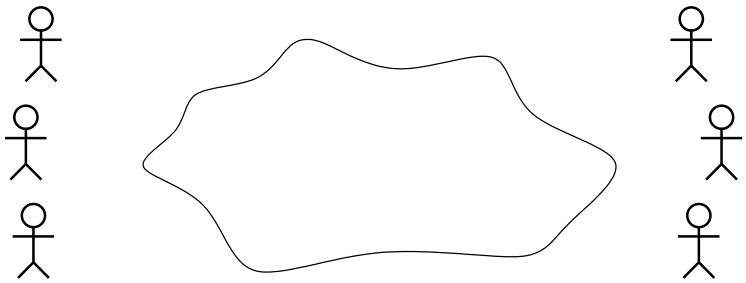
* identifikace aktérů
* identifikace případů použití
* vytvoření popisu pro každý případ použití
* popis toku událostí pro každý případ použití
* strukturování případů použití
* identifikace analytických tříd atd.

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 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?

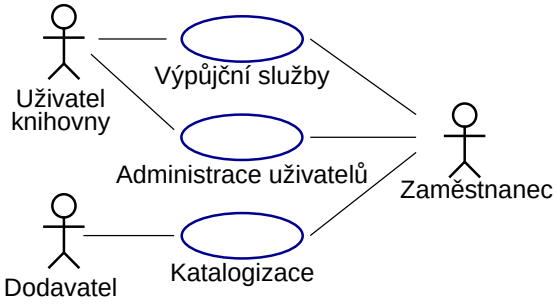
[]

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 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 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 požadavky zpět k jejich zdroji (např. z DSP zpět k požadavku v ConOps dokumentu, ze kterého vyplývá), dopředu k návrhu 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 je DSP změněn nekonzistentně se změnou implementace.

❇

KIV/ZSWI 2004/2005
Přednáška 4

Úvod do OO terminologie

> I should mention that I was the one who made up the unfortunate term ”object-oriented” in the mid60s (I should have called it something else). In any case, I have some strong ideas about what this term means and should mean. --Alan Kay, 02/2004

Za počátek objektově orientovaného programování se považuje vznik jazyka Simula67 (Norwegian Computing Centre, asi 1967). Prvním čistě objektově orientovaným jazykem byl jazyk Smalltalk-80, vyvinutý v Xerox PARC (spolu s prvním osobním počítačem, prvním okenním prostředím, myší, Ethernetem atd.). Čistě objektově orientovaný jazyk = všechno je objekt, včetně základních datových typů a řídících konstrukcí. Později se objevily další OO jazyky (C++, Objective C, Eiffel, Python, Java (čti [džava]), C# (čti [sí šarp] nebo česky [cis])...

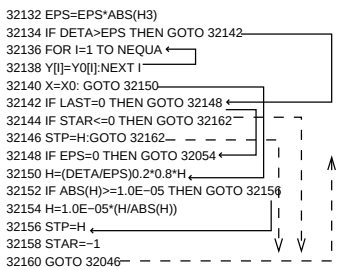
Objektová orientovanost se postupně rozšířila do všech fází vývoje SW - vznikly a stále ještě vznikají objektově orientované metodiky pro analýzu, návrh, implementaci, testování, zpětné inženýrství (reverse engineering) a přepracování kódu (refactoring) atd.

V dalším textu si nejprve uvedeme základní motivaci, terminologii a poté metodiku OO analýzy a návrhu.

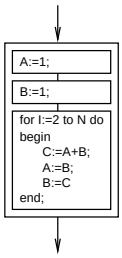
Proč objektově orientované programování?

Vývoj nástrojů pro tvorbu SW (např. programovacích jazyků) odráží nutnost vytvářet a zacházet se stále rozsáhlejšími celky. Nemáme-li k dispozici přiměřenou technologii, nazýváme takový stav ”krize”; řešením krize je změna paradigmatu (např. postupně rozšíření strukturovaného, objektově orientovaného a komponentově orientovaného programování).

* historicky první programovací jazyky (asseblery a FORTRAN) byly nestrukturované
  - základní řídící konstrukcí byl skok - příkaz GOTO (spustí provádění instrukcí od zadaného návěští)
  - srozumitelnost silně klesá s velikostí projektu
  - proto snaha vnést strukturu - tzv. strukturované programování



a) nestrukturovaný kód v jazyku BASIC



b) strukturovaný kód v jazyku Pascal

* strukturované programování podporují všechny moderní imperativní programovací jazyky
  - ve strukturovaném kódu snadněji vidíme, které části jednoho podprogramu se k sobě vztahují a jak
  - s rostoucí velikostí programů ale opět problém: příliš mnoho věcí a vztahů, které musíme brát do úvahy najednou
    . ”věci” = podprogramy, funkce a data
    . ”vzájemné vztahy” = které podprogramy na sobě závisejí, jaké používají datové struktury
  - proto sdružování souvisejících podprogramů a dat do modulů tak, aby změna jednoho modulu měla minimální dopad na obsah ostatních modulů

a) jeden celek b) totéž rozděleno do modulů

* strukturovaně navržené systémy se ale časem obtížně udržují
 - při strukturovaném programování se zabýváme zvlášť funkcemi a zvlášť daty
  . fce jsou aktivní, mají chování
  . data jsou pasivní, jsou zpracovávána funkcemi
  . problém: fce musejí znát strukturu dat
  . pokud se změní struktura dat (změna požadavků), je třeba změnit všechny fce pracující s datovou strukturou
  . pokud máme zpracovávat podobná data, do všech fcí musíme přidat podmínky
  . tím vzrůstá ”entropie” (neuspořádanost) SW; pokud dosáhne určitého limitu, cena modifikací naroste tak, že je ekonomicky neobhajitelné systém nadále udržovat => je nutné systém přestrukturovat

* související problém: sémantická mezera mezi externím a interním pohledem na systém (tj. mezi aplikační doménou a vytvářeným systémem)
 - potřebujeme, aby malá změna požadavků vedla k malé změně systému
  . pozorování: ”co” má systém dělat se mění méně, než ”jak” to má systém dělat
  . řešení - mapování mezi skutečností, konceptuálním modelem a systémem by mělo být co nejjednodušší (např. faktura -> datový typ ”faktura”)

* další problém - znovupoužitelnost kódu v dalších aplikacích
 - proč knihovny procedurálních programovacích jazyků (C, Pascal, FORTRAN...) obsahují pouze fce poměrně nízké úrovně?
 - v klasických procedurálních jazycích nelze vysokoúrovňovou funkčnost snadno přenést z jednoho kontextu do druhého - nutno vytvářet znovu
 - řešení: principy abstrakce, zapouzdření a dědičnosti podporované programovacím jazykem = objektově orientované programování (viz dále)

* další krok po OO stejným směrem bude pravděpodobně komponentově orientované programování a objektově orientované aplikační rámce

Základní koncepce OO technologie

* pojem ”objektově orientovaný” znamená přibližně to, že SW je organizován jako množina objektů, které spolu komunikují
 - objekty slučují data a nad nimi pracující fce => změny budou lépe lokalizovány
 - tím, že problém strukturujeme do objektů existujících v aplikační doméně => menší sémantická mezera mezi aplikační doménou a modelem
 - některé objekty mohou sloužit jako znovupoužitelné komponenty (např. univerzální kontejner)
 - pokud celý vývoj probíhá objektově orientovaným (dále jen ”OO”) způsobem, můžeme pro všechny fáze vývoje používat stejnou terminologii a grafickou notaci (i když objekty budou na různé úrovni obecnosti/abstrakce)

* na začátku analýza požadavků = zabývá se vytvářením konceptuálního modelu studovaného systému
* při postupu k implementaci zjemňujeme předchozí model přidáváním detailů a nových objektů poskytujících funkčnost
* protože informace je ukryta v objektech, může být rozhodnutí o konkrétní realizaci dat odloženo až na implementaci
* v některých případech můžeme obdobně odložit rozhodnutí o distribuci objektů a zda budou sekvenční nebo paralelní - může záviset na prostředí, kde budou běžet

* např. v analýze požadavků modelujeme systém jako množinu interagujících objektů aplikační domény, tj. entit a operací sdružených s řešeným problémem (např. nemocnice: pacienti, sestry, lékaři, výkony...)
* v OO návrhu vytváříme OO model SW systému, který bude požadavky identifikované v analýze implementovat

- některé objekty aplikační domény budou odpovídat objektům implementace, ale při postupu k implementaci je třeba vytvářet další objekty
* při OO implementaci realizujeme systém v OO jazyce jako je např. Java

Poznámka (modelovací jazyk UML)

Podobně jako se v elektrotechnice používají schémata, používají se při návrhu SW systémů modely. Různé modely zachycují různé klíčové vlastnosti studovaného systému a vynechávají detaily (podobně jako el. schémata vynechávají velikost a rozmístění prvků). Téměř všechny modely mají nějakou grafickou notaci s textovým popisem.

Pro popis objektově orientovaných modelů budeme používat notaci UML (Unified Modeling Language, 1996), která se stala de-facto standardem pro modelování OO systémů. UML podporuje mnoho CASE nástrojů (Rational Rose, Together, MagicDraw, Fujaba...).

Před vznikem UML existovalo nejméně 50 různých notací, některé z nich můžete ještě najít např. v nových překladech starší literatury.

[]

V literatuře existují určité neshody v tom, co přesně znamená pojem ”objektově orientovaný”, nicméně většina autorů se shoduje na těchto základních principech:

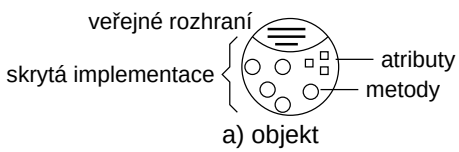
1. abstrakce
2. zapouzdření
3. dědičnost
4. polymorfismus

Abstrakce
....
[Rumbaugh: OMT, Sigfied: Understanding OOSE, Cox: OOP]

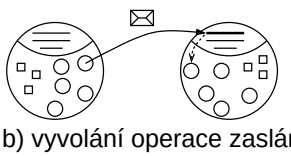
* abstrakce = zaměření se na podstatné vlastnosti entity a zanedbání detailů, které jsou pro daný účel nepodstatné
 - při OO modelování umožňuje zaměřit se nejprve na to, co objekt je a co dělá
 - detaily (např. návrh a implementaci) doplníme, až budeme problému lépe rozumět
 - nakonec se sice musíme detaily zabývat, ale můžeme je např. vhodně uspořádat za použití vrstvené abstrakce (vrstvená abstrakce = vysokoúrovňová abstrakce je popsána pomocí nízkoúrovňových abstrakcí)

Zapouzdření
....
[Gosling-McGilton: The Java Language Environment. 1996]

* zapouzdření (encapsulation) = entita má dobře definované rozhraní a ukrytou vnitřní reprezentaci
* v OO řešení problému je jednotkou zapouzdření objekt
 - zveřejňuje rozhraní - množinu nabízených operací
 - skrývá implementaci
 - objekt můžeme používat podle jeho specifikace nezávisle na vnitřní implementaci (která se může změnit)



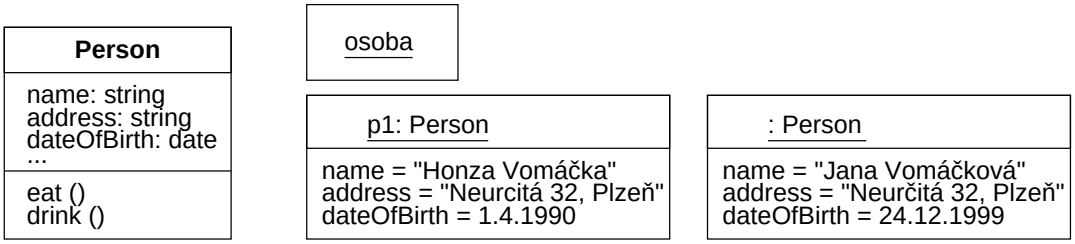
a) objekt


b) vyvolání operace zasláním stimulu

Příklad:

Příkladem zapouzdření v reálném světě je např. auto: ovládá se pomocí volantu, spojky, brzdy, plynu atd. - téměř všechna auta ovládáme podobně

```
 nezávisle na jejich vnitřní implementaci.

Objekt
......

* objekt je entita, která má jedinečnou identitu, stav a množinu operací,
  které pracují se stavem (mění stav, zjišťují stav, vyvolají určité chování)
* identita objektu trvá po celou dobu existence objektu, pomocí ní se na
  objekt odkazujeme
* stav objektu je reprezentován množinou atributů objektu
* operace sdružené s objektem poskytují služby jiným objektům (klientům)
* klienti vyvolají operaci zasláním stimulu (stimuly se často nazývají
  ”zprávy”, i když nemusejí nutně nést informaci)

Příklad (objekty na vysoké úrovni abstrakce):

  Objekty mohou odpovídat objektům reálného světa. Objektem na určité úrovni
  abstrakce může být např. ”Honza Vomáčka”, jeho atributy budou např. ”věk 18
  let”, ”výška 177 cm”, operacemi může být ”jez”, ”pij” apod.

[]

Třída
.....

* třída objektu popisuje skupinu objektů stejnou strukturou (atributy),
  chováním (operacemi), vztahy k ostatním objektům a významem
 - objekty dané třídy mají obdobný význam - např. i když kůň i stáj mají
   atributy ”stáří” a ”cena”, pravděpodobně budou patřit do různých tříd
 - sdružováním objektů do tříd abstrahujeme společné vlastnosti objektů

* objekty stejné třídy = instance třídy
* atributy i operace objektů jsou deklarovány jejich třídou
* v OO jazycích se objekty vytvářejí podle definice třídy

Příklad (třída na vysoké úrovni abstrakce):

  Objekty ”Honza Vomáčka” a ”Kateřina Drsná” mohou být instancemi třídy
  Člověk. I když se ”Kateřina Drsná” vdá a bude mít nové jméno, její identita
  se tím nezmění (identita je nezávislá na stavu objektu).

[]

* notace pro třídy a instance v UML:
 - třídy - znázorňují se jako obdélník se jménem třídy (jméno tučně a
   vycentrovat) a dvěma volitelnými sekcemi
  . střední sekce = atributy objektu (”attributes”)
  . spodní sekce = operace objektu (”operations”)
 - instance - podobně, jméno ve tvaru:   ”objekt” : ”Třída”  (podtržené),
   lze zkrátit i jen na ”objekt” nebo  : ”Třída” (viz obrázek)
  . případně atributy, které nás zajímají, a jejich hodnota
```



a) třída b) instance třídy

```
* termín ”operace” se používá pro akci, termín ”metoda” pro implementaci
  operace
* v OO jazycích se metody se spouštějí zasláním zprávy buď samotné třídě, aby
  vytvořila objekt, nebo již vytvořenému objektu
* zprávu zašleme / metodu vyvoláme např.
```

```
        adresát_zprávy.metoda(argumenty);     // Java, C#
  nebo
        [ adresát_zprávy metoda argumenty ]   // Objective C
```

* v některých distribuovaných systémech je zaslání zprávy implementováno přímo odesláním textové zprávy - příjemce ve zprávě najde požadovanou operaci a data, podle názvu operace určí metodu, vyvolá ji a předá jí data
* pokud objekty koexistují ve stejném programu, vyvolání metod je implementováno obdobně jako volání procedury v Pascalu nebo v C - komunikace je synchronní
* pokud jsou objekty implementovány pomocí paralelních procesů nebo vláken, může být operace asynchronní => volající může pokračovat, zatímco se služba vykonává, popíšeme později

Dědičnost

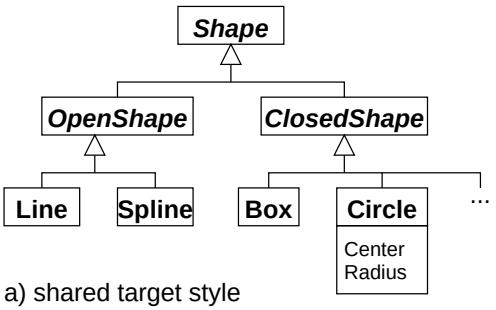
* angl. inheritance
* některé třídy budou mít společné vlastnosti
* např. třídy Muž a Žena budou mít mnoho společných vlastností (operace ”jez” a ”pij”, atributy ”jméno”, ”adresa”, ”datum narození”)

* společné vlastnosti můžeme sdílet tak, že je vyjmeme a vložíme do samostatné třídy Osoba (používají se termíny generalizace, zobecnění)
 - ostatní třídy tyto společné vlastnosti (atributy a operace) mohou sdílet mechanismem dědění
 - ve třídách Muž a Žena můžeme popsat jenom nové vlastnosti
 - pokud je zapotřebí modifikovat společné chování, stačí změnit v definici Osoby

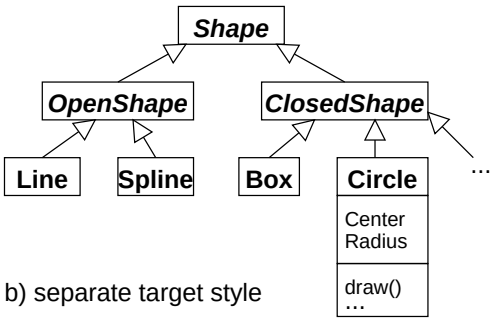
* někdy provádíme specializaci existující třídy - najdeme třídu, která poskytuje operace a atributy potřebné pro novou třídu, nová třída je zdědí a přidá nové vlastnosti
 - např. specializací třídy Zaměstnanec může být Programátor, který bude mít další atributy (”programovací jazyk”) a operace (”programuj”)

* zobecňováním a specializací vzniká hierarchie tříd
 - předkové, nadtřídy (ancestors, super-classes) - jsou obecnější, jednodušší
 - potomci, podtřídy (descendants, sub-classes) - specializovanější, složitější

* v UML se dědičnost znázorňujeme šipkou, která vede k rodičovské třídě
 - tři tečky (...) znamenají, že část modelu není zobrazena
 - znázorněny jsou dvě varianty pro kreslení šipek



a) shared target style

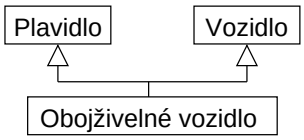


b) separate target style

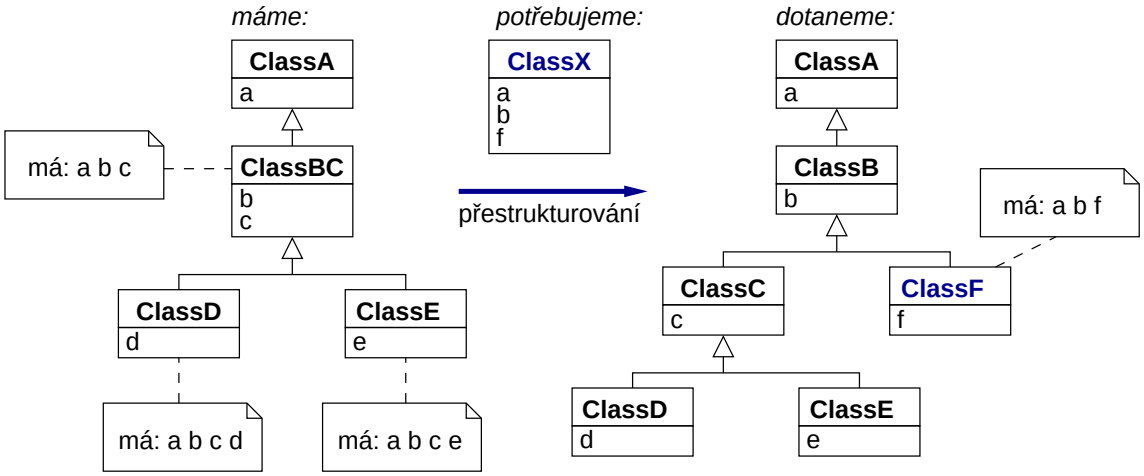
* předkové vytvoření pouze pro účel dědění ostatními se nazývají abstraktní třídy - nevytvářejí se z nich instance
 - v UML se název abstraktní třídy zobrazuje kurzívou

* dědičnost může být jednoduchá a vícenásobná (multiple inheritance)
 - jednoduchá = každá třída dědí pouze z jedné rodičovské třídy
 - vícenásobná = třída může mít více než jednoho rodiče
  . vícenásobná dědičnost zvyšuje složitost hierarchie tříd, pravděpodobnost koncepčních chyb, je obtížně srozumitelná atd., proto jí mnoho OO jazyků nepodporuje (např. Java; toto omezení je možné v případě potřeby obejít vytvářením tzv. složených tříd)

. v UML vícenásobnou dědičnost znázorňujeme:



* někdy musíme hierarchii tříd přestrukturovat, abychom získali vhodnou třídu, ze které bychom mohli dědit
 - například v následujícím obrázku máme hierarchii tříd s vlastnostmi ”a”, ”abc”, ”abcd”, ”abce” a potřebujeme třídu s vlastnostmi ”abf”
 - ze současné hierarchie tříd není možné takovou třídu získat děděním, musíme přestrukturovat



* ve výše uvedeném obrázku také ukázka UML notace pro poznámky:
 - obdélník s přehnutým pravým horním rohem, připojen přerušovanou čarou

* použití dědičnosti v praxi:
 - sdílení kódu - při návrhu mohu sdílet kód mezi podobnými třídami
 - mohu si přizpůsobit existující třídu (např. z minulého projektu)
 - koncepční zjednodušení - zmenšení počtu nezávislých vlastností systému

* tradiční pravidla pro použití dědičnosti:
 - nová třída musí obsahovat všechny atributy původní třídy, lze přidávat nové atributy
 - nová třída musí rozumět stejným zprávám jako původní třída, lze přidávat nové operace
 - implementace metody může být v podtřídě změněna, neměl by se ale změnit její význam (pak už by nebylo čisté zobecnění/specializace)

* nesprávné použití dědičnosti - pokud mezi třídami není vztah zobecnění/specializace ale např. vztah kompozice (okno není specializace domu, ale dům může kromě jiného obsahovat okna)
 - jinými slovy: dědičnost bychom neměli používat pouze jako mechanismus pro ”vypůjčení si” kódu bez ohledu na cokoli, protože tím zaneseme do modelu konceptuální problémy a do kódu skryté předpoklady

Polymorfismus

* pokud bychom v klasickém procedurálním programovacím jazyku potřebovali vytisknout dva geometrické obrazce (reprezentované datovými strukturami typu ”čtverec” a ”obdélník”), nejspíše bychom pro to měli k dispozici samostatné procedury:

```
tiskni_čtverec(a);
tiskni_trojúhelník(b);
```

* v OO jazycích najdeme mechanismus polymorfismu = možnost zasílat stejnou

```
  zprávu různým objektům, které na ni odpoví každý svým způsobem
 - např. ve výše uvedeném případě může každý objekt mít metodu ”vytiskni_se”,
   vyvoláme:

  a.vytiskni_se;  // vytiskne se čtverec
  b.vytiskni_se;  // vytiskne se trojúhelník

 - příjemce zprávy může patřit do libovolné třídy implementující metodu
   ”vytiskni_se”, vysílající nemusí znát konkrétní třídu příjemce
 - pokud bychom měli např. seznam skládající se ze čtverců a trojúhelníků,
   mohli bychom napsat něco jako:

   foreach anObject in list  // projdi všechny objekty v seznamu ”list”
      anObject.vytiskni_se;  // vyvolej jejich metodu ”vytiskni_se”

 - například každý objekt může mít metodu ”ulož se do souboru” apod.
 - zvýšení flexibility

* většinou se polymorfismus omezuje (omezený polymorfismus)
 - například zprávu ”vykresli se na obrazovku” má smysl posílat pouze
   grafickým objektům (čára, obdélník...)


Paralelní objekty
.................

* objekty koncepčně žádají o provedení služby zasláním ”zprávy”, v tom není
  explicitní požadavek na sériové vykonávání
 - např. pokud bychom chtěli vytisknout soubor (a.printFile(f)), bylo by
   přirozené, kdyby volající nemusel čekat na dokončení tisku
 - obecný model dovoluje objektům běžet paralelně (např. na stejném počítači
   nebo jako distribuované objekty na různých strojích)
 - kdyby se ale jednalo např. o zavření dveří výtahu (a.closeDoors), mohlo by
   mít paralelní provedení nepříjemné důsledky
 - většinou předpokládáme návrat ve chvíli, kdy je to bezpečné
 - v UML se paralelismus operace označuje připojením vlastnosti
  { concurrency = concurrent }
```

Printer
printFile() {concurrent}

```
* většina jazyků implementuje vyvolání metody stejně jako volání fce, tj.
  volající objekt je pozastaven do dokončení operace
* některé jazyky (např. Java) umožňují vytvářet paralelně běžící objekty

* existují dva druhy implementace paralelně běžících objektů:
 - servery - objekt je implementován jako paralelně běžící proces, metody
   odpovídají operacím
  . metoda se spustí jako odpověď na příchozí požadavek, může běžet paralelně
    s metodami sdruženými s ostatními objekty
  . po dokončení se objekt pozastaví a čeká na další zprávu = požadavek
 - aktivní objekty - stav objektu se může měnit interní operací samotného
   objektu
  . proces představující objekt běží neustále nebo svůj stav upravuje v
    určitých intervalech, např. v RT systému zjišťuje stav okolí
  . např. v Javě - udržuje informaci o pozici letadla pomocí satelitního
    navigačního systému:

        class Transponder extends Thread {

                Position currentPosition;
                Coords c1, c2;
                Satellite sat1, sat2;
                Navigator navigator;

                public Position givePosition() {
```

```
                return currentPosition;
        }

        public void run() { // zavolá se po vytvoření vláken
                while (true) {
                        c1 = sat1.position();
                        c2 = sat2.position();
                        currentPosition = navigator.compute(c1, c2);
                }
        }
} //Transponder
```

Modelovací jazyk UML

* modelování = návrh aplikace před kódováním
* model hraje stejnou roli při vývoji SW jako studie a projektová dokumentace při stavbě domu
  - tj. slouží pro vizualizaci návrhu, kontrolu a dokumentaci před zahájením realizace (kódování)
* model abstrahuje základní detaily skutečnosti nebo vytvářeného systému
* standardní notace pro OO modelování je UML

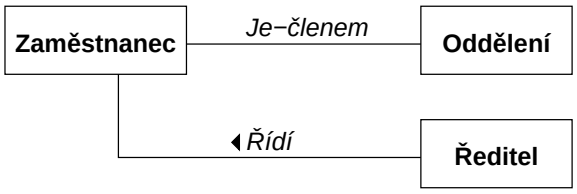
* UML definuje 12 typů diagramů rozdělených do 3 kategorií:
  - strukturální diagramy (diagram tříd, diagram objektů, diagram komponent a diagram nasazení - všechny popisují statickou strukturu systému)
  - diagramy chování (diagram případů použití - některé metodiky ho používají pro sběr požadavků, diagram spolupráce a sekvenční diagram - pro popis komunikace, stavový diagram a diagram činností)
  - diagramy pro správu a strukturování modelů (balíčky, podsystémy a modely)

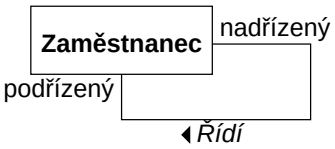
Diagram tříd a diagram objektů

* diagram tříd ukazuje třídy a vztahy mezi nimi (zobecnění, asociace, agregace...)
  - vztahy zobecnění (dědičnosti) už byly popsány, ukážeme si další typy vztahů

Asociace

* asociace říká, že objekty které jsou instancemi jedné třídy, mohou mít vztah s objekty jiné nebo stejné třídy
* např. objekty třídy Zaměstnanec budou mít asociaci k objektům třídy Oddělení
* v UML se znázorňují plnou čarou mezi třídami, u čáry volitelně název vztahu
  - u názvu volitelně malý černý trojúhelníček ukazující, kterým směrem se má název vztahu číst
  - asociace může být i rekurzivní

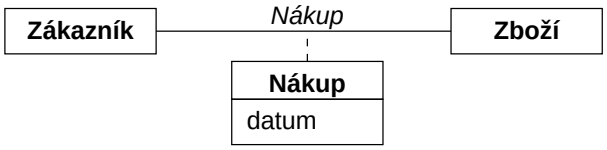




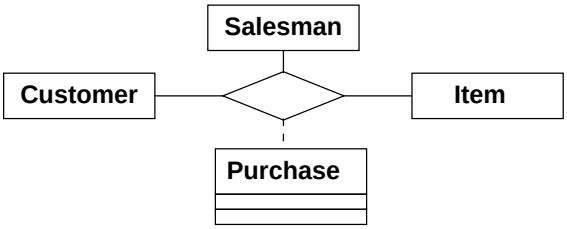
  - konce asociace mohou být volitelně popsány rolemi ve vztahu
    . role popisující vzdálené konce asociací stejné třídy mají být jedinečné
    . u rekurzivních vztahů (nadřízený řídí podřízeného) by role měla být uvedena vždy
    . pojmenování rolí užitečné, pokud je více než jedna asociace mezi stejným párem tříd

* pokud má asociace vlastnosti jako atributy, operace a další asociace, můžeme pro ní vytvořit tzv. asociační třídu (analogie ”asociativního indikátoru typu” v ERA diagramech)

- příklad: zákazník nakoupil zboží
  . asociace ”nakoupil” sdružuje zákazníky a položky zboží
  . pokud bychom chtěli uchovat informaci o datu nákupu atd., nepatří ani k zákazníkovi, ani ke zboží



* všechny dosavadní asociace byly binární, tj. do vztahu vstupovaly dvě strany
 - binární asociace jsou zdaleka nejčastější, občas se může vyskytnout ternární atd.
 - n-ární asociaci můžeme znázornit pomocí prázdného kosočtverce
 - následující obrázek ukazuje ternární asociaci, která je zároveň asociační třídou:



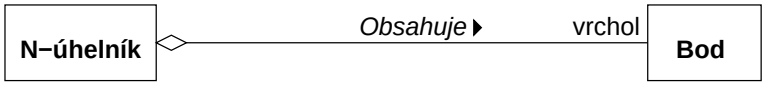
* asociace je velmi obecný typ vztahu, v UML se často používá pro označení, že některý atribut je asociovaný objekt nebo že implementace některé metody závisí na asociovaném objektu
 - v počátečních fázích návrhu je vhodné nezahrnovat asociace, které bezprostředně nepotřebujeme
 - v konečných fázích návrhu je dobré konkretizovat (např. vyjádřit vztah agregace a kompozice)

Agregace
.....

* jednou z nejčastějších binárních asociací je agregace
* objekt je vytvořen z dalších objektů = je agregátem množiny objektů
 - např. objekt Stádo je agregátem Ovcí, Les je agregátem Stromů, Rodina bude agregátem objektu typu Muž, objektu typu Žena a množiny objektů Dítě
 - nebo Předmět se může skládat z Přednášek, Cvičení, Zápočtové_úlohy, Zkoušky atd.

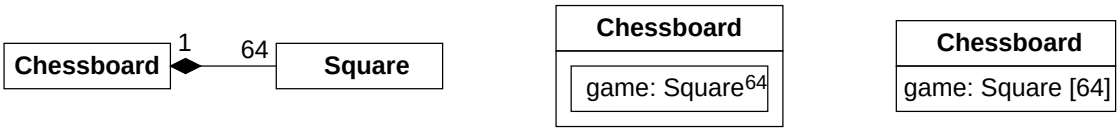
* agregace je více než pouze součet svých částí, vzniká něco nového (agregát)
 - agregát může vystupovat v některých operacích jako samostatná jednotka
 - části mohou existovat samostatně, mohou být součástí dalších agregací

* v UML se agregace znázorňuje prázdným kosočtvercem na straně agregátu:



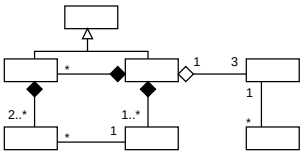
Kompozice
.....

* kompozice je silná asociace - součást náleží právě jednomu složenému objektu
 - součást nemůže existovat samostatně (políčko nemůže existovat bez šachovnice)
 - při zániku celku tedy zaniknou i jeho části
 - v UML se kompozice znázorňuje plným kosočtvercem nebo grafickým vnořením:



- kompozice je jediný vztah, který smíme modelovat ”pohřbením” objektů jako atributů jiného objektu (objekt silně patří jediné třídě)
  . kromě tohoto případu se reference na objekty nemají vyskytovat jako atributy objektů, ale mají být vždy modelovány jako příslušné asociace

* v jednom diagramu mohou vystupovat různé typy vztahů:

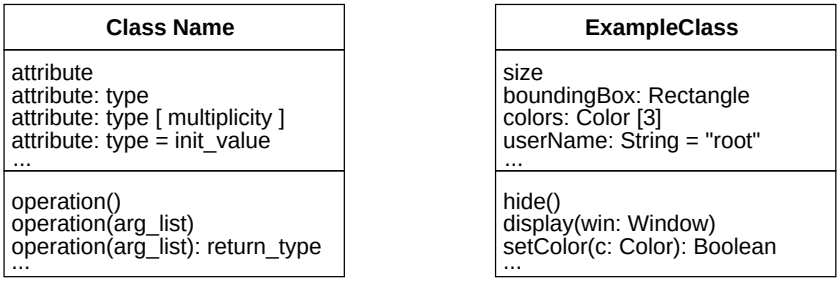


[na přednášce bude uveden konkrétní příklad]

* v příkladu uvedena také násobnost (kardinalita) vztahu = počet elementů, které mohou vstoupit do vztahu; uvádí se intervalem nebo výčtem
 - intervalem: 0..1 = žádný nebo jeden, 1..* jeden a více, * je totéž co 0..*
 - výčtem: 1..3,7,10
 - pokud násobnost není uvedena, nelze o ní nic předpokládat (kromě kompozice, kde plný kosočtverec zároveň značí násobnost 1)

Rozhraní a jeho realizace

* specifikace atributů a operací ve třídě:



- specifikace atributů:
  <atribut> : <typ> [<násobnost>] = <počáteční hodnota>
- specifikace operací:
  <operace> (<seznam parametrů>) : <typ návr.hodnoty>
  kde parametr má tvar: <in|out|inout> <parametr> : <typ> = <default.hodnota>
- defaultní je ”in”; volba ”in”, ”out” nebo ”inout” zvýrazněně
- typy jsou závislé na implementačním jazyce
- před atributem nebo operací může být uvedena <viditelnost>
  . ”+” public = viditelné (přístupné) pro všechny třídy
  . ”#” protected = viditelné pouze uvnitř třídy a pro potomky
  . ”-” private = viditelné pouze uvnitř třídy, není viditelné ani pro potomky
  . ”~” package = viditelné pouze uvnitř balíčku

* rozhraní (interface) je specifikace navenek viditelných operací třídy, komponenty apod.
 - specifikuje pouze rozhraní pro operace bez jejich implementace
 - nemůže mít atributy, stav nebo asociace
 - dva způsoby zakreslení:
  . podobně jako třída, nad jménem rozhraní klíčové slovo <<interface>>
  . část s atributy můžeme vynechat (je vždy prázdná)
  . pokud třída rozhraní realizuje (tj. pokud implementuje všechny operace

rozhraní), znázorňujeme to přerušovanou šipkou s trojúhelníkovou hlavou, která směřuje od třídy k rozhraní
- rozhraní se může kreslit také jako kroužek spojený plnou čarou s třídou, která ho implementuje, pod kroužkem název rozhraní

* rozhraní mají v mnoha OO jazycích přímou realizaci (v Javě klíčová slova interface a implements)
 - stejné rozhraní mohou implementovat jinak nepříbuzné třídy

Stereotypy
....

* výše uvedené klíčové slovo <<interface>> je příklad tzv. stereotypu
 - stereotyp = rozšíření UML o nové prvky, které mají stejnou podobu jako prvky existující, ale odlišný účel
 - klíčové slovo stereotypu se píše nad jméno prvku, např. nad jméno třídy do francouzských uvozovek (= znaky ”guillemotleft” a ”guillemotright”), pokud tyto nejsou k dispozici, pak mezi dvojice znaků << a >>

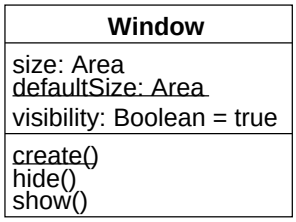
❇

KIV/ZSWI 2004/2005
Přednáška 5

Diagramy tříd (pokračování)

Poznámka (atribut patřící třídě)

* atributy a operace patřící třídě (nikoli instanci třídy) se znázorňují podtržením
  - odpovídají statickým atributům a metodám v jazycích C++, Java a C#.



[]

Poznámka (názvy asociací a názvy rolí)

* názvy rolí podstatná jména, např. zaměstnavatel, manager, zaměstnanec, adresa pro fakturaci apod.
  - před jméno role můžeme připojit indikátor viditelnosti, většinou se používá '+' (asociace je ve směru k roli viditelná)
* názvy asociací bývají slovesa, např. zaměstnává, řídí, pracuje pro apod.
  - asociace je třeba nazývat konkrétně a vyhýbat se příliš obecným názvům jako např. má, je součástí apod. (lepší je "učitel učí předmět" než "učitel má předmět")
  - neumíme-li asociaci rozumně pojmenovat, můžeme totéž vyjádřit přiměřeným jménem role (asociaci pak nemusíme pojmenovávat)
    . například místo "počítač má monitor" pojmenujeme roli monitoru - monitor je "zobrazovací zařízení"

* pokud jsou mezi stejnými třídami dvě asociace, znamená to, že objekt dané třídy může být pomocí každé z nich spojen s jiným objektem (instance asociace se nazývá "link" - uvidíme je např. v diagramech objektů)
  - pak musíme asociace rozlišit názvem asociace nebo názvem role

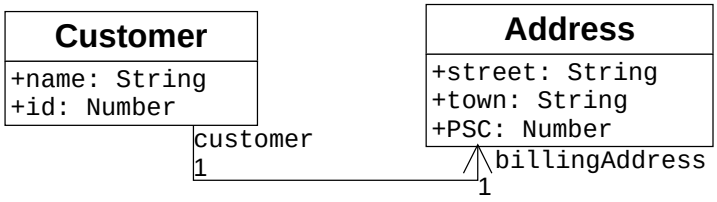
[]

Uspořádání

* pokud je násobnost větší než 1, pak množina prvků může být neuspořádaná nebo uspořádaná
  - není-li v diagramu uvedeno jinak, je množina neuspořádaná
  - pokud je uspořádaná, uvádíme to klíčovým slovem {ordered}

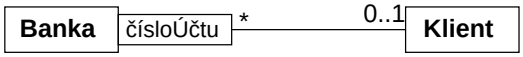
Průchodnost

* průchodnost (navigability) říká, že se pomocí asociace můžeme dostat z dané třídy do cíle
  - znázorňuje se šipkou směřující k cíli
  - pro větší přehlednost se v CASE nástrojích pro asociace s průchodností oběma směry často potlačuje zobrazení šipek (tj. šipky se zobrazují pouze pro asociace s průchodností jedním směrem)



Kvalifikované asociace
.....

* kvalifikátor je jeden nebo více atributů, jejichž hodnoty slouží pro určení množiny instancí, se kterou je objekt sdružen pomocí asociace
 - kvalifikátor musí rozdělovat instance do disjunktních množin
 - v UML se zobrazuje jako malý obdélníček připojený ke konci asociace, je umístěn u zdrojového konce asociace (kvalifikátory nepatří ke třídě, ale k asociaci)
  . atributy kvalifikátoru se znázorňují uvnitř obdélníčku (ve stejném tvaru jako atributy třídy)
  . násobnost umístěná u cílového konce asociace značí, jak velká bude množina cílových instancí (možné kardinality)

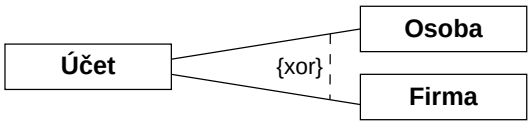


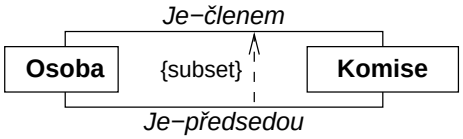
Pokud je na jednom konci asociace více symbolů (tzv. ozdob), např. označení pro kompozici i pro průchodnost, zobrazují se v tomto pořadí: navigační šipka, symbol agregace/kompozice, kvalifikátor.

Omezení
.....

* omezení (constraint) je sémantický vztah mezi prvky, který musí být splněn, aby byl model platný (tj. musí ho splňovat každá implementace modelu)
 - v UML můžeme specifikovat pomocí podmínky zapsané v {}
 - může se týkat libovolného elementu, např. třídy, asociace, atributu třídy
 - pokud se týká jednoho prvku, kreslí se u prvku
 - pokud se týká dvou prvků, přerušovaná šipka nebo čára spojující prvky

* například následující asociace má omezení {xor}, tj. bankovní účet může být buď osobní, nebo firemní





Poznámka (properties)

Pro zmatení veřejnosti se v UML stejným způsobem znázorňují i libovolné další vlastnosti, které nemají grafické vyjádření.

Například operace může mít vlastnost {query} značící, že operace nemění stav systému nebo {concurrency=cuncurrent} značící paralelní vykonávání. Konce asociace mohou mít vyznačeno např. {frozen}, tj. po vytvoření a inicializaci objektu u konce asociace nebude možné přidávat, rušit nebo měnit spojení tvořící tuto asociaci.

[]

Praktická poznámka (diagram tříd)

Při návrhu systému obvykle vytváříme více diagramů tříd, každý z nich zachycuje jeden aspekt statického pohledu na systém - obsahuje pouze prvky potřebné pro pochopení příslušného aspektu. Každý diagram by měl proto mít

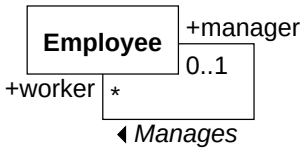
jméno, které vypovídá o jeho účelu.

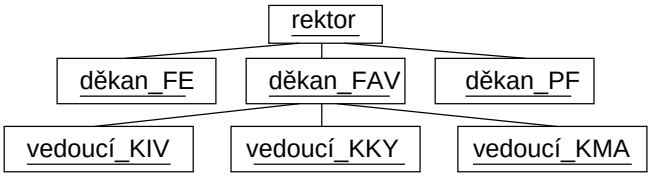
Úplný statický pohled na systém je tedy tvořen všemi diagramy tříd společně.

[]

Diagramy objektů
.....

* instance místo tříd, ukazuje stav systému v daném časovém okamžiku
* používá se poměrně zřídka, vhodné pro vysvětlení malých částí systému se složitými (zejména rekurzivními) vztahy





* instancí asociace je ”link” (propojení)
 - link je n-tice (nejčastěji dvojice) odkazů na objekty
 - znázorňuje se čarou (podobně jako asociace)

Poznámka pro zajímavost (UML a diagramy objektů)

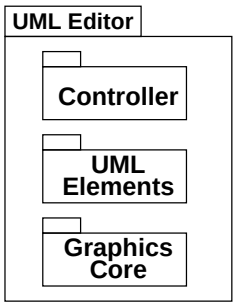
V diagramech tříd je dovoleno uvádět objekty a jejich vztahy (používá se především pro uvedení příkladů datových struktur, viz výše). Z hlediska UML je diagram objektů takový diagram tříd, ve kterém nejsou uvedeny žádné třídy, ale pouze instance (tj. v UML neexistuje samostatný typ diagramu ”diagram tříd”).

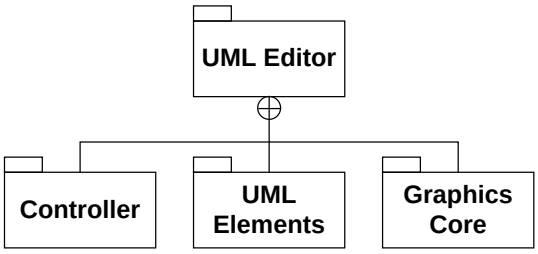
[]

Balíčky (packages)
.....

* slouží pro zjednodušení složitých diagramů
 - sdružují množinu libovolných prvků diagramu (uvnitř balíčků mohou být další balíčky)
 - každý prvek může být nejvýše v jednom balíčku

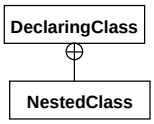
* zakreslují se jako obdélníky s malým držátkem na levé horní straně
 - prvky obsažené v balíčku lze kreslit buď do balíčku, nebo je možné nakreslit strom obsahu balíčku
 - viditelnost prvků vně balíčku je možné označit obvyklým způsobem (např. ’+’ pro ”public”)





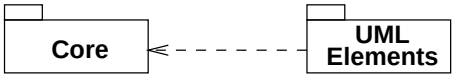
Poznámka pro zajímavost (označení vnořených podtříd)

V diagramu tříd se pro označení vnořených podtříd používá stejná notace jako pro označení stromu obsahu balíčku, tj. kroužek s křížkem:



[]

* závislosti mezi balíčky je možné znázornit přerušovanou čarou s otevřenou šipkou (A závisí na B jako šipku od A k B)
 - balíček A závisí na balíčku B, pokud změny v B mohou způsobit změny v A
 - notace pro závislost se používá i jinde, např. třída závisí na rozhraní



* speciálním druhem balíčku z hlediska UML je podsystém
 - podsystém úplně zapouzdřuje svůj obsah (tj. své chování zpřístupňuje pouze prostřednictvím rozhraní podsystému)
 - tj. dokud zůstane stejné rozhraní, obsah podsystému se může libovolně měnit
 - znázorňuje se jako balíček, v pravém horním rohu velkého obdélníka je ”vidlička”

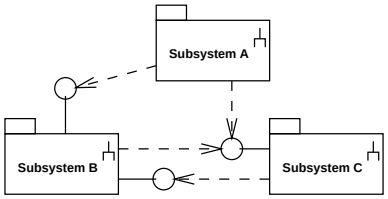
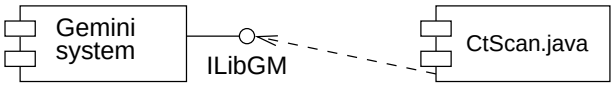


Diagram komponent
----

* diagram komponent je fyzická analogie diagramu tříd
 - komponenta zapouzdřuje implementaci a zveřejňuje množinu rozhraní
 - komponenta může být implementována např. jedním nebo více spustitelnými soubory, zdrojovými texty nebo objektovými moduly, knihovnami, příkazovými soubory apod.

* komponenta se znázorňuje jako obdélník, po jeho straně dva menší obdélníčky
 - prvky se znázorňují umístěné uvnitř komponenty
 - diagram komponent znázorňuje také závislosti komponent (přerušovaná šipka od závislého obvykle k rozhraní jiné komponenty)



Základní rozdíl mezi balíčkem a komponentou: balíček je čistě koncepční mechanismus pro organizaci modelů v UML, zatímco komponenty existují skutečně.

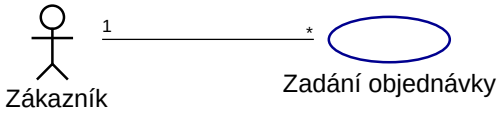
Nyní přejdeme od statického popisu systému k diagramům popisujícím chování systému nebo jeho částí. Nebudu samozřejmě popisovat všechny typy diagramů (na to je UML příliš rozsáhlé), ale popíšu nejdůležitější diagramy pro analýzu a návrh.

Diagramy případů použití
----

* popisují, co systém dělá z hlediska vnějšího pozorovatele (nikoli jak to dělá - k tomu slouží jiné typy diagramů, např. stavový diagram, viz dále)

* všechny diagramy případů použití obsahují

 - aktéry (nejčastěji znázorněni jako panáčci, ale používají se i jiné ikony)
  . aktér = cokoli co potřebuje komunikovat se systémem
  . představuje roli nebo množinu rolí uživatele při komunikaci s entitou
 - případy použití (znázorněny elipsou)
  . představují dialog nebo transakci vykonávanou systémem, podsystémem nebo třídou
 - asociace (čáry mezi aktéry a případy použití znázorňující komunikaci)
  . spojuje aktéry s případy použití
  . konce asociace mohou mít označení násobnosti

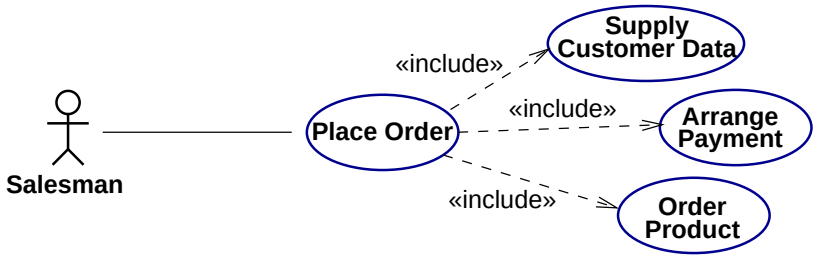


* případy použití mohou být spojeny se scénáři
 - scénář = příklad co se stane pokud někdo komunikuje se systémem
 - scénář je posloupnost kroků (v diagramu je nezobrazujeme)
 - popisuje se obyčejným textem, stavovým automatem apod.
  . např. bankomat - aktér vloží kartu, zadá PIN, zadá typ operace, zadá částku, bankomat předá peníze, předá částku, vrátí kartu

* diagram případů použití může dále obsahovat
 - zobecnění případů použití a aktérů
 - vztahy ”include” (zahrnout, vložit) a ”extend” (rozšířit)
 - balíčky sdružující prvky modelu do větších celků
 - poznámky

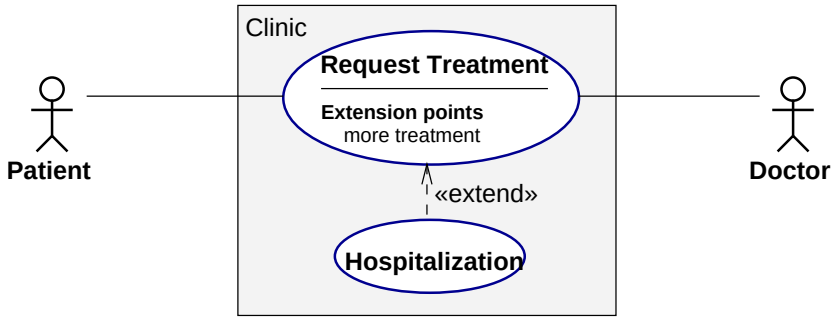
* zobecnění případů použití a aktérů (generalization)
 - ukazuje že jeden případ použití nebo aktér je zobecněním jiného (podobně jako rodičovská třída je zobecněním svých potomků)
 - znázorňuje se šipkou s trojúhelníkovou hlavou

* vztah ”zahrnout, vložit” (include)
 - rozložení případu použití na části, užitečné pokud stejnou část můžeme použít ve více případech použití (např. ”převod částky” může být součástí případu použití ”platba za službu” i ”platba za zboží”)
 - vkládaný případ nemůže existovat samostatně, je vždy součástí jiného
 - notace přerušovaná čára označená klíčovým slovem <<include>>



* vztah ”rozšířit” (extend)
 - označuje že bázový případ použití může být upraven podle obsahu jiného, tím vznikne varianta (rozšíření) bázového případu
 - používá se pokud jsou části případu použití volitelné, používají se zřídka apod.
 - notace přerušovaná čára označená <<extend>>, šipka k bázovému případu použití
  . v bázovém případu použití se uvedou ”body rozšíření”, tj. kdy se použije

rozšířený případ
. ”body rozšíření” mají nejčastěji podobu obyčejného textu



* někdy se v diagramu uvádí hranice systému = obdélník oddělující systém od externích aktérů, název systému se uvádí uvnitř obdélníku

* diagramy případů použití se používají:
 - pro sběr požadavků
 - pro komunikaci s klienty - jsou jim srozumitelné
 - pro generování testovacích případů - množina scénářů spojená s případem použití může být odrazovým můstkem pro vytvoření testovacích případů pro scénáře

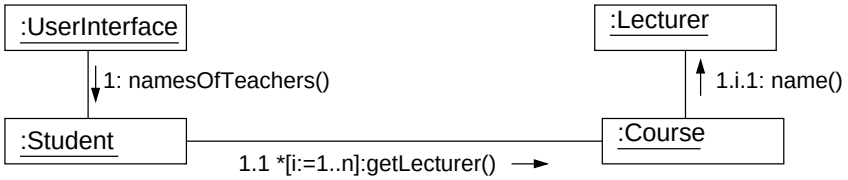
* nejběžnější modely:
 - model kontextu systému nebo podsystému, tj. určení co leží uvnitř systému (znázorňujeme ohraničením systému, viz výše), popis aktérů a jejich rolí
 - modelování požadavků - popis kontraktu mezi systémem a aktéry

Diagramy spolupráce

* diagramy spolupráce (collaboration diagrams) popisují spolupráci mezi objekty nebo jejich rolemi
 - mohou být připojeny např. k případu použití jako jeho popis
  . popisují které objekty nabízejí chování popisované případem použití
  . jak objekty případ použití vykonávají
 - mohou mít i vyšší úroveň podrobnosti, lze je použít např. pro popis chování operací třídy apod.

* v diagramu spolupráce se vyskytují
 - objekty účastnící se interakce, případně role objektů v interakci
 - spojení pro přenos zprávy - kreslí se jako plná čára
  . často se kreslí s šipkou ukazující průchodnost (v příkladech jí neuvádím)
 - zprávy (stimuly) - kreslí se jako šipečka blízko čáry
  . šipečka s plnou hlavou znamená volání procedury, volající pokračuje až po návratu z procedury

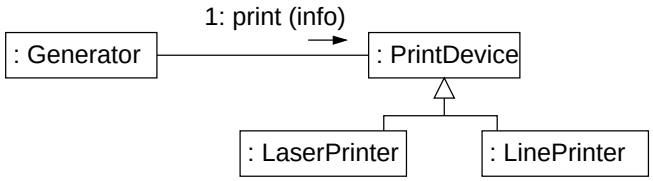
 - každá zpráva má pořadové číslo, končící dvojtečkou
  . zprávy na nejvyšší úrovni jsou číslovány postupně 1, 2, 3 atd.
  . zprávy na další úrovni vnoření během stejného volání 1.1, 1.2, 1.3 atd.
 - za pořadovým číslem může být uvedeno: zpráva(argumenty) nebo případně návratová_hodnota := zpráva(argumenty)
  . popisuje zaslanou zprávu, její argumenty a návratovou hodnotu



* v diagramu spolupráce lze znázornit také iterace a podmínky
 - iterace - jako pořadí zprávy uvedeme hvězdičku (pokud nechceme uvádět

podrobnosti) nebo např. *[i := 1..n] (pokud chceme iteraci popsat)
- podmíněná zpráva - jako prefix uvedeme podmínku. např. [x>0]
. co má být uvnitř hranatých závorek UML nespecifikuje; obvykle se používá pseudokód nebo syntaxe cílového programovacího jazyka
. například: 4[x<0]: display(x)
- zprávy je možné také "číslovat" identifikátorem

* obdélníčky v diagramech spolupráce jsou buď objekty (název je podtržený) nebo role (název není podtržený)
- plný zápis objektu "objekt/role : Třída", plný zápis role "/role : Třída"
* lze kombinovat se vztahy z diagramu tříd, např. zobecnění, agregace apod.



Pomocí diagramů spolupráce se obvykle znázorňují jednoduché posloupnosti zpráv. Pro složitější chování se obvykle používají (významově ekvivalentní) sekvenční diagramy, ve kterých se lépe zobrazují časové závislosti.

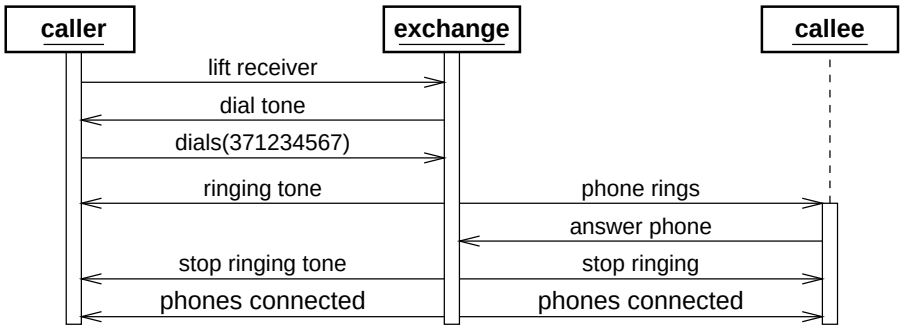
Sekvenční diagramy

* sekvenční diagramy (sequence diagrams)
- popisují stejnou informaci jako diagramy spolupráce, ale soustředí se na časové závislosti
- v některých metodikách (např. OOSE) se k případům použití vytvářejí sekvenční diagramy místo diagramů spolupráce

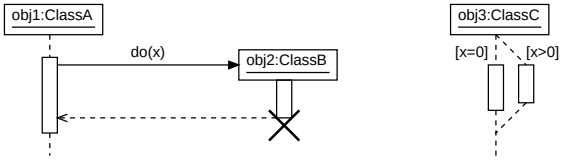
* sekvenční diagramy znázorňují aktéry, objekty v systému, se kterými interagují, a posloupnost vyměněných zpráv
- čas plyne shora dolů, pro každý objekt svislá přerušovaná "čára života" (lifeline) reprezentující čas, kdy objekt existuje a hraje určitou roli, doba aktivity objektu se znázorňuje úzkým obdélníkem
- horizontální uspořádání není podstatné a má být zvoleno s ohledem na srozumitelnost diagramu
- vlevo od diagramu sloupec popisující interakci s okolním světem ("hranice systému")

* šipky podle typu komunikace:
- volání procedury nebo jiný vnořený tok řízení (tj. vnější sekvence může pokračovat až po dokončení vnitřní sekvence) - plná šipka, viz (a)
- asynchronní komunikace - šipka tvořená čarami, viz (b)
- návrat z procedury - přerušovaná šipka, viz (c)
. pokud řízení toku procedurální (viz "návrh mechanismu řízení"), můžeme vynechat šipku pro návrat z procedury
- stejné tři typy šipek lze použít i v diagramech spolupráce
- horizontální šipka = atomické zaslání zprávy
- šipka směřující šikmo dolů = během zaslání zprávy může nastat další událost, např. zaslání zprávy opačným směrem
- vlevo od diagramu bývá posloupnost akcí okomentována např. pseudokódem

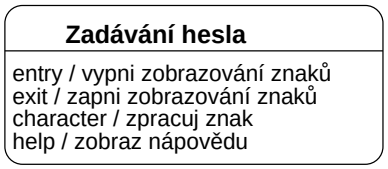
* příklad - dva telefonní účastníci a ústředna:



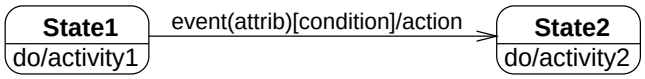

```
* další možnosti (uvádím pouze pro zajímavost)
 - zpráva může zapříčinit vznik nebo zánik objektu (šipka směřuje k symbolu
   objektu resp. symbolu zániku objektu X); objekt může provést autodestrukci
 - neexistuje-li objekt po celou dobu pokrytou diagramem, čára života začíná
   a končí na místě vzniku a zániku objektu (objekt se kreslí na její začátek)
 - čára života se může rozdělit na dvě pro znázornění podmínky (podmínka se
   uvádí v hranatých závorkách), čáry se později mohou opět spojit
 - podmínkou se mohou označovat i zprávy
```



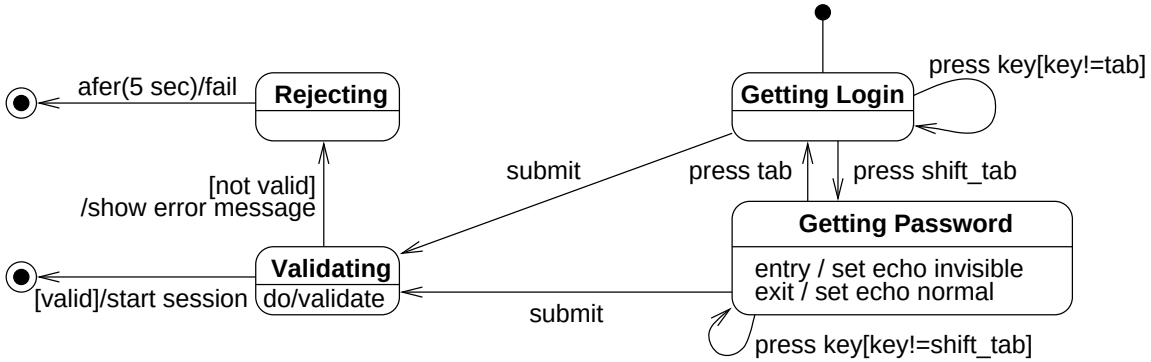

```
Zatímco sekvenční diagramy se používají pro vyjádření časových závislostí,
diagramy spolupráce slouží spíše pro znázornění struktury systému a vztahu
mezi instancemi.


Stavové diagramy
----------------

* typicky se používají pro popis chování instance třídy, někdy také pro
  případy použití nebo pro celý systém nebo podsystém
 - popisují možné posloupnosti stavů, kterými objekt prochází v důsledku
   reakcí na události (událostí může být např. vyvolání operace, uplynutí
   času apod.)
 - oproti klasickým "plochým" stavovým diagramům umožňují stavové diagramy v
   UML strukturování (které nebudu příliš podrobně popisovat, ale pro
   rozsáhlejší diagramy je nutné)

* stav = situace během života objektu, kdy objekt splňuje nějakou podmínku,
  provádí nějakou akci nebo čeká na událost
* událost = výskyt stimulu, který může spustit přechod do jiného stavu
* přechod = změna stavu způsobená událostí; nový stav závisí na původním
  stavu a na události

* stavový diagram je graf, kde
 - uzly grafu = stavy, znázorněny jako obdélníky s kulatými rohy, uvnitř
   volitelně část s názvem stavu a s posloupností akcí
  . entry/akce - akce prováděná při vstupu do stavu
  . do/akce    - akce prováděná během stavu
  . exit/akce  - akce prováděná při opuštění stavu
  . mohou být uvedeny další uživatelem definované akce
```



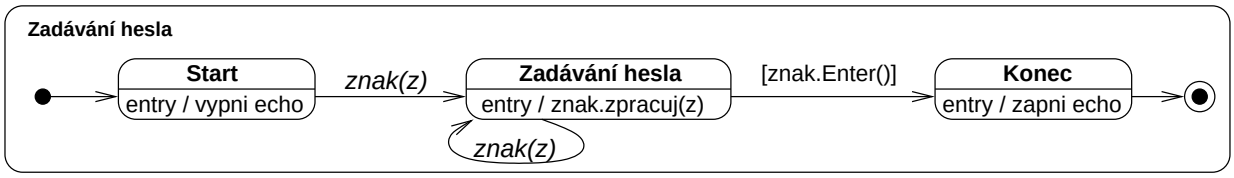
- orientované hrany grafu = přechody mezi stavy
  . popis hrany = událost, která způsobí přechod a akce provedené jako
    důsledek přechodu ve tvaru: událost/akce, případně událost[podmínka]/akce
  . událost má tvar: jméno_události(parametry)



- počáteční stav - znázorněn jako černé kolečko
- koncový stav - znázorněn jako černé kolečko v kroužku (”býčí oko”)



* činnost v daném stavu lze popsat opět pomocí stavového automatu - vnoření
  - počáteční a koncový pseudostav je znázorněn stejně jako počáteční a
    koncový stav



Poznámka (volně šířený nástroj Fujaba)

Stavové diagramy lze vytvářet i ve volně šířeném nástroji Fujaba, který z
nich umí vygenerovat kód v jazyku Java. To se může hodit např. pro
implementaci síťových protokolů apod.

[]

❇

KIV/ZSWI 2004/2005
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.

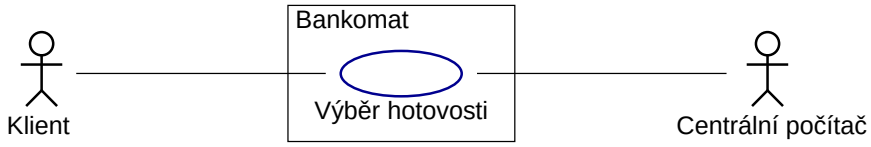
Eliminujeme vágní třídy: software, zabezpečení.

[]

* 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:



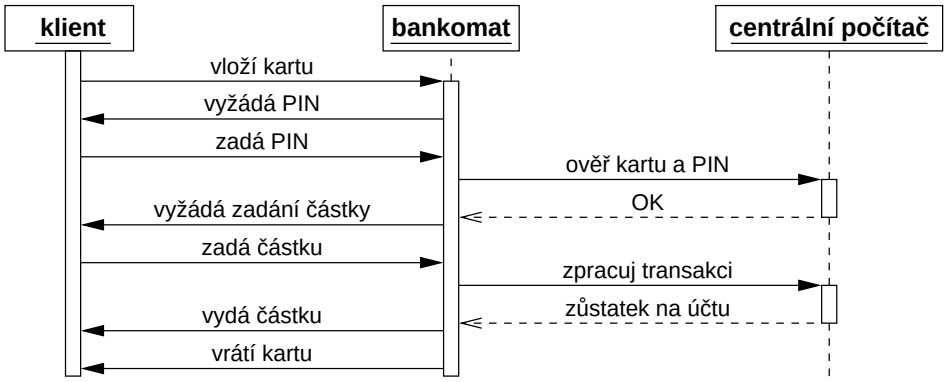
Případ použití: Výběr peněz z bankomatu.
Aktéři: Klient, Centrální počítač
Stručný popis: Zákazník vloží kartu a požádá o výběr určité částky. Bankomat mu po potvrzení centrálním počítačem požadovanou částku vydá.

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
```

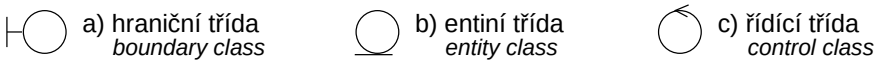
Příklad (sekvenční diagram pro případ použití ”Výběr peněz z bankomatu”)



[]

* 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 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>>

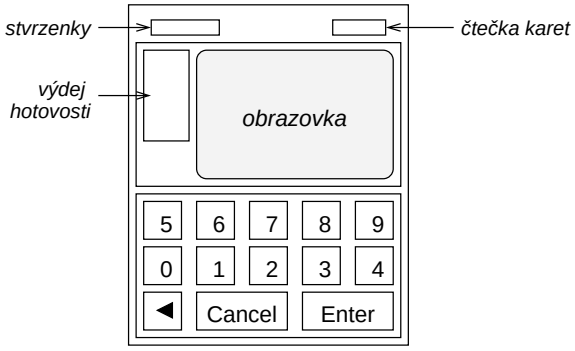


Příklad (opět bankomat):

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.

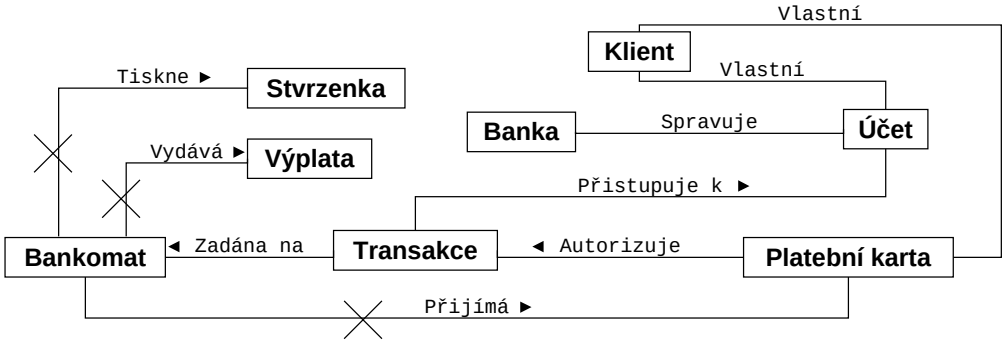
[]

* 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

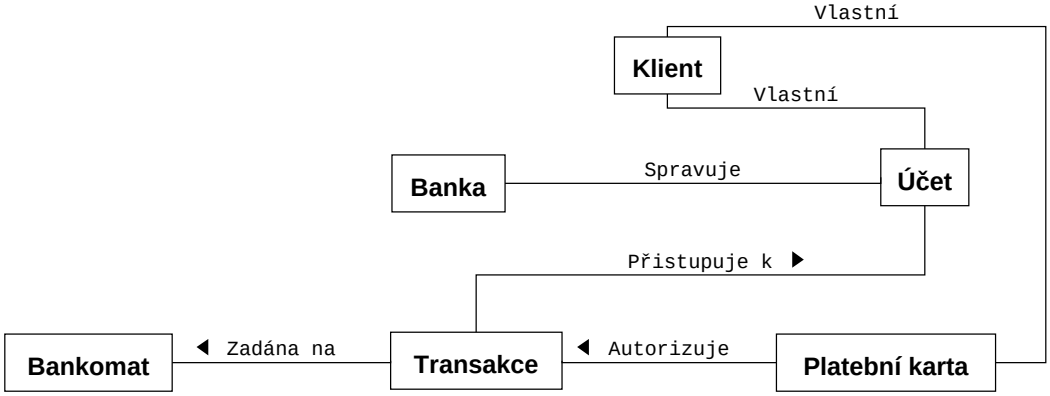
* vytváříme předběžné asociace mezi třídami
 - představují vztah mezi instancemi, který nějakou dobu přetrvává (např. Osoba pracuje pro Firmu); objekty o sobě navzájem ”vědí”
 - asociace mohou odpovídat fyzickému umístění nebo vztahu vlastnictví (má spojení s, je součástí, je obsažen v, patří), řízení 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:

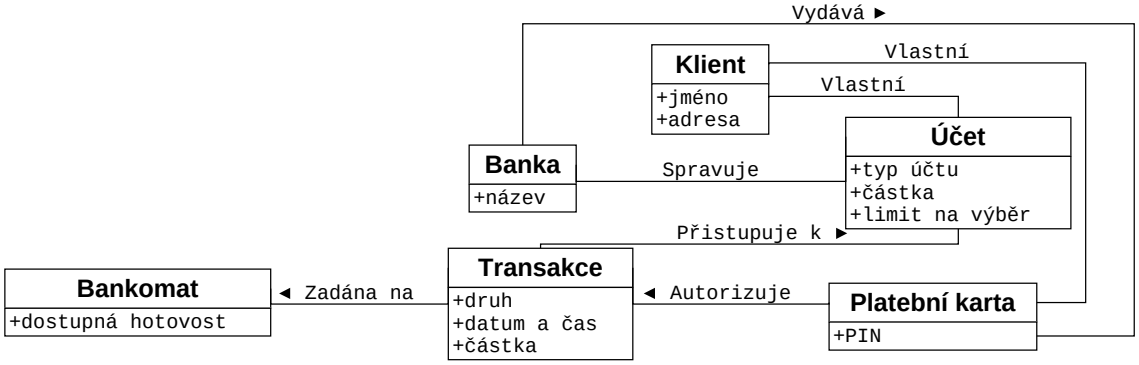


* zrušíme nepotřebné nebo nesprávné asociace
 - v této fázi zrušíme asociace, které nejsou podstatné pro řešený problém, jsou redundantní nebo které představují popis implementace
  . například asociace ”je prarodičem” lze popsat pomocí dvou asociací ”je rodič” (někdy ale můžou být i odvozené asociace 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”

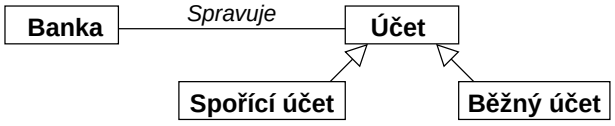


* vytváříme agregace a kompozice
 - agregace, pokud objekt je součástí jiného nebo objekt je podřízený jinému, např. pokud se některé operace nad celkem provedou nad všemi částmi
 - kompozice, pokud je silné vlastnění (strom součástí, součásti nelze sdílet) a stejná doba života

* 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

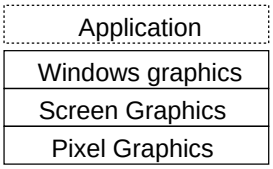
Rozdělení do vrstev
....

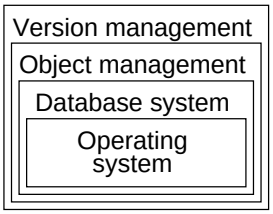
* vrstvené systémy = uspořádaná množina virtuálních světů
 - každý svět je vystavěn z prvků nižšího světa a poskytuje stavební prvky vyššímu světu
 - mezi vrstvami vztah klient-poskytovatel - nižší vrstvy (poskytovatelé) poskytují služby vyšším vrstvám (klientům)
 - znalost je jednosměrná, tj. podsystém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

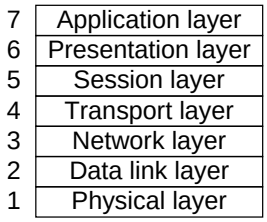
* objekty ve stejné vrstvě mohou být nezávislé, ale mezi objekty v různých vrstvách obvykle existuje korespondence (např. poskytují obdobné služby na různých úrovních abstrakce)

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
```

[]

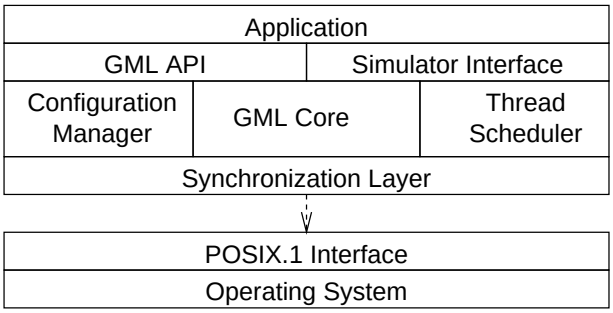
* není-li prostředí přenositelné, považuje se za vhodné vytvořit alespoň
  jednu vrstvu mezi aplikací a službami poskytovanými OS nebo HW (vrstva
  poskytuje logické služby a mapuje je na fyzické)
 - přepsáním této vrstvy můžeme systém přenést na jinou platformu

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, měli 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. 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

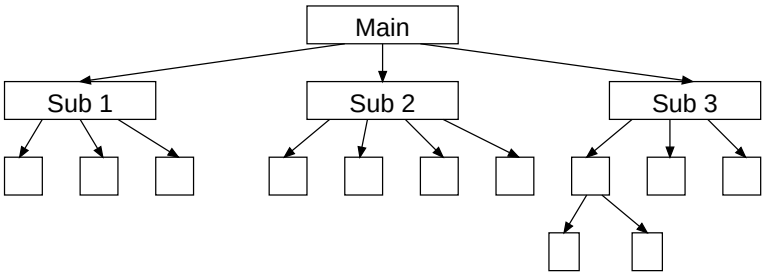
* databáze - silnější prostředek
 - společné rozhraní pro množinu aplikací pomocí standardního přístupového jazyka (SQL)
 - výhodné pro data, ke kterým bude přistupovat více uživatelů, případně více programů, a pro data, která mohou být efektivně spravována příkazy DB jazyka

- 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, sekvenční systém řízený událostmi, 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)

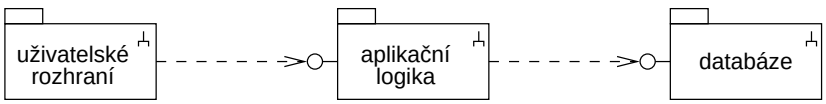


* systémy řízené událostmi
  - běh systému řídí dispečer poskytovaný podsystémem, programovacím jazykem nebo OS
  - s jednotlivými událostmi jsou svázány procedury aplikace
  - systém někdy poskytuje frontu událostí, nově příchozí události se řadí do fronty, ze které dispečer vybere následující událost 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.

❇

KIV/ZSWI 2004/2005
Přednáška 7

V klasickém životním cyklu vývoje SW pod pojem ”návrh” spadají dvě aktivity, které jsou umístěny mezi analýzou požadavků a kódováním programu:

* návrh architektury systému - rozděluje systém do podsystémů či komponent; o návrhu architektury jsme mluvili na minulé přednášce
* podrobný návrh systému - každá součást systému je popsána natolik podrobně, aby to postačovalo pro kódování

V dalším textu se budeme věnovat podrobnému objektově orientovanému návrhu a jeho implementaci.

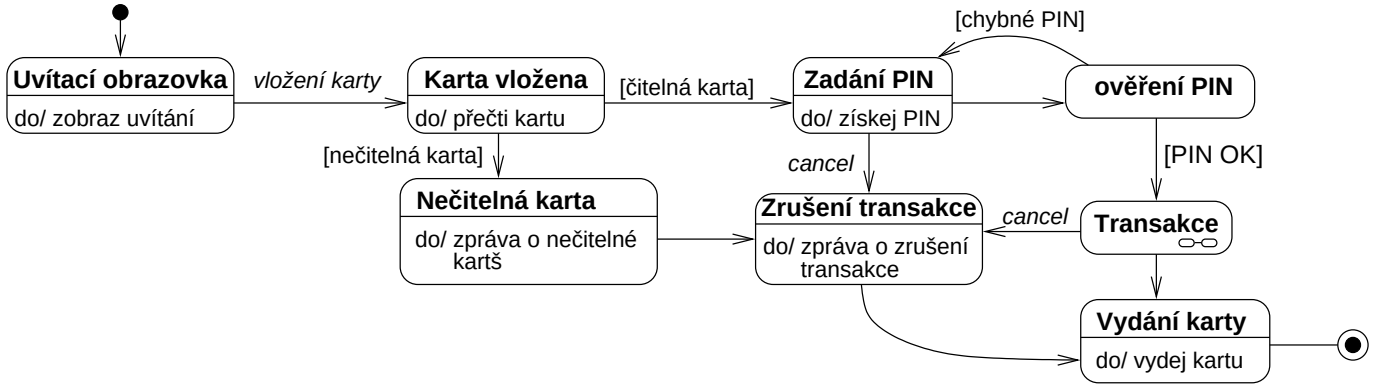
Objektově orientovaný návrh

Přehled:

* vstupem jsou analytické třídy představující role, které mohou být pokryty jednou nebo více třídami návrhu; analytická třída se v návrhu může stát jednou třídou, částí třídy, agregovanou třídou, skupinou spřízněných tříd, asociací, naopak asociace třídou apod.
* vytvoření návrhových tříd (design classes)
* definice operací
* definice asociací, agregací a kompozic

* pro analytické třídy budeme vytvářet jednu nebo více návrhových tříd
  - návrh hraničních tříd
    . pokud jsou k dispozici nástroje pro návrh GUI, pak jedna hraniční třída bude odpovídat jednomu oknu nebo formuláři
    . jednu třídu pro každé API nebo protokol
  - entitní (datové) třídy
    . jsou často pasivní a perzistentní, musíme pro ně vybrat způsob implementace: v souboru, v relační databázi (z perzistentních tříd vytvoříme tabulky atd.); nejsou-li třídy perzistentní, pak budou implementovány v paměti
  - řídící třídy
    . obsahují aplikační logiku

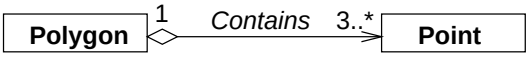
* vyplňování tříd 1: definice operací
  - standardní konstruktory se předpokládají => z návrhu je vynecháváme
  - operace, které čtou a nastavují veřejné atributy, se (většinou) předpokládají
  - hledáme operace - prvním vodítkem může být již vytvořený seznam sloves
  - další způsob - získat operace z popisu interakce objektů
    . nakreslíme diagramy spolupráce nebo sekvenční diagramy (resp. upravíme diagramy získané ve fázi analýzy, doplníme zasílané zprávy)
    . z nich zjistíme, jaké stimuly musí být objekt schopen přijmout, navrhneme odpovídající operace
  - další možnosti, co může být operace:
    . inicializace nově vytvořené instance včetně propojení s asociovanými objekty
    . případné vytvoření kopie instance (pokud je zapotřebí)
    . případný test na ekvivalenci instancí (pokud je zapotřebí)
  - operace popíšeme: název, parametry, návratová hodnota, krátký popis, viditelnost
  - pokud je algoritmus složitý, popíšeme metodu (= implementaci operace) nebo nakreslíme stavový diagram objektu nebo operace; např. stavový diagram pro bankomat:



- základní kritérium: každá metoda by měla dělat jednu věc dobře

* vyplňování tříd 2: definice atributů
 - při hledání atributů můžeme vycházet z logických atributů (produkt analýzy) popisujících, co je potřebné pro uchování stavu objektu
 - další možnost - jaké atributy jsou třeba pro implementaci operací
 - atributy v návrhu mají být ”jednoduché” (int, boolean, float, ...) nebo mají mít sémantiku hodnoty (tj. být nezměnitelné, např. String)
  . jinak použijeme asociaci
 - atributy popíšeme: jméno, typ, počáteční hodnota, viditelnost
  . měli bychom se snažit o skrývání informací - soukromé atributy (viditelné pouze pro potomky = protected ”#”)
 - ověříme, že všechny atributy jsou potřebné
 - pokud zjistíme, že se atributy a operace dělí do dvou tříd, které spolu příliš nesouvisejí, pak se pravděpodobně jedná o dvě různé třídy - měli bychom třídu rozdělit

* definice asociací, agregací a kompozic
 - stejně jako v analýze, máme ale podrobnější informace o chování
 - navíc můžeme přidat tzv. průchodnost (navigability) - od kterého ke kterému objektu budeme chtít snadno přecházet
  . označuje se šipkou v asociačním vztahu



* pokud třída obsahuje více než cca 10 atributů, 10 asociací nebo 20 operací, pak asi není dobře navržená a potřebuje rozdělit

* definice zobecnění (hierarchie dědičnosti)
 - stejně jako v analýze, tj. společné vlastnosti vyjmeme do nadtříd
 - pokud má některá třída sjednocení atributů svých podtříd (místo průniku), zřejmě spolu podtřídy ve skutečnosti nemají nic společného
 - hierarchie by měla být vyvážená, tj. neměly by být třídy, pro které je hierarchie neobvykle plochá nebo naopak hluboká

* kontrola modelu
 - ověřit realizace případů použití, v návrhu nám nesmí chybět chování potřebné pro některý případ použití

* volitelně vytváření balíčků
 - buď pro lepší srozumitelnost (strukturování), nebo pro izolaci částí, které se budou častěji měnit (případně obojí)
 - do balíčku vkládáme funkčně příbuzné třídy; např. pokud by změna chování jedné třídy způsobila změnu chování druhé, jsou třídy funkčně příbuzné
 - třídy uvnitř balíčku mohou být veřejné (public) nebo soukromé (private)
  . veřejná třída může mít asociace s libovolnou jinou třídou; veřejné třídy tvoří rozhraní balíčku
  . soukromá třída může být asociována pouze se třídami uvnitř stejného balíčku
 - pokud má třída v jednom balíčku asociaci se třídou v jiném balíčku, pak balíčky na sobě navzájem závisejí - modeluje se vztahem závislosti (přerušovaná šipka)

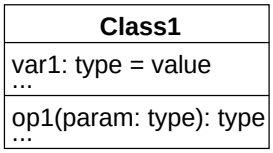
- v jednom diagramu můžeme zakreslit i závislosti mezi třídami apod.

* tím končí návrh a můžeme přejít k implementaci

Objektově orientovaná implementace

Poznámka: Obrázky v této části nejsou platné UML, ale snaží se naznačit způsob implementace návrhu.

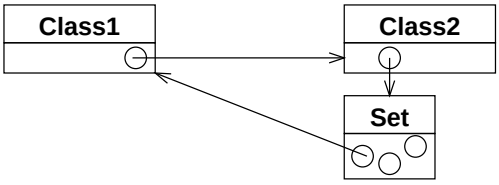
Kostru implementace v objektově orientovaném programovacím jazyce můžeme snadno vytvořit z diagramu tříd:



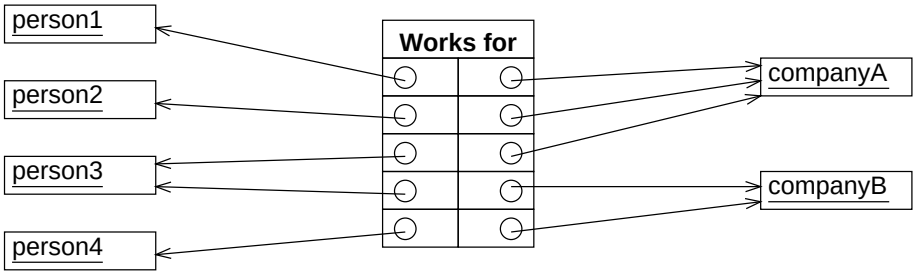

```
public class Class1 {
    // konstruktory
    public Class1() {};
    public Class1(type param) { setVar1(param) };

    // veřejné operace get a set, soukromá proměnná
    private type  var1 = value;
    public type getVar1()           { return var1; }
    public void setVar1(type param) { var1 = param; }

    // operace pro každou metodu
    public type op1(type param) { ... }
}
```

* převod asociací - asociace nejsou přímo podporovány OO programovacími jazyky
 - pokud je násobnost 1 a průchodnost jen jedním směrem (tj. jedním směrem přístup podstatně častěji), implementujeme asociaci jako odkaz na objekt (referenci objektu, v některých jazycích jako ukazatel na objekt)

 - pokud je násobnost 1 a průchodnost oběma směry - implementujeme jako vzájemné odkazy
 - pokud je násobnost 1:M - množinu odkazů udržujeme např. ve stromu apod.



 - pokud je k asociaci přidán nebo z ní odebrán link, je třeba upravit oba odkazy
 - pokud jsou změny asociací častější než vyhledávání, je vhodná implementace pomocí asociačního objektu (asociačního objekt implementuje např. hashovanou tabulku, prvek tabulky obsahuje dvojici (ObjA, ObjB))



- asociace M:N nemohou být přiřazeny jako atribut ani jednomu konci, tj. pro ně musíme také vytvořit asociační třídu
- ternární a vyšší asociace buď rozložíme na binární (pokud to jde), nebo opět vytvoříme asociační třídu (N-ární asociace naštěstí nejsou v praxi časté)

* převod agregátů a kompozic
 - agregáty implementujeme stejně jako asociace, označení v diagramu je pouze ”komentář” ukazující, že vztah je bližší než u ostatních asociací
 - kompozice = existenční závislost součástí
  . v destruktoru zrušíme součásti
  . pokud mají součásti stejnou dobu života, vytvoříme je v konstruktoru

* dědičnost se na konstrukce OO programovacího jazyka mapuje přímočaře, například v jazyku Java:

```
public class Podtřída extends Nadtřída {
  ...
}
```

Implementace objektového návrhu v neobjektovém jazyce

Výše uvedené koncepce lze implementovat i v neobjektovém jazyce, jako je např. C nebo Pascal - implementace ovšem nebude tak přímočará.

Níže uvedeným způsobem byly implementovány některé SW systémy před příchodem spolehlivých překladačů jazyka C++.

Implementace probíhá v následujících krocích:

* překlad tříd do datových struktur
* vytvoření konstruktorů
* implementace metod

Překlad tříd do datových struktur

* třídu obvykle implementujeme jako blok atributů - typ ”struct” v jazyku C, typ ”record” v Pascalu, například:

```
struct Čára {
    int x1, y1;
    int x2, y2;
};
```

 - proměnná reprezentující objekt musí být reprezentována ukazatelem na záznam, aby odkaz bylo možné předávat a sdílet, např.:

```
struct Čára *čára;
```

Vytvoření konstruktorů

* objekty mohou být alokovány staticky nebo dynamicky (na hromadě)
 - staticky - pro globální objekty, pokud předem víme, jaký počet objektů daného typu bude zapotřebí
 - dynamicky (pomocí ”malloc()” v C nebo ”new” v Pascalu) - pokud už není

zapotřebí, musí být explicitně dealokován (”free()” v C, dispose v Pascalu)

```
struct Čára *create_čára(int x1, int y1, int x2, int y2)
{
    struct Čára *čára;

    čára = (struct Čára*) malloc(sizeof(struct Čára));
    čára->x1 = x1;
    čára->y1 = y1;
    čára->x2 = x2;
    čára->y2 = y2;

    return čára;
}
```

Metody
......

* pro pojmenování metod se obvykle používá konvence, která popisuje, které třídě metoda patří (jméno třídy + dvě podtržítka + jméno metody)
* každé metodě je třeba předat odkaz na objekt, nad kterým je operace prováděna (”self” nebo ”this” v OO jazycích); konvence je předávat ”self” jako první argument

```
void Čára__posun(struct Čára *self, int posun_x, int posun_y)
{
    self->x1 += posun_x;
    self->y1 += posun_y;
    self->x2 += posun_x;
    self->y2 += posun_y;
}
```

Dědičnost
....

V neobjektovém jazyce je nejlepší se dědičnosti vyhnout, tj. přestrukturovat diagram tříd nebo jeho části tak, aby dědičnost nebyla zapotřebí. Abychom se dědičnosti vyhnuli, můžeme také použít následující dvě možnosti:

1. ”zděděnou” operaci implementujeme znovu jako samostatnou metodu
   - v tomto případě ovšem dochází k duplikaci kódu
2. místo dědění z nadtřídy jsou ”děděné” atributy a operace implementovány samostatným objektem
   - podtřída bude obsahovat odkaz na tento nový objekt a všechny ”děděné” operace deleguje tomuto novému objektu

Poznámka pro zajímavost (implementace dědičnosti v neobjektovém jazyce)

Pokud musíme implementovat dědičnost a potřebujeme polymorfismus, pak se třídy implementují pomocí dvou záznamů:

* první záznam obsahuje atributy objektu (podobně jako ve výše uvedeném příkladu)
  - jako první položku navíc obsahuje ukazatel na druhý záznam, obsahující odkazy na metody třídy (tento druhý záznam je sdílen všemi instancemi třídy)
  - počáteční položky prvního záznamu jsou v potomkovi stejného typu a ve stejném pořadí jako v nadtřídě (to umožňuje, aby nad nimi pracovaly i metody nadtřídy)

```
struct Shape {  /* abstraktní třída Shape (česky ”tvar”) */
    struct ShapeClass *class;  /* odkazy na metody třídy */
    int x, y;                  /* umístění tvaru */
};

struct Line { /* konkrétní třída Line (česky ”čára”) */
    struct LineClass *class;   /* odkaz na metody třídy */
    int x, y;                  /* opakuje atributy z abstraktní třídy */
```

```
            int x2, y2;               /* nové atributy */
        };
```

* pro zjištění skutečných metod máme druhou datovou strukturu, obsahující ukazatele na metody dané třídy:

```
        struct ShapeClass {
            void (*move)(int x, int y);  /* přesun objektu */
            void (*scale)(int newsize);  /* změna velikosti objektu */
        };

        struct LineClass {
            void (*move)(int x, int y);  /* přesun objektu */
            void (*scale)(int newsize);  /* změna velikosti objektu */
            void (*setColor)(int color); /* nová metoda - změna barvy objektu */
        };
```

- vytvoříme a inicializujeme proměnnou, obsahující odkazy na metody:

```
        struct LineClass LineClass =
        {
            Shape__move,   /* dědíme po rodičovi */
            Line__scale,   /* překrytí metody vlastní metodou */
            Line__setColor /* nová metoda */
        };
```

- metodu bychom pak mohli vyvolat následovně:

```
         line->class->scale(5); /* vyvolání metody */
```

* konstruktor třídy ”Line” by mohl vypadat takto:

```
        struct Line *create_line(int x1, int y1, int x2, int y2)
        {
            struct Line *line;

            line = malloc(sizeof(struct Line));
            line->class = &LineClass;
            line->x = x1;
            line->y = y1;
            line->x2 = x2;
            line->y2 = y2;

            return line;
        }
```

* použití:

```
         struct Line *line = create_line(1, 1, 2, 2); /* vyvolání konstruktoru */
```

[]

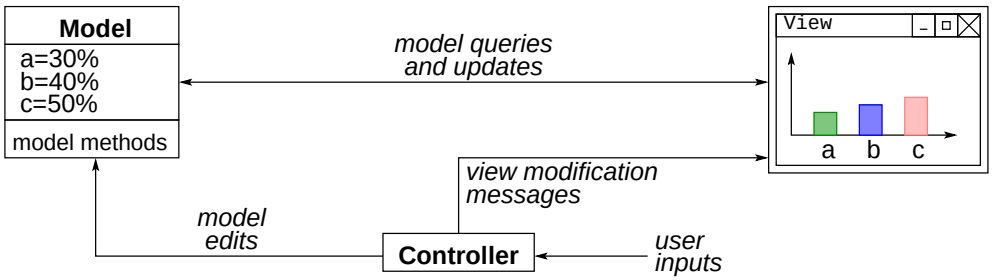
Návrhové vzory

* ”vzory” (patterns) jsou řešení problémů, které se při vytváření SW systémů často opakují
* vzory existují na různých úrovních:
  - návrhové vzory - dokumentují řešení problémů na úrovni návrhu
  - analytické vzory - dokumentují řešení doménových problémů (o nich zde nebudeme z časových důvodů mluvit)
  - architektonické vzory - řešení problémů vysokoúrovňového návrhu

V následujícím textu uvedu několik často používaných návrhových vzorů.

Termín návrhové vzory (design patterns) byl zaveden stejnojmennou knihou, která byla přeložena také do češtiny:

```
  Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns.
  Elements of Reusable Object-Oriented Software.
  Addison-Wesley, 1995.
  (český překlad: Návrh programů pomocí vzorů. Grada 2003.)

Co je návrhový vzor
...................

Jako příklad pro ilustraci uvedu trojici tříd nazývaných
Model/View/Controller (MVC) pro návrh interaktivních aplikací.

* myšlenka pochází z prostředí jazyka Smalltalk (Goldbergová a Robson 1983)
* funkčnost aplikace rozdělíme na 3 typy objektů, mezi objekty je vztah
  vydavatel-předplatitel (publisher-subscriber)
 - "model" je objekt aplikace, zapouzdřuje data, která mají být zobrazena
 - "view" (česky "náhled") je prezentace modelu na obrazovce
  . ve chvíli, kdy dojde ke změně stavu modelu, oznámí to model všem "view",
    které na něm závisejí; view zjistí od modelu nové hodnoty a znázorní je
  . způsobů prezentace může být více, je možné ho změnit např. na koláčový
    graf nebo tabulku, aniž by to ovlivnilo model nebo controller
 - "controller" definuje způsob, jak uživatelské rozhraní reaguje na vstup
  . způsob reakce na uživatelský vstup je možné změnit, aniž by to ovlivnilo
    model nebo view; například místo klávesových zkratek použijeme výběr z
    menu
```




```
* v tomto příkladu je vidět několik dalších návrhových vzorů:
 - návrhový vzor vydavatel-předplatitel (angl. publisher-subscriber nebo
   subject-observer)
  . vydavatel = instance, v níž může událost vzniknout
  . předplatitel = instance, která má zájem reagovat na určitou událost
  . předplatitel si u vydavatele zaregistruje metodu, která má být v důsledku
    události vyvolána
  . předplatitelů může být i více
  . nastane-li událost, vydavatel vyvolá registrované metody předplatitelů
 - vztah mezi "view" a "controller" je příklad návrhového vzoru "stategie"
   atd.


Iterátor
........

* jeden z nejjednodušších a nejčastěji používaných návrhových vzorů je
  iterátor
* problém: agregovaný objekt (jako je např. seznam) by měl umožňovat
  procházení, aniž by k tomu uživatel musel znát vnitřní strukturu agregátu
  (která se může změnit)
 - iterátor udržuje informaci o aktuálním objektu v agregátu, umí se posunout
   na následující prvek agregátu a prvek poskytnout uživateli
 - moderní OO programovací jazyky obsahují podporu iterátorů; např. v jazyce
   Java se používá rozhraní Iterator:

     public interface Iterator
     {
         boolean hasNext(); // Vrací "true" pokud jsou další prvky.
         Object next();     // Vrací další prvek agregátu.
     }
```

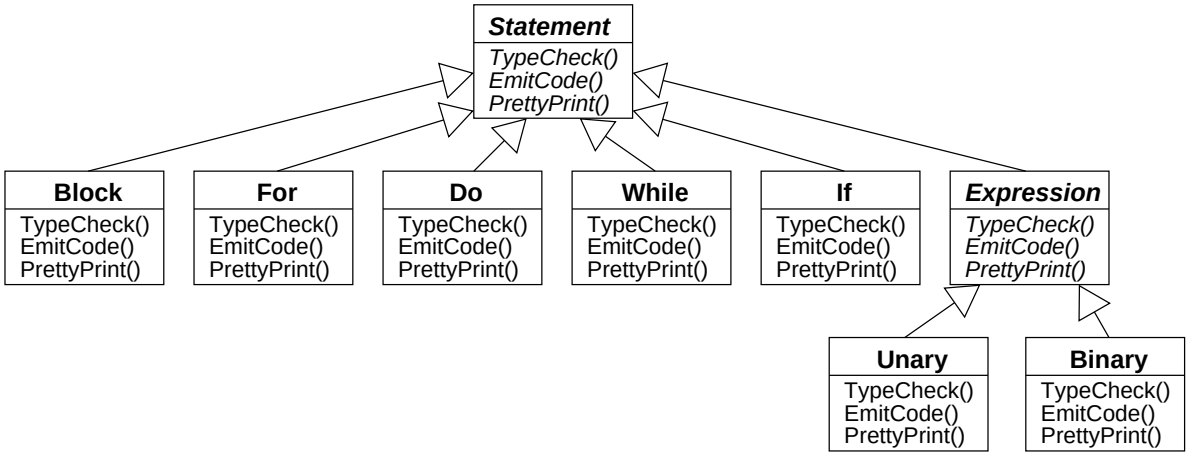
- v Javě poskytují iterátor třídy Set, List, Map, atd.; lze psát např.

```
for (Iterator e = v.iterator(); e.hasNext();) {
    System.out.println(e.next());
}
```

Návštěvník
.....

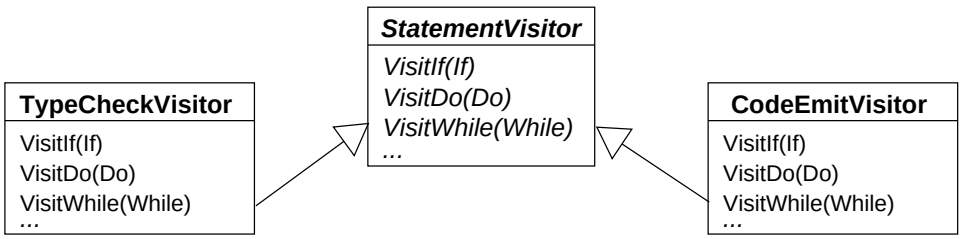
* představme si program, který vytváří strukturu objektů, nad kterými se provádějí další operace
 - například překladač vytváří syntaktický strom odpovídající zdrojovému programu
 - nebo formátovací program vytváří strom objektů odpovídající struktuře vstupního textu

* nad strukturou objektů budeme chtít provádět různé operace
 - například překladač - kontrola syntaktické správnosti, optimalizace, generování kódu, měření vlastností zdrojového textu apod.



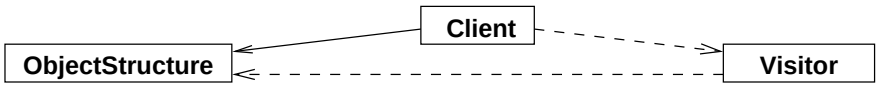
 - pokud potřebujeme nad objekty struktury provádět mnoho různých operací, bylo by lepší, kdybychom nové operace mohli přidávat samostatně a kdyby původní strom objektů byl nezávislý na operacích, které nad ním budou prováděny (abychom pro každý typ prvku nemuseli implementovat všechny operace - obtížnější udržovatelnost)

* řešení - příbuzné operace každé třídy zabalíme do samostatného objektu, nazývaného návštěvník (visitor)
 - při průchodu stromem objektů předáváme návštěvníka jednotlivým prvkům
 - navštívený prvek pošle návštěvníkovi požadavek, v názvu volané metody je zakódována třída prvku, prvek předá sám sebe jako argument
 - například TypeCheckVisitor bude při procházení stromu objektů volat jejich metody Accept()
  . metoda Accept() vyvolá odpovídající metodu návštěvníka, např. objekt třídy If vyvolá VisitIf, objekt třídy While vyvolá VisitWhile apod.
  . návštěvník může při průchodu strukturou akumulovat stav, čímž eliminujeme potřebu globálních proměnných



 - jinými slovy - definujete dvě hierarchie tříd, jednu pro objekty, nad kterými se provádějí operace (If, While...) a jednu pro návštěvníky, kteří provádějí operace nad objekty (TypeCheckVisitor, CodeEmitVisitor,

PrettyPrintVisitor...)
- kromě návštěvníka a struktury objektů hraje v systému často roli ještě klient, který např. vyvolá vytvoření hierarchie objektů a žádá zpracování této hierarchie



Jedináček (singleton)
....

* v některých případech potřebujeme třídu, která bude mít jedinou instanci
 - např. pokud celý systém sdílí jednoho správce oken, jednu tiskovou frontu
 - třída může zajistit, že bude vytvořena její jediná instance, následujícím způsobem:
  . skryjeme konstruktor třídy (bude protected)
  . pro vytvoření instance poskytneme operaci třídy
  . pokud instance existuje, vrátí ji; pokud neexistuje, vytvoří a vrátí ji

```
public class Singleton {
    protected static Singleton instance = null;

    private Singleton() {}

    public static Singleton instance()
    {
        if (instance == null)
            instance = new Singleton();
        return instance;
    }
}
```

* klienti si vytvoří instanci pomocí Singleton.instance()
* pozor, jedináček de facto zavádí globální proměnné (i když s možností vytvářet potomky apod.), proto by jeho použití mělo být uvážené

Výše uvedená kniha (Design patterns) uvádí katalog 23 základních návrhových vzorů, se kterými se můžete často setkat, proto jí doporučuji k prostudování.

❇

KIV/ZSWI 2004/2005
Přednáška 8

Strukturovaná analýza a návrh systému
=====================================

* strukturované metodiky pro analýzu a návrh systému historicky předcházely
  objektovým metodikám
 - objektově-orientovaný vývoj - data a nad nimi pracující funkce chápeme
   jako objekty
 - strukturované - na funkce a na data se zaměřují víceméně odděleně
  . odpovídá strukturovanému programování
  . funkce jsou aktivní a mají chování
  . data jsou pasivní, ovlivněna funkcemi
  . fce systému postupně rozdělujeme shora dolů na části, nejčastěji pomocí
    diagramů datových toků (data-flow diagrams)

* dnes se všeobecně dává přednost OO metodikám, ale strukturované metodiky
  mohou stále ještě posloužit v následujících případech:
 - malé programy (několik set řádek kódu): příliš jednoduché, aby se
   vyplatilo vytvářet třídy
 - programy s krátkou dobou života, např. prototypy, které budou zahozeny
   (pokud cílem není získat představu o vytvářených třídách): opět se nemusí
   vyplácet vytvářet třídy
 - pokud se pravděpodobně bude měnit funkčnost, ale ne data: v OO přístupu by
   funkčnost byla rozprostřena mezi více objektů, proto může být výhodnější
   strukturovaný přístup (pokud se naopak budou měnit data, je OO přístup
   výhodnější, protože změny jsou zapouzdřeny do jednotlivých objektů)

* výhoda strukturovaných metodik
 - metodiky mohou být jednodušší, vytvářené modely mohou být srozumitelné
   zákazníkovi => zákazník se snáze účastní strukturované analýzy
 - návrh systému může být i rychlejší (nevytváříme přídavnou strukturu tříd)
* nevýhody strukturovaných metodik
 - jsou považovány za nemoderní
 - výsledný systém se většinou hůře udržuje

* podobně jako u objektové analýzy a návrhu se vytvářejí modely = abstrakce
  klíčových vlastností studovaného systému
 - model slouží jako vstup do dalších fází SW procesu:
  . model kontextu systému - určuje hranice vytvářeného systému
  . modely strukturované analýzy: diagramy datových toků, datový slovník, ERA
    diagramy, specifikace činností procesů
  . modely strukturovaného návrhu: strukturogramy (structure charts) - na
    jejich základě můžeme zahájit kódování
 - vytváření modelu může mít podporu CASE nástrojů (editor modelů, částečná
   kontrola modelu, automatická tvorba dokumentace)

Poznámka pro zajímavost (CASE nástroje podporující strukturované metodiky)

 Srovnání některých CASE nástrojů podporujících strukturované metodiky můžete
 najít např. v článku V. Řepy: Programování ve velkém, Softwarové noviny
 5/2003.

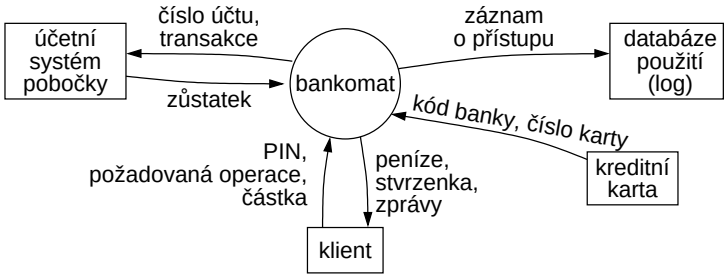
[]

Model kontextu systému
----

* již úplně na začátku získávání požadavků je třeba určit hranice
  systému, tj. co bude tvořit systém a co bude okolí systému
* v některých případech nemusí být úplně zřejmé: např. evidence příjmu dřeva
  v papírně = váha, terminály pro vstup informací o objemu a kvalitě dřeva,
  databáze...
* zvolenou hranici často určují netechnické faktory, např. jí určíme tak,
  abychom měli co nejvíce věcí pod kontrolou

* po definici hranic určíme kontext a závislosti systému na okolí
* obvykle znázorňujeme nakreslením jednoduchého diagramu kontextu (context

diagram)
- hranice vytvářeného systému je znázorněna kolečkem s vepsaným názvem systému
- lidé, organizace nebo jiné systémy, se kterými náš systém komunikuje, se znázorňují pojmenovaným obdélníkem; ve strukturované analýze se nazývají terminátory (mají stejný význam jako aktéři v OO analýze)
- vstupující a vystupující data jsou znázorněna šipkami mezi systémem a terminátorem
- například model kontextu systému pro bankomat:

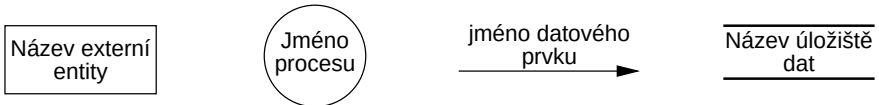


Diagramy datových toků

* angl. data-flow diagram, DFD, ale někdy také data-flow graph, work flow diagram, function model, bubble chart, process model...
* součástí různých strukturovaných metod cca 1955-1990 (ale i některých OO metodik, např. OMT), v literatuře užívány různé notace
* DFD jsou jednoduché a intuitivní (je možné je vysvětlit zákazníkovi)
* lze je použít pro modelování toku dat SW systémem, ale také modelování toku dat a fyzických předmětů (materiálu) v organizaci

* my budeme především modelovat tok dat SW systémem
* data jsou zpracovávána posloupností kroků (kroky provádějí lidé, funkce programu)

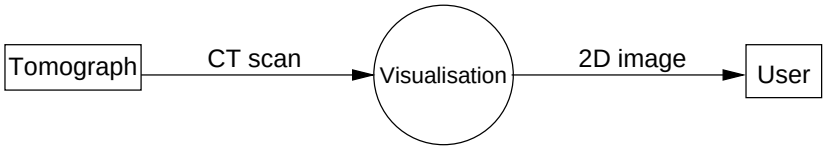
* základní notace pro DFD:
 - externí entita neboli terminátor
  . producent nebo konzument dat (začíná nebo končí v něm tok dat)
  . je mimo hranice modelovaného systému
  . může být osoba, jiný systém, hardware apod.
  . znázorněna obdélníkem s názvem uvnitř
 - proces
  . provádí transformaci dat
  . znázorněn kolečkem (bublinou), je vhodné bubliny číslovat a nutné konkrétně pojmenovat - sloveso + podstatné jméno (”ověř telefonní číslo”)
 - tok dat
  . reprezentuje ”data v pohybu”
  . znázorněn šipkou, šipka ukazuje směr toku dat; datový prvek má být pojmenován
 - paměť (datový sklad)
  . úložiště dat pro použití jedním nebo více procesy, pracujícími v různých časových obdobích
  . znázorněn dvojitou čarou
  . pokud je vyjímána pouze jedna datová položka, nemusíme označovat datový tok



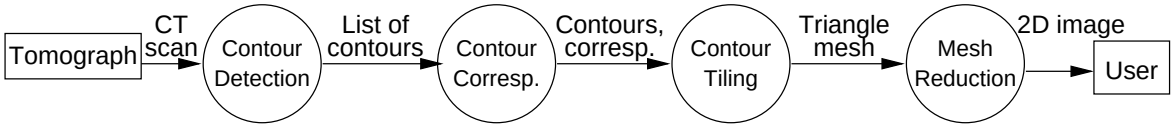
a) externí entita *b) proces* *c) tok dat* *d) úložiště dat*

* DFD může být použit pro reprezentaci systému libovolné úrovně abstrakce
 - začínáme na DFD úrovně 0 = fundamentální model systému, je vlastně totéž co již popsaný model kontextu systému

- celý SW systém je zakreslen jako jedna bublina, má jeden nebo více vstupů a výstupů
- například vizualizace snímků z tomografu:
  . vstupem je množina 2D řezů tělem pacienta
  . výstupem je 2D pohled na snímek



- systém můžeme rozdělit do menších částí a znázornit na větší úrovni podrobnosti:



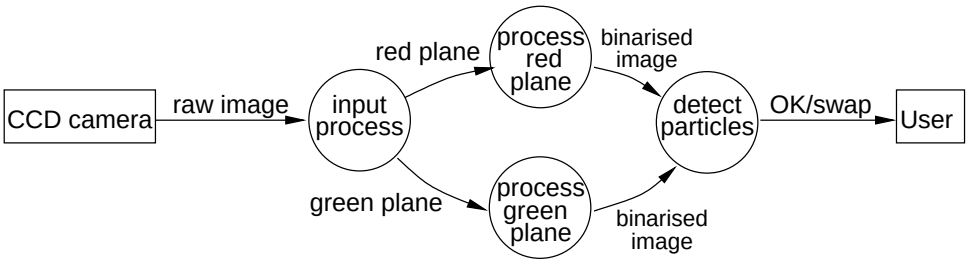
* výhodná vlastnost DFD - model může být postupně zjemňován

- např. fundamentální model systému ”Monitor kvality oleje” má vstup ”obrázek sejmutý CCD kamerou mikroskopu”, výstupem je rozhodnutí, zda je třeba olej vyměnit:

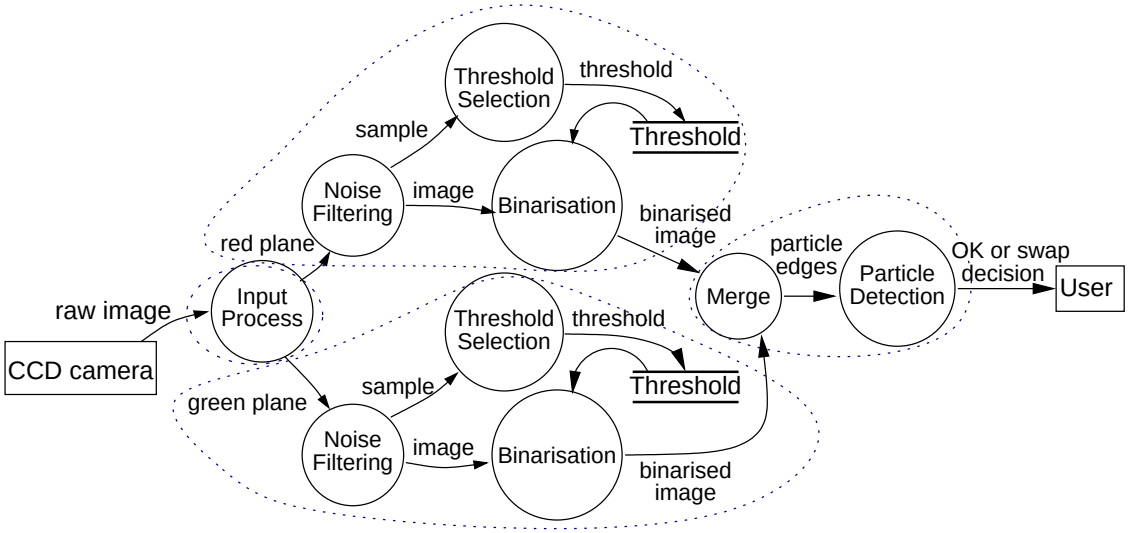
L0 DFD:

- zjemníme model systému
  . najdeme kandidáty procesů, datových toků a pamětí
  . všechny šipky, bubliny a úložiště smysluplně pojmenujeme
  . musí být udržena ”kontinuita toku dat”, tj. původní vstupy a výstupy musejí zůstat zachovány
  . pokud je DFD nepřehledný, překreslíme ho
- např. výše uvedený systém rozdělíme do čtyř transformací:

L1 DFD:



- v dalším kroku postupujeme ve zjemňování bublinu po bublině (tečkováním jsem naznačil obsah bublin z L1 DFD):



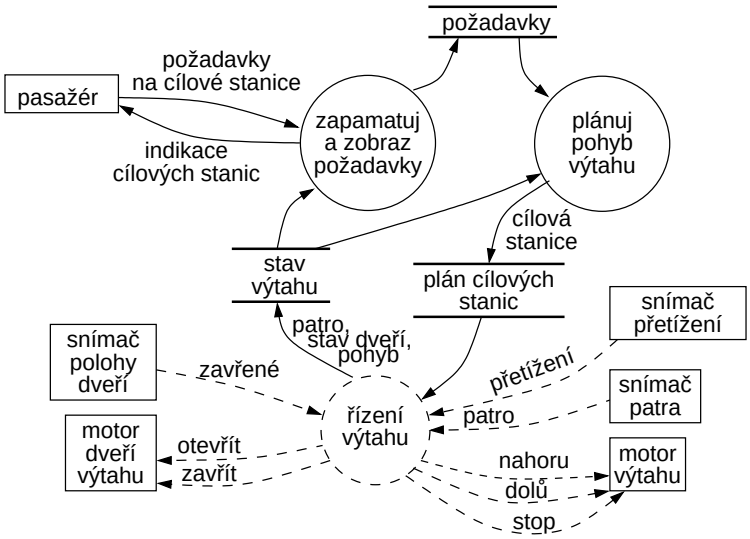
- vytvořené DFD by neměly být příliš velké - měly by se bez problémů vejít na papír velikosti A4

Co zkontrolovat:

* černé díry - bubliny, které mají vstup, ale nemají výstup
* spontánní generátory - mají výstup, ale žádný vstup; jediný rozumný případ tohoto typu je generátor náhodných čísel
* neoznačené procesy a datové toky - častým důvodem pro výskyt je to, že analytik není schopen najít pro ně rozumný název, např. protože sdružují nesouvisející procesy/toky dat (pak je řešením proces nebo datový tok rozdělit)

Poznámka (rozšíření DFD)

Výše uvedená základní varianta DFD je vhodná pro modelování systémů, které jsou řízeny datovými vstupy a kde se nezpracovávají téměř žádné vnější události. Zcela jinak je tomu v RT systémech; pro taková použití byla navržena celá řada rozšíření základního DFD. Tok řízení se většinou znázorňuje přerušovanou šipkou a řídící proces jako bublina zakreslená přerušovanou čarou. Například v následujícím DFD je řídícím procesem ”řízení výtahu”:



[]

* v dalším kroku potřebujeme
 - vytvořit definici dat
 - specifikovat logiku procesů
  . na konečné úrovni zjemnění doplňujeme popis specifikací procesu buď v přirozeném jazyce, nebo v pseudokódu

Datový slovník

* v DFD
 - šipky reprezentují tok dat
 - paměť je množina datových položek
 - je třeba mít metodu reprezentace obsahu dat v datových tocích a pamětech
 - pro popis se používá datový slovník = seznam datových prvků s definicemi

Důležitost popisu dat viz ”rozhovor s marťanem” (Yourdon 1989):

M: Co je to vlastně jméno?
Z: To je, jak se navzájem nazýváme.
M: (zmateně): Znamená to, že se nazýváte jinak, když jste šťastní než když
   máte vztek?
Z: (trochu překvapeně, že M je snad mimozemšťan): Ne, jméno je pořád stejné.
M: (konečně pochopil): Aha, rozumím, u nás máme to samé. Moje jméno je 3.1415.
Z: Ale to je přece číslo, ne jméno.
atd.; v praxi často zákazník považuje analytika za marťana.

Např. při určení ”co je to jméno” se můžete setkat s problémy:

* musí mít každý křestní jméno a příjmení? Co je ”křestní jméno” a co
  je ”příjmení” např. pro jméno ”Ing. Tran Quoc Trung”?
* jaké znaky mohou být součástí jména? Co ”Kropáčková-Jouzová” nebo ”D’Arcy”?
* jak budeme zacházet s dodatky typu ”Jiří Stivín III.”?

* proto vytváříme formálnější definici datového slovníku
* popisovaná data jednoduchá nebo složená
 - jednoduchá - známé typy dat, je třeba zadat obor hodnot, příp.
   použité jednotky, příp. přesnost
 - složená - popíšeme odkazem na jednoduché položky

Notace podle Yourdona (1989):

* neterminální symboly uvádím malými písmeny

symbol	význam	příklad	vysvětlení
=	skládá se z	X = Y	X se skládá z Y
+	a	Z = X + Y	Z se skládá z X a Y
()	může být vynecháno	Z = X + (Y)	Z se skládá z X a případně Y
{}	opakování	Z = { X }	Z se skládá z libovolného počtu X
	- pokud je nutné určit dolní resp. horní mez počtu opakování, zapisuje se před resp. za složené závorky:		
		Z = 1{X}	Z se skládá z jednoho nebo více X
		Z = {X}1	Z se skládá z 0 nebo jednoho X
		Z = 1{X}10	Z se skládá z 1 nebo 2 nebo ... 10 X
[]	jedna z možností	Z = [X | Y]	Z se skládá buď z X nebo z Y
**	komentář	*toto je komentář*	
@	klíčová položka		
@<číslo>	část složeného klíče		

Jméno by mohlo být definováno následovně:

```
jméno = (tituly) + @<2>křestní_jméno + (@<3>prostřední_jméno) +
                   @<1>příjmení + (vědecká_hodnost)
tituly = {titul}
titul  = [ Pan | Paní | Dr. | Ing. | RNDr. | MUDr. | JUDr. | Prof. | Doc. ]
vědecká_hodnost = [ CSc. | DrSc. ]
křestní_jméno    = platné_jméno
prostřední_jméno = platné_jméno
příjmení         = platné_jméno
platné_jméno = velké_písmeno + { písmeno }
písmeno = [ velké_písmeno | malé_písmeno ]
velké_písmeno = *velká písmena české abecedy*
 [ A | Á | B | C | Č | ... | Z | Ž ]
malé_písmeno  = *malá písmena české abecedy*
 [ a | á | b | c | č | ... | z | ž ]
```

[]

Vše, co se nedá popsat výše uvedeným způsobem, uvedeme jako komentář:
 - význam datového prvku v kontextu aplikace (pokud je stejný jako
   název, pak se uvádí prázdný komentář ”**”)
 - z čeho se prvek skládá, pokud je složen ze smysluplných elementů
 - rozsah hodnot, pokud element nelze dále dekomponovat; případně přesnost
   jako počet významných číslic za desetinnou čárkou

Příklady:

```
      výška    = *výška pacienta při přijetí do nemocnice*
                 *jednotka: cm; rozsah: 10-250*

      hmotnost = *hmotnost pacienta při přijetí do nemocnice*
                 *jednotka: kg; rozsah: 1-200; přesnost: 0.1 kg*

      datum-narození = **
                 *jednotka: počet dní od 1.1.1900; rozsah: 0-73000*
```

[]

* každá šipka v DFD by měla mít popis v datovém slovníku

* ve velkých systémech může mít datový slovník několik tisíc položek
* pro definici, kontrolu konzistence a úplnosti je vhodné používat
  nástroje, jsou součástí DB systémů, notace se může poněkud lišit

* je velmi žádoucí udržet co nejmenší míru redundance kvůli lokalizaci změn
* pro případná synonyma vždy pouze jednu definici

```
      zákazník = jméno + adresa + kategorie
      klient   = zákazník
                 *synonymum pro ”zákazník”*
```

* kdybychom uvedli dvě (stejné) definice, je nebezpečí, že při změně jednu z
  nich zapomeneme opravit
* někteří autoři doporučují zvýraznit to, že se jedná o synonymum uvedením
  hvězdičky ve všech výskytech (např. klient*), definici bychom zapsali
  takto:

```
         klient* = zákazník
```

ERA diagramy
----

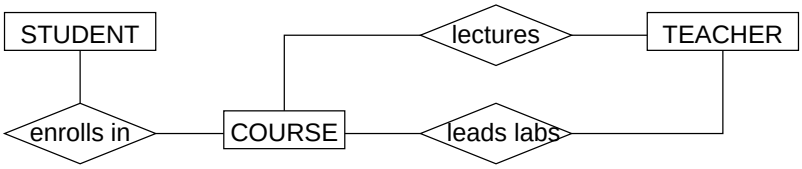
* v češtině také entitně-relační diagramy, E-R diagramy, v angl.
  nejčastěji entity-relationship diagrams (E-R diagrams, ERD)
* popisují strukturu uchovávaných dat na vysoké úrovni abstrakce
* základy Chen 1976, později vývoj různými směry - existuje mnoho notací,
  zde uvedeme pouze základní notaci

Poznámka pro zajímavost (ERA diagramy a diagram tříd v UML)

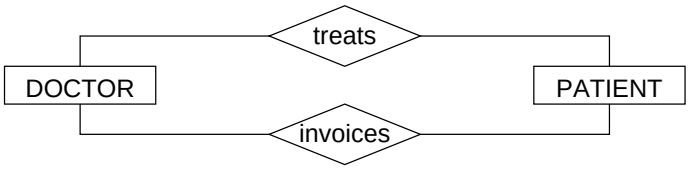
 Notace pro diagramy tříd v UML se historicky vyvinula z notace pro ERA
 diagramy, tj. podobnost mezi nimi není náhodná.

[]

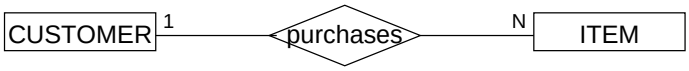
* při vytváření systému si klademe otázky:
 - které údaje musíme uchovávat v systému?
 - jaký je vzájemný vztah údajů?
* zakreslujeme nejčastěji pomocí ERA diagramu, např.:
 - student volí předměty, předměty učí a cvičí učitelé
 - přehledně můžeme nakreslit následovně:



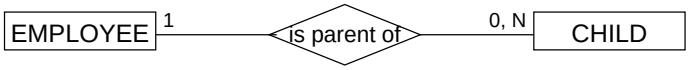
kde:
* obdélníkem zakreslujeme typy objektů (datové entity)
 - představuje množinu navzájem rozlišitelných objektů reálného světa
* kosočtvercem zakreslujeme množinu vztahů (relace)
 - relace = množina vztahů, které si systém musí zapamatovat (nelze je odvodit)
 - mezi dvěma entitami může být více relací



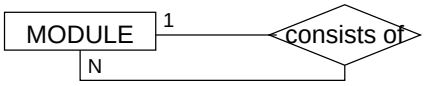
 - relace charakterizována aritou = počet objektů účastnících se vztahu
 - nejčastěji se rozlišuje 1:1, 1:N, M:N



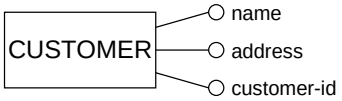
 - nepovinný (volitelný) vztah = nemusí nastat



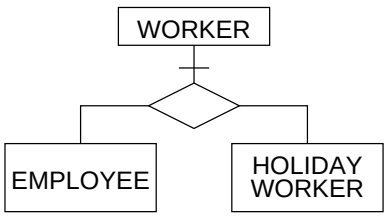
 - můžeme popsat i rekurzivní vztah



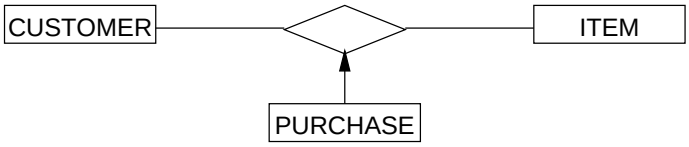
* atributy
 - popisují entity; entita popsána pomocí jednoho nebo více atributů
 - atributy se týkají všech instancí entity
 - v ERA diagramu se někdy znázorňují malým pojmenovaným kolečkem



* někdy můžeme zjistit, že různé entity jsou ve skutečnosti pouze odlišné
  formy základní entity
 - například všichni zaměstnanci mají jméno, adresu, rodné číslo
 - stálí zaměstnanci mají navíc měsíční plat a osobní ohodnocení
 - brigádníci mají navíc hodinovou mzdu
 - jinými slovy nadtyp je popsán atributy, které se týkají všech podtypů
 - analytik může znázornit pomocí nepojmenovaného kosočtverce s přeškrtnutím
   vztahu k nadtypu:



* někdy zjistíme, že potřebujeme uchovávat také informaci o relaci
 - příklad: zákazník nakoupil zboží
  . relace ”nakoupil” sdružuje zákazníka a jednu nebo více položek zboží
  . pokud bychom chtěli uchovat informaci o datu nákupu, tato informace nepatří ani k zákazníkovi, ani ke zboží
  . zavedeme ”nakoupil”, má fci typu objektu (tj. má atributy) a zároveň vztahu (spojuje zákazníka s položkami zboží)
 - nazývá se asociativní indikátor typu objektu nebo průnikový typ
 - znázorňuje se následovně:



* ERA diagramy vstup pro návrh databáze, např. ”pamětí” v DFD
* zde byly uvedeny pouze základy notace, podrobnější popis viz např.:

  Jaroslav Pokorný: Konstrukce databázových systémů.
  Vydavatelství ČVUT 1999

Specifikace činnosti procesů

* zjemňováním DFD nám vznikl rozklad problému na elementární procesy komunikující pomocí datových toků
* nyní potřebujeme specifikovat, co se děje v elementárním procesu

* pro specifikace procesů se používají následující nástroje:
 - pseudokódy nebo jejich grafický ekvivalent
 - rozhodovací tabulky
 - rozhodovací stromy
 - konečný automat (můžeme použít konvenci stavového diagramu z UML)

Pseudokódy

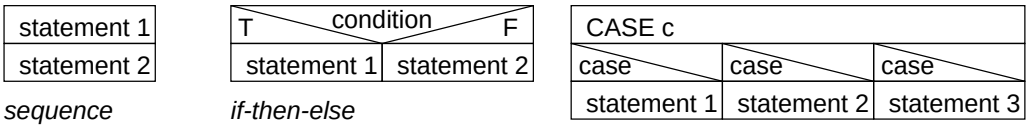
* jako příklad uvedu Yourdonovu ”strukturovanou angličtinu”, resp. ”strukturovanou angločeštinu”

* základní myšlenka - omezený slovník běžného jazyka, přidáme strukturu programovacího jazyka
* slovník se skládá z:
 - příkazů
 - rozhodovacích konstrukcí
 - opakovacích konstrukcí

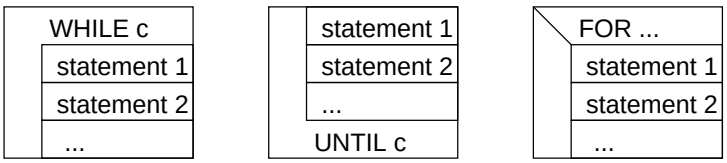
* příkazy:
 - výraz, např. X = Y*Z
 - sloveso, pojmy definované v datovém slovníku
  . pro jakoukoli specifikaci by mělo postačovat 40-50 sloves, např. READ, WRITE, CREATE, APPEND, FIND, ... resp. české načti, vypiš, vytvoř, přidej atd.
 - vnoření konstrukcí nejčastěji vyjádřeno odsazením, někdy se můžeme setkat také s číslováním 1.1.1, 1.1.1.1...
* rozhodovací konstrukce:

```
IF podmínka            IF podmínka            DO CASE
    akce                   akce1              CASE podmínka1
ENDIF                  ELSE                       akce1
                           akce2                  ...
                       ENDIF                  CASE podmínka_n
                                                  akce_n
                                              OTHERWISE
                                                  akce
                                              ENDCASE
```

```
* cykly:
  DO WHILE podmínka      REPEAT
      akce                   akce
  ENDDO                  UNTIL podmínka

Příklad:

  PROCES 1.2.3: Účtování objednávek.
  DO WHILE jsou objednávky ke zpracování
      READ objednávka
      celková-částka = 0
      DO WHILE jsou položky v objednávce
          celková-částka = celková-částka + cena-položky
      ENDDO
      WRITE (do procesu 1.1.5) ID-zákazníka + celková-částka
  ENDDO


* definice procesu by neměla přesáhnout jednu stránku A4 (cca 70 řádek)
 - pokud se nevejde, měla by se použít jiná formulace, resp. jednodušší
   algoritmus
 - pokud to nejde, možná že proces není elementární


Nassi-Shneidermanovy diagramy
.............................

* někdy se používají místo pseudokódu; uvádím pro zajímavost
```

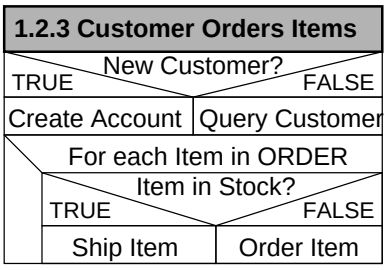


sequence *if-then-else*



while-do *repeat-until* *for-do*

```
* příklad Nassi-Shneidermanova diagramu procesu:
```

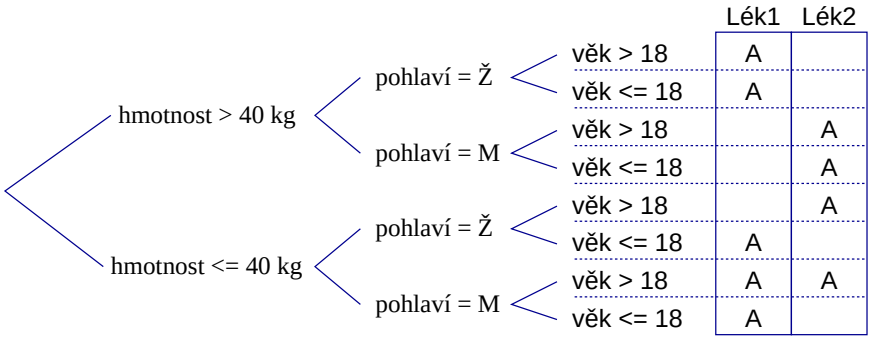



```
* po vysvětlení jsou pro zákazníky často srozumitelnější než pseudokód (podle
  publikovaných výzkumů pro 75-80% lidí)
* vyžaduje nemalé množství grafiky - vyplatí se pouze, pokud máme SW podporu
  pro jejich vytváření

* neměli bychom překročit 3 úrovně vnoření, jinak je vhodná spíše rozhodovací
  tabulka

Vstupní a výstupní podmínky
...........................

* v některých případech se udává pouze vstupní a k ní odpovídající výstupní
```

```
  podmínka:

  PROCES 1.2.3: Účtování objednávek.
  Vstupní podmínka:
      na vstupu (z procesu 1.1.4) se objeví objednávka
  Výstupní podmínka:
      na výstup (do procesu 1.1.5) je zasláno ID-zákazníka + celková-částka
  Vstupní podmínka:
      na vstupu (z procesu 1.1.4) je objednávka, zákazník není v databázi
  Výstupní podmínka:
      je vygenerována chybová zpráva

* běh procesu je spouštěn určeným vstupem

Rozhodovací tabulky a stromy
............................

* používají se, pokud by byl pseudokód vyjadřující podmínku složitý

* nejprve najdeme všechny relevantní vstupy, učíme počet možných kombinací
* vytvoříme tabulku
 - sloupce = pravidla: počet pravidel je dán počtem kombinací vstupů
 - řádky tabulky: nejprve všechny vstupy, pak všechny výstupy
```

	1	2	3	4	5	6	7	8
věk > 18	A	N	A	N	A	N	A	N
pohlaví = Ž	A	A	N	N	A	A	N	N
hmotnost > 40 kg	A	A	A	A	N	N	N	N
podávat lék 1	A	A				A	A	A
podávat lék 2			A	A	A		A	

```
* horní polovina tabulky definuje podmínky
* ve spodní polovině je pro každý sloupec (= podmínku) uveden výstup

* pokud uživatel, se kterým musíme konzultovat, odmítne tabulku jako
  nesrozumitelnou, můžeme použít rozhodovací strom:
```




```
* výhoda obou způsobů:
 - specifikací není určen způsob implementace
 - s uživatelem můžeme probrat jedno pravidlo po druhém
 - a hlavně máme jasnou odpověď pro všechny kombinace podmínek

Strukturovaná analýza
---------------------

V předcházejícím textu jsme popsali nástroje potřebné pro strukturovanou
analýzu, ale zatím jsme neuvedli, jak tyto nástroje použít.

V dalším textu si proto uvedeme základy dvou nejznámějších metodik, a to
klasické DeMarcovy metodologie a Yourdonovy "moderní strukturované analýzy".

* vstupem strukturované analýzy (DeMarco) jsou uživatelské požadavky,
```

výstupem je tzv. strukturovaná specifikace
- systém je specifikován pomocí DFD, uvedeny podstatné procesy, paměti a údaje
- případné jednodušší procesy v DFD nižší úrovně
- elementární procesy zapsány v pseudokódu, rozhodovací tabulkou nebo stromem
- v datovém slovníku popis dat

Otázka: co má analytik vyrobit - model původního systému nebo nového?

* předpokládejme:
 - existující systém s ne zcela zřetelnou strukturou plní určité funkce
 - systém z nějakého důvodu nahrazujeme novým systémem
 - jak najít specifikaci nového systému?

* klasická strukturovaná analýza pomocí vytvoření 4 modelů systému:
1. fyzický model stávajícího systému (jaký systém používá zákazník?)
   - analytik zmapuje stávající systém, jeho fční strukturu a data
   - může obsahovat zpracovávaní a přesun fyzických formulářů apod.
2. logický model stávajícího systému (jaká je jeho logická struktura?)
   - z fyzického modelu vytvoříme logický (cca 75% redukce)
   - zrušíme všechny implementační detaily, co by systém dělal kdybychom měli ideální technologii (nekonečné paměti, nekonečnou rychlost atd.)
   - získáme logické procesy a podstatu transformace dat
3. logický model nového systému (co je třeba změnit?)
   - většina systému pravděpodobně zůstane stejná + požadavky na nové fce
   - po konzultaci s uživatelem promítnuty změny do logického modelu
4. fyzický model nového systému (jak to nejlépe implementovat?)
   - návrh implementace.

Problémy při striktním dodržování modelu:

* při vytváření fyzického modelu ví uživatel o systému víc než analytik; u uživatele tím často vznikne dojem, že analytik problematice nerozumí a že se jí teprve za jeho peníze učí
* uživatel odmítá spolupráci na vývoji nového logického modelu; má pocit, že pokud analytik neumí vytvořit bez pomoci zákazníka fyzický model stávajícího systému, pak nemůže umět ani dobře navrhnout nový systém
* analytickou práci někteří uživatelé považují za oddech vývojářů před ”skutečnou prací” (kódováním), tvorba 4 modelů tuto dobu prodlužuje, což snižuje ochotu spolupracovat s analytikem (tj. 4 modely je prostě moc)
* zmíněné problémy řeší ”moderní strukturovaná analýza” (Yourdon 1989)

Vyvažování modelů

Strukturovaná analýza:
* model kontextu systému
* diagramy datových toků (DFD)
* datový slovník
* ERA diagramy
* specifikace činnosti procesů
 - pseudokódy
 - Nassi-Shneidermanovy diagramy
 - rozhodovací tabulky a stromy

* každý model se zabývá nějakým aspektem modelovaného systému
 - DFD, specifikace procesů - funkce systému
 - datový slovník, ERA diagramy - data systému
* ve velkých systémech by bylo snadné vytvořit několik nekonzistentních pohledů na systém
 - např. v datovém slovníku mohou zůstat položky vzniklé v počáteční fázi analýzy systému, které se v DFD už nevyskytují
* proto je po dokončení modelů třeba provést tzv. vyvažování modelů

* vyvažování DFD a datového slovníku = zajistíme následující:
 - každý datový tok a každá paměť musí být definována v datovém slovníku
 - každá datová položka v datovém slovníku se musí vyskytovat v DFD
 - mechanická práce => je vhodné mít podporu CASE nástroje
* vyvažování DFD a specifikace procesů

- každá bublina v DFD musí být sdružena buď s DFD nižší úrovně, nebo se specifikací procesu
- každá specifikace procesu musí mít bublinu v DFD nejnižší úrovně
- vstupy a výstupy si musejí vzájemně odpovídat

* vyvažování specifikace procesů a datového slovníku
- všechna data použitá ve specifikaci procesu musejí být definována buď lokálně, nebo v datovém slovníku
- každá položka v datovém slovníku musí být odkazována ze specifikace procesu nebo z jiné položky datového slovníku

* vyvažování ERA diagramu oproti DFD a specifikaci procesu
- každá paměť v DFD musí odpovídat entitě, relaci nebo entitě+relaci v ERA diagramu
- položky v datovém slovníku popisují jak toky v DFD tak entity v ERA modelu
- procesy musejí být schopny:
  . vytvářet a rušit instance všech entit a relací v ERA diagramu
  . nastavovat hodnotu a používat hodnotu instance.

❇

KIV/ZSWI 2004/2005
Přednáška 9

Moderní strukturovaná analýza

* Yourdon 1989
* místo čtyř modelů systému se zaměřuje přímo na nalezení esenciálního modelu systému, tj. logického modelu nového systému

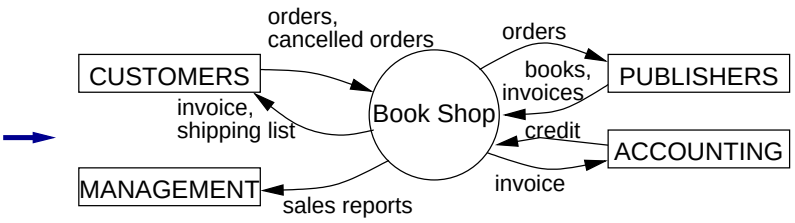
* v moderní strukturované analýze se postupuje v těchto krocích:
 - vytvoříme model prostředí (definuje hranici mezi systémem a zbytkem světa)
  . nejprve definujeme účel systému (neměl by být delší než odstavec)
  . vytvoříme model kontextu systému
  . vytvoříme počáteční datový slovník definující data putující mezi systémem a terminátory
 - vytvoříme seznam událostí
 - na základě událostí vytvoříme předběžný model chování systému
 - model chování systému přestrukturujeme do konečného modelu chování (= esenciální model systému)
 - vytvoříme uživatelský implementační model (doplňuje esenciální model o informace nutné pro implementaci modelu)
 - konec analýzy, následuje návrh architektury, podrobný návrh a kódování

Vytvoření seznamu událostí

* události se klasifikují na:
 - událost datového toku (F) - událost která se projeví příchodem dat; např. událost ”přišla objednávka”
 - časová událost (T) - např. zpracování transakcí mezi bankovními účty nastane ve 3:00
 - řídící událost (C) - asynchronní událost např. v RT systémech

* identifikace událostí: procházíme každý terminátor, ptáme se jaké akce může provádět nad systémem (obvykle probíráme společně s uživateli, kteří hrají role terminátorů - analogicky jako při identifikaci případů použití v OO analýze)

EVENT LIST:
1. something happened (F)
2. ....



 - méně častá alternativa: vycházíme z ERA diagramu a hledáme události které způsobují vznik a zánik instancí entit a relací

* pro každého kandidáta na událost se ptáme, zda je skutečně událostí, tj. zda v jejím důsledku systém vyprodukuje výstup nebo změní svůj stav
* pro kandidáta na událost se ptáme, zda se všechny instance události týkají stejných dat (pokud ne, budou to nejspíš dvě různé události; cílem je rozlišit mezi různými událostmi, které se náhodou dějí společně nebo vypadají podobně)
* pak se pro každou událost ptáme ”musí systém nějak reagovat, pokud se událost neudá podle očekávání?” (tj. modelujeme odpovědi na chyby/poruchy terminátorů; např. zboží nepřijde v očekávané době, co musí systém udělat?)

Vytvoření předběžného modelu chování

* v klasické strukturované analýze se využívá postup shora dolů, pak ale často nastává tzv. problém ”paralýzy analýzy”: při definici velkého systému

  postrádáme vodítko, jak vytvořit z kontextového diagramu DFD úrovně 1
* proto se v moderní strukturované analýze využívá identifikace odpovědí na
  události

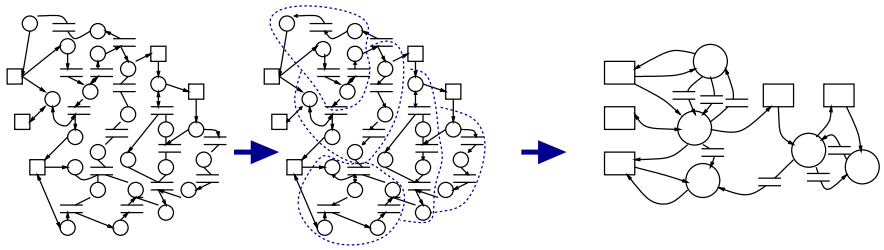
* postup pro vytváření předběžného modelu chování:
 - pro každou událost ze seznamu nakreslíme bublinu, očíslujeme jí podle
   čísla události
 - bublinu pojmenujeme podle odpovědi, která by měla být sdružena s
   událostí (např. odpověď na ”zákazník zaplatil fakturu” je ”vlož
   platbu mezi příjmy”)
 - pokud je na událost více odpovědí, přidáme pro další nezávislou
   odpověď další bublinu (nezávislé odpovědi = mezi bublinami není
   komunikace)
 - k bublině nakreslíme vstupy, výstupy a potřebné paměti
 - identické bubliny sdružíme do jedné (identické bubliny mají stejný
   vstup, výstup a proces; např. různé typy objednávek mohou mít
   stejnou odpověď)
 - výsledný DFD zkontrolujeme oproti kontextovému diagramu a seznamu
   událostí

* výsledný model by měl
 - popisovat pouze logické procesy a podstatu transformace dat
 - zcela vynechat implementaci (nerozlišuje ani, zda fci provádí člověk nebo
   existující počítačový systém) - proto vynecháme např.:
  . procesy, jejichž účelem je přenos dat z jedné části systému do jiné
  . procesy pro verifikaci dat vzniklých uvnitř systému
  . procesy pro fyzický vstup a fyzický výstup (”vytiskni fakturu”)
 - je třeba rozlišovat mezi zdrojem informace (”obchodní zástupce”) a
   mechanismem pro vstup/výstup (”systém pro zadávání objednávek”);
   mechanismy vynecháme

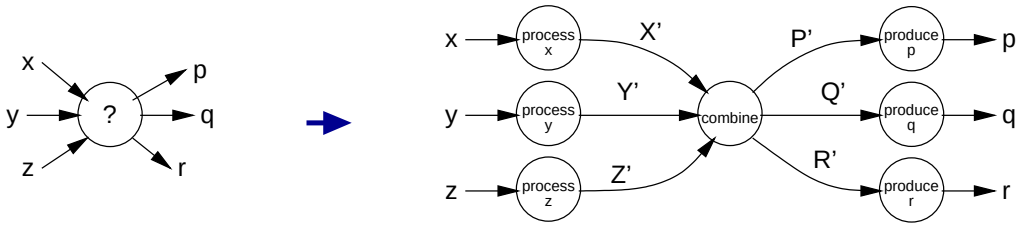
Dokončení modelu chování
.....

* předběžný DFD chování systému bude zbytečně komplikovaný, zjednodušíme ho
* dokončíme specifikaci procesů
* dokončíme datový slovník

* zjednodušení DFD chování systému strukturováním
 - agregace = spřízněné procesy sdružíme do agregátů (= seskupení),
   které budou tvořit bubliny v DFD vyšší úrovně
 - každý agregát by se měl týkat úzce příbuzných odpovědí na události,
   tj. většinou procesů zpracovávajících příbuzná data
 - sdružovat bychom měli skupiny po cca 7+-2 procesech+pamětech
 - při tom máme příležitost ”schovat” paměti, které jiné procesy nepotřebují



* někdy procesy naopak nejsou primitivní a vyžadují další rozdělení
 - např. v následujícím příkladu: bublinu jsme nedokázali dobře pojmenovat,
   proto jí rozdělíme na primitivní procesy



* po dokončení modelu chování systému vytvoříme specifikaci procesů
* dokončíme datový slovník, ERA model

* výše uvedené informace tvoří tzv. esenciální model systému

Vytvoření uživatelského implementačního modelu
.....

Uživatelský implementační model pokrývá:

* alokaci esenciálního modelu na lidi a stroje (které bubliny a paměti budou realizovány manuálně) - to musí rozhodnout uživatel
* rozhraní aplikace s uživatelem (volba vstupních a výstupních zařízení, formáty obrazovek, formáty výstupů)
* omezení, např. objemy dat, časy odpovědí na různé události atd., tj. mimofunkční požadavky na aplikaci

* vytvoření uživatelského implementačního modelu je oficiálně posledním krokem moderní strukturované analýzy požadavků.
 - následuje návrh architektury, podrobný návrh, implementace, testování

Poznámka:

Při analýze nemusejí být použity všechny nástroje (modely) - používají se vždy pouze ty nástroje, které mají v daném kontextu smysl. Například pokud jsou datové struktury jednoduché, nemusíme vytvářet ERA diagramy.

[]

Kromě klasické (DeMarcovy) metodologie a (Yourdonovy) moderní strukturované analýzy se v praxi používají další metodiky pro strukturovanou analýzu, např. SADT, SREM/RDD, SA/SD. V Česku je z nich asi nejpoužívanější SSADM (Structured Systems Analysis and Design Method); má v podstatě podobný záběr a nástroje jako DeMarcova a Yourdonova strukturovaná analýza (výstupy - DFD, model entit, ...), ale je to standard - velmi podrobně definované kroky včetně výstupů a předepsaných kontrol před přechodem k dalšímu kroku. Použití metodiky SSADM je v některých zemích podmínkou pro získání státních zakázek.

Návrh architektury
==========

* architektura softwaru je popis podsystémů (komponent) a vztahů mezi nimi
 - jednotlivé aspekty architektury se obvykle nazývají ”pohledy” (viewpoints)
 - např. fyzický pohled (umístění komponent), procesní pohled (týká se paralelismu), strukturální pohled (jak vývojáři rozdělí systém do implementačních částí) apod.
* při klasickém postupu vycházejícím ze strukturované analýzy nás bude zajímat zejména fyzický a strukturální pohled
 - nejprve rozhodneme, které části esenciálního modelu budou alokovány na které počítače (např. celý systém bude implementován na jednom počítači)
 - pak přiřadíme ”procesy” a ”paměti” alokované na stejný počítač jednotlivým procesům

Podrobný postup návrhu architektury byl již popsán v 6. přednášce - pro strukturované metody vývoje je postup v zásadě stejný, a proto ho zde nebudu opakovat. V mnoha případech je vhodné vycházet z tzv. architektonických stylů (makro-architektonických vzorů), které představují typické řešení architektury pro daný typ aplikace.

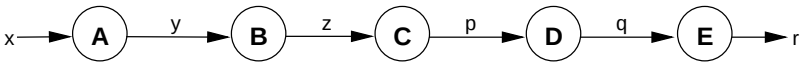
Příklady architektonických stylů uvedu později.

Strukturovaný návrh systému
==========

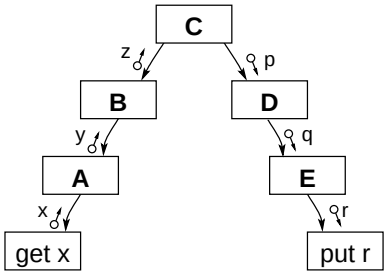
* někdy se také nazývá funkčně orientovaný návrh (function-oriented design)
* předcházela mu strukturovaná analýza a návrh architektury
 - produktem analýzy jsou:

. model kontextu systému resp. DFD úrovně 0
. množina diagramů datových toků (DFD)
. specifikace činnosti elementárních procesů (pseudokódy,
Nassi-Shneidermanovy diagramy, rozhodovací tabulky a stromy)
. datový slovník
. ERA diagramy
- produktem návrhu architektury je rozhodnutí, které části esenciálního modelu budou přiřazeny jednotlivým podsystémům, případně na které počítače nebo procesory budou alokovány (např. celý systém bude implementován na jednom počítači)

* podrobný návrh aplikace probíhá v následujících krocích:
- v rámci podsystémů transformace DFD na hierarchii podprogramů (každá bublina se stane podprogramem)
- např. z DFD



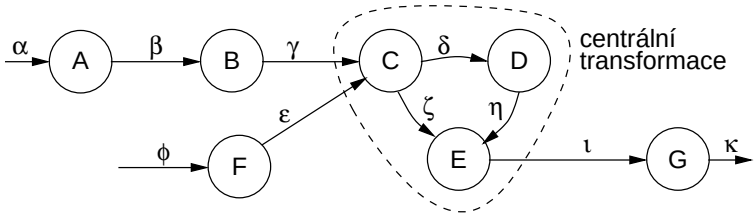
vzniknou podprogramy s následující hierarchií:



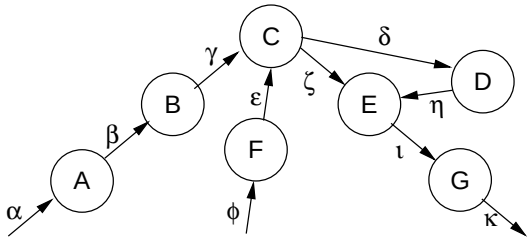
Transformace DFD na hierarchii podprogramů
.....

V obecném případě není převod DFD tak přímočarý jako v předchozím příkladu, proto se používá následující postup:

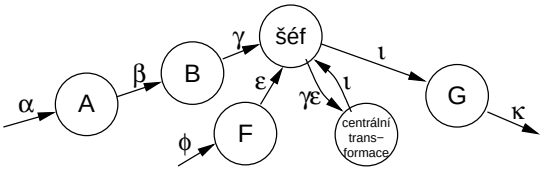
* vycházíme z DFD nejnižší úrovně (bubliny = elementární procesy); pro každý DFD provedeme:
- identifikujeme centrální transformaci = část DFD popisující základní fce systému, nezávislá na implementaci vstupů a výstupů
- dva způsoby hledání centrální transformace:
. buď předpokládáme ”ideální svět”, kde vstupy nikdy neobsahují chyby, výstupy nemusí být formátovány apod.; odsekáváme všechny nepotřebné vstupní a výstupní proudy, co nám zůstane je centrální transformace
. nebo najdeme ”střed” DFD odhadem (to je horší varianta, ale nic jiného nám nezbývá, pokud nedokážeme centrální transformaci identifikovat jinak)
- kolem centrální transformace nakreslíme obláček



* vytvoříme hierarchii procesů
- hierarchie procesů bude vyjadřovat vztah nadřízenosti a podřízenosti
- nejprve musíme najít kandidáta na šéfa
. pokud je dobrý kandidát (v centrální transformaci), zvedneme ho a necháme ostatní bubliny viset dolů, viz obrázek (a)
. pokud je potenciálních kandidátů více, vyzkoušíme, který nám poskytne nejlepší návrh



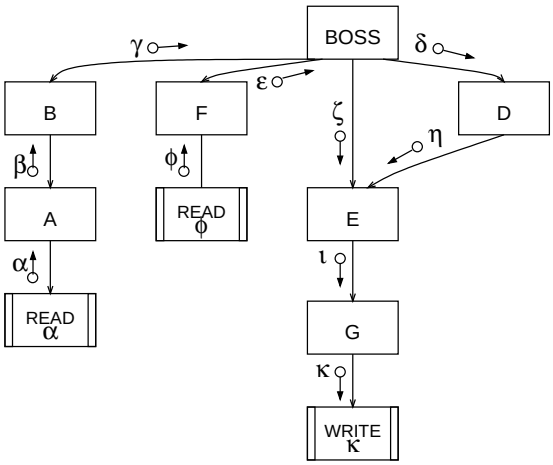
a) vyzvednutí šéfa



b) vytvoření nového šéfa

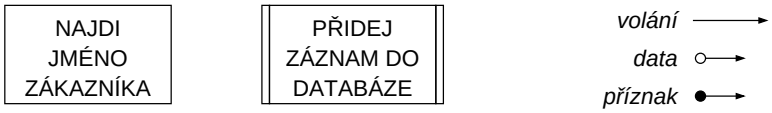
. pokud není dobrý kandidát, vytvoříme nového šéfa, centrální transformaci nahradíme šéfem a původní transformaci pod šéfa zavěsíme (jako jednu bublinu, vstupy a výstupy budou proudit přes šéfa; viz obrázek (b))

* transformace DFD na hierarchii bloků (každá bublina se stane blokem)
- ”šéf” se stane řídícím blokem, ostatní budou jeho podřízení
- vytvoříme počáteční strukturogram



- šipky ukazují směr volání, přidali jsme bloky pro vstup a výstup
- jména bloků by měla odpovídat jejich rolím v hierarchii, tj. nemusejí odpovídat názvům bublin - jméno by mělo sumarizovat i činnost podřízených bloků
  . příklad názvů bloků - postupně: ”získej položku” (blok pro čtení dat), ”získej transakci” (pomocí nižšího bloku vytvoří transakci), ”získej platnou transakci” (ověří platnost transakce)

- strukturovaný design se znázorňuje pomocí tzv. strukturogramů (structure charts)
  . strukturogramy ukazují rozdělení systému do hierarchie bloků a jejich vzájemnou komunikaci
  . blok představuje proceduru nebo fci programovacího jazyka; blok se znázorňuje jako obdélník obsahující název bloku
  . knihovní procedury a fce (předdefinované bloky) se znázorňují pomocí zdvojení svislých čar obdélníka
  . volání podprogramu je znázorněno obyčejnou šipkou (od volajícího k volanému)
  . předávání zpracovávaných dat je znázorněno šipkou s prázdným kroužkem (šipka směřuje od odesilatele k příjemci dat)
  . předávání příznaků je znázorněno šipkou s plným kroužkem



blok *knihovní (předdefinovaný) blok*

Poznámka pro zajímavost (blok vs. modul ve strukturovaném designu)

V původní literatuře o strukturovaném návrhu se o blocích hovoří jako o ”modulech”. Termín ”modul” má ale dnes už jiný význam než v roce 1980, proto

ho v souvislosti se strukturogramy raději nebudeme používat.

[]

* úprava strukturogramu
 - přidáme čtecí a zápisové bloky, bloky pro čtení z databáze apod.
 - centrální transformaci můžeme rozdělit podle DFD
 - přidáme bloky pro obsluhu chyb
 - pokud jsou potřebné, přidáme bloky pro inicializaci a ukončení programu
 - pokud je nesprávný výběr šéfa, změníme ho (nesprávný výběr šéfa se obvykle projeví tak, že pravý šéf signalizuje svému nadřízenému, co má dělat)

* ověření funkčnosti návrhu - implementuje strukturogram požadavky vyjádřené v DFD?
 - pokud si nějaký blok potřebuje vyžádat data, může to udělat?
 - pokud ne, můžeme změnit hierarchii volání

* pokud jsme výše uvedené kroky provedli pro jednotlivé DFD, máme k dispozici množinu nezávislých strukturogramů
 - vytvoříme nadšéfa, který bude volat jednotlivé šéfy
 - optimální je, pokud nadšéf obsahuje konstrukci case, která vyvolá některého podřízeného (např. výběrem položky v menu); v takovém případě bychom museli už DFD rozdělit podle typů transakcí

* tím končí fáze strukturovaného návrhu
 - hlavním účelem strukturovaného návrhu je rozdělení systému do bloků
 - výstupem je jeden nebo více strukturogramů, z bloků vzniknou podprogramy
 - strukturovaný návrh neřeší sdružení podprogramů do modulů (to probereme dále) nebo návrh vnitřku podprogramů (zde můžeme vycházet z pseudokódu, viz přednáška o kódování)

Charakteristiky kvalitních podprogramů

Dvě základní charakteristiky dobrého návrhu podprogramů je silná soudržnost a slabá provázanost.

Soudržnost

* soudržnost (cohesion) = jak blízký vztah k sobě mají operace uvnitř podprogramu
* existuje několik úrovní soudržnost: funkční, sekvenční, komunikační atd.

* funkční soudržnost - podprogram vykonává právě jednu funkci
 - např. fce sin() bude mít funkční soudržnost, protože vykonává jedinou fci
 - fce sin_and_tan() by neměla funkční soudržnost, protože by prováděla dvě fce
 - podprogramy by měly být silně soudržné, tj. dělat pokud možno jen jednu věc
 - důsledkem silné soudržnosti je vyšší spolehlivost (to je prokázáno studiemi)

Z výše řečeného vyplývá, že funkce by neměly být víceúčelové (klasický špatný příklad: fce realloc(ptr, size) jazyka C zvětšuje alokovaný úsek paměti, zmenšuje ho, uvolňuje (pokud size=0), alokuje nový úsek paměti (ptr=NULL), atd.). Proto je to také nejhůře srozumitelná fce standardní knihovny jazyka C.

* sekvenční soudržnost - slabší typ soudržnosti než je funkční soudržnost
 - podprogram sestává z kroků, které se musejí provést v určitém pořadí
 - kroky postupně zpracovávají sdílená data, ale netvoří ucelenou fci
 - například program bude provádět 10 kroků, pokud je prvních 5 v jedné fci a druhých 5 ve druhé, je to sekvenční soudržnost
 - funkční soudržnost je lepší => fce sdružíme do jedné, operace bude ucelená

* komunikační soudržnost - ještě slabší typ
 - operace provádějí zpracování stejných dat, ale jinak spolu nemají nic společného; např. změna dvou nesouvisejících prvků datové struktury
 - z praktických důvodů ještě akceptovatelné

Poznámka (analogie soudržnost podprogramů pro třídy: soudržnost třídy)

Jedna z charakteristik dobře navržených tříd je soudržnost třídy; je to analogie soudržnosti podprogramů, avšak o jednu úroveň zapouzdření výše. Ukazuje vzájemný vztah sady atributů a operací tvořících rozhraní třídy. Podrobněji viz (Page-Jones 2001), kap. 9.3.

[]

Provázanost
....

* stupeň provázanosti (degree of coupling) = jak blízký vztah mají dva podprogramy
 - např. fce sin(x) má slabou provázanost s ostatními podprogramy, protože jediné, co potřebuje znát, je jednoduchý vstupní parametr x
 - poněkud silnější typ provázanosti nastává, pokud si podprogramy předávají datové struktury
 - ještě silnější provázanost nastává, pokud procedury používají globální data atd.

V praxi obvykle není důležité určovat přesnou úroveň soudržnosti a provázanosti, ale je třeba se snažit o silnou soudržnost a slabou provázanost.

Vytváření modulů
----

* pro účely dalšího výkladu zadefinujeme následující termíny:
 - podprogram = procedura nebo funkce
 - modul = množina dat a podprogramů, například ”unit” v některých verzích Pascalu, zdrojový soubor v C apod.
  . bývá obvykle definován jako nejmenší jednotka zdrojového textu programu, kterou můžeme samostatně přeložit

Vylepšení modularity podsystému
....

Poté, co byla vytvořena struktura podsystému, můžeme jí dále vylepšit pomocí následujících pravidel:

* projdeme jednotlivé podprogramy, zmenšujeme provázanost a zvyšujeme soudržnost
 - pokud je ve dvou podprogramech stejná činnost, může být vyjmuta do samostatného podprogramu
 - pokud jsou dva podprogramy vysoce provázány, může být vhodné je spojit (zjednoduší se jejich rozhraní)
* omezení působnosti procedury: procedura by měla mít vliv pouze na podřízené procedury
 - podřízených procedur nemá být velké množství (”low fan-out”)
* s rostoucí hloubkou strukturogramu se snažte o to, aby procedura byla volána vyšším počtem nadřízených procedur (”high fan-in”)
 - strukturogram by měl mít tvar oválu, na spodní úrovni by měly být všeobecně užitečné procedury, které bude volat více nadřízených
* rozhraní každého podprogramu se snažíme co nejvíce zjednodušit

Návrh modulů
....

* perfektní modularita jednotlivých podprogramů = komunikace pouze pomocí jednoduchého rozhraní
 - skupina podprogramů může sdílet společná data, podprogramy pak nejsou perfektně modulární
 - pokud společná data skryjeme uvnitř modulu (budou přístupná pouze podprogramům uvnitř modulu) a zbytek programu může s modulem komunikovat pouze pomocí jeho oficiálního rozhraní, bude skupina podprogramů perfektně

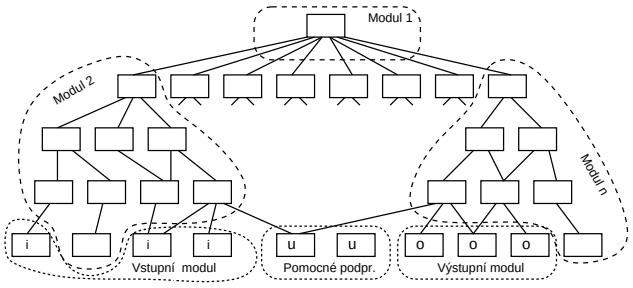
modulární, i když jednotlivé podprogramy perfektně modulární nejsou

* moduly procedurálních programovacích jazyků jsou vlastně ”poor folks’ objects”
 - sdružují data a operace nad daty, případně jinou navzájem související množinu služeb
 - moduly podporují OO koncepty abstrakce a zapouzdření, nepodporují dědičnost
 - dobrý modulární návrh podstatně usnadňuje údržbu (lokalizace změn)

* moduly je vhodné vytvořit pro všechny služby, jejichž implementace se může změnit, nebo pro součásti, které lze využít v jiných projektech
 - uživatelské rozhraní: bude se pravděpodobně vyvíjet, zbytek systému by jím neměl být ovlivněn
 - HW nebo systémové závislosti: pro snadnější přenos systému do jiného prostředí (například operační systémy: moduly pro jednotlivé HW architektury, ovladače apod.; nebo Netscape - moduly pro práci se sítí ve Windows a v UNIXu)
 - vstupy/výstupy: snadnější změna formátu souborů, výstupních tiskových sestav apod.
 - správa dat - moduly pro každý typ dat; k datům přistupujeme pouze pomocí podprogramů, což usnadňuje změnu reprezentace apod. Např. modul implementující zásobník by měl procedury rozhraní init_stack(), push() a pop()
 - moduly pro navzájem příbuzné operace, např. manipulace s řetězci, statistické fce, grafické podprogramy apod. Např. modul obsahující trigonometrické fce sin(), cos(), tan() atd.

Vyváření modulů v návaznosti na strukturovaný design
....

* pokud navazujeme na strukturovaný design, máme k dispozici strukturogram
 - sdružíme související bloky nízké úrovně (bloky nízké úrovně budou nejčastěji vstupy, výstupy, přístup k databázi, pomocné podprogramy, uživatelské rozhraní)
 - zbytek strukturogramu rozdělíme na části, které jsou silně kohezivní a k ostatním částem se váží volně (snažíme se o minimální rozhraní mezi částmi)
 - bloky v každé části budou tvořit jeden nebo více modulů



Podle Page-Jonese (1980) se ukazuje jako nejlepší následující přiřazení modulů týmům programátorů:

* návrháři systému kódují moduly vyšší úrovně
* jednotlivé typy modulů nižší úrovně přiřadíme týmům, které s danou oblastí mají zkušenosti (databáze, uživatelské rozhraní apod.)
 - týmy mohou obsahovat i méně zkušené programátory, návrháři systému na ně dohlížejí

Poznámka (klasická chyba při návrhu modulů)

Mezi klasické chyby patří návrh modulů tak, aby odpovídal počtu dostupných programátorů. Správný návrh zapouzdřuje podprogramy do modulů podle výše uvedených kritérií (jako je zapouzdření příbuzných operací atd.) a na počtu programátorů by neměl přímo záviset.

[]

❇

KIV/ZSWI 2004/2005
Přednáška 10

Návrh uživatelského rozhraní

* většinu aplikací můžeme rozdělit do dvou částí - aplikační logika a uživatelské rozhraní (user interface, dále jen UI)
* starší aplikace textové rozhraní, v současných systémech grafické uživatelské rozhraní (GUI), protože dovoluje podstatně více možností

Doporučení pro návrh UI

Mandel (1997) popisuje ve sv knize o návrhu uživatelských rozhraní tři často citovaná ”zlatá pravidla”:

* systém by měl reagovat na potřeby uživatele, tj. uživatel by ho měl řídit (nikoli naopak)
* rozhraní by mělo být konzistentní = všechny jeho části by se měly chovat podobně
* dobře navržené UI by nemělo mít požadavky na paměť uživatele

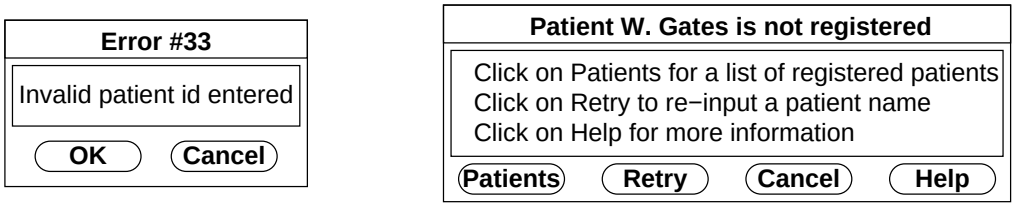
Tato pravidla můžeme trochu rozepsat:

* systém by měl reagovat na potřeby uživatele, tj. uživatel by ho měl řídit
  - uživatel by neměl být nucen používat nějaké rozhraní jen proto, že se snadno implementuje; uživatel by neměl být nucen vykonávat nechtěné akce
    . např. pokud spustím fci spell check textového procesoru a uvidím místo, které bych chtěl opravit, aplikace by mě neměla nutit zůstat v režimu spell check
  - základní terminologie rozhraní by měla vycházet z aplikační domény (uživatel se pohybuje ve virtuálním světě aplikace, např. interaguje s objekty, které se objevují na obrazovce)
    . např. v systému pro řízení letového provozu bude operátora informovat o letadlech, letových koridorech, radiomajících apod.; implementace (přístup k databázi, komunikace atd.) by měla být před uživatelem skryta
  - pokud uživatel udělá chybu, má mít možnost operaci předčasně ukončit (cancel) nebo se vrátit do stavu před vykonáním chybné akce (undo)
    . mechanismus ”cancel” - uživatelé budou nevyhnutelně při používání systému chybovat, UI má obsahovat možnost zrušení části nebo celé transakce (např. zadávání pacienta)
  - UI by mělo poskytnout přiměřené možnosti pro různé typy uživatelů; zkušení uživatelé budou chtít např. klávesové zkratky pro zrychlení přístupu k možnostem systému, na druhou stranu začátečníci potřebují vedení a nápovědu ve stylu kuchařky
    . např. některé volně šířené programy dovolují nakonfigurovat, zda je uživatel začátečník, středně pokročilý či pokročilý, a podle toho uživatele v různé míře ”obtěžují” nápovědou

* UI by mělo být konzistentní, tj. všechny příkazy a menu by měly používat stejný formát
  - konzistence redukuje čas učení, protože znalost získanou v jedné části systému lze použít i v jiné části systému
  - UI by mělo být založeno na srozumitelné metafoře (analogii) z reálného světa
* s konzistencí souvisí princip nejmenšího překvapení - podobné akce vykonaná různých kontextech by měly mít podobné důsledky; pokud se systém zachová neočekávaným způsobem, budou uživatelé nemile překvapeni
  - např. pokud při kreslení jednoho objektu znamená stisk pravého tlačítka myši zrušení objektu a při kreslení jiného připojení další čáry, není UI aplikace konzistentní a může dojít k nemilému překvapení uživatele
  - UI by mělo být konzistentní i mezi podsystémy, např. klávesa Backspace by měla vždy dělat to samé (např. zrušit znak vlevo od kurzoru)

* dobře navržené UI by nemělo mít požadavky na paměť uživatele
  - UI by mělo omezovat potřebu uživatele pamatovat si předchozí akce a jejich výsledky (mělo by uživateli napomáhat zjistit či připomenout apod.)
  - UI by mělo obsahovat nápovědu, při chybě by měla být poskytnuta smysluplná

zpětná vazba v terminologii uživatele
- chybové zprávy by měly sdělit typ chyby a kde se chyba udála (oznámení "CHYBNÝ VSTUPNÍ ÚDAJ" nám nic neřekne)
- chybové zprávy by měly být konstruktivní a kdekoli je to možné, tam by měly napovědět, jak může být chyba opravena (levá zpráva je negativní, obviňuje uživatele z toho že udělal chybu, nepoužívá jazyk uživatele (patient id); pravá je pozitivní, říká že problém je v systému a poskytuje návod, jak chybu napravit)



Webové aplikace by navíc měly uživateli poskytovat odpovědi na tři hlavní otázky (Dix 1999):

* kde jsem? - tj. jakou aplikaci právě používám, na jakém místě jsem v hierarchii stránek
* co mohu dělat? - tj. jaké funkce jsou nyní dostupné
* kde jsem byl a kam jdu? - srozumitelná navigace

Další poznámky:

* barva by neměla být primárním nositelem významu, protože cca 10% populace je v různé míře barvoslepých
* zvukovým efektům se pokud možno vyhýbáme
* důležité je uvažovat požadavky na čas odpovědi (délku i variabilitu); pokud je čas delší než cca 2s, je důležité, aby uživatel viděl, že se něco děje (např. zobrazením "teploměru")
* je třeba počítat s tím, že uživatelé prakticky nikdy nečtou manuál; ovládání musí být intuitivní, nápověda snadno dostupná

Poznámka pro zajímavost (doporučení pro konkrétní GUI)

Pro konkrétní GUI obvykle existují doporučení výrobce (style guide) pro návrh aplikace s využitím poskytovaných prvků GUI. Doporučení bude obsahovat například pravidlo, že defaultní hodnota ve formuláři má být nejpravděpodobnější volba, případně nejméně nebezpečná volba; pokud je defaultní hodnota odvozena z jiných hodnot zadaných uživatelem, pak ty hodnoty, ze kterých odvozujeme další, mají být blízko začátku formuláře atd.

Jako příklad uvedu "Java Look and Feel Design Guideline", kterou najdete na adrese http://java.sun.com/products/jlf/ed2/book/index.html .

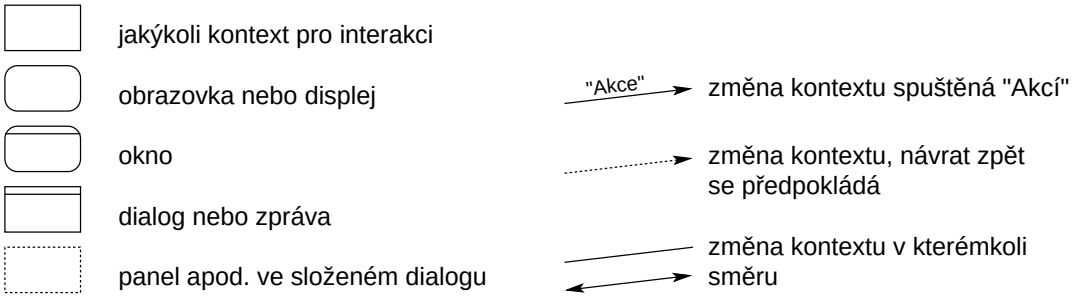
[]

Návrh UI
.....

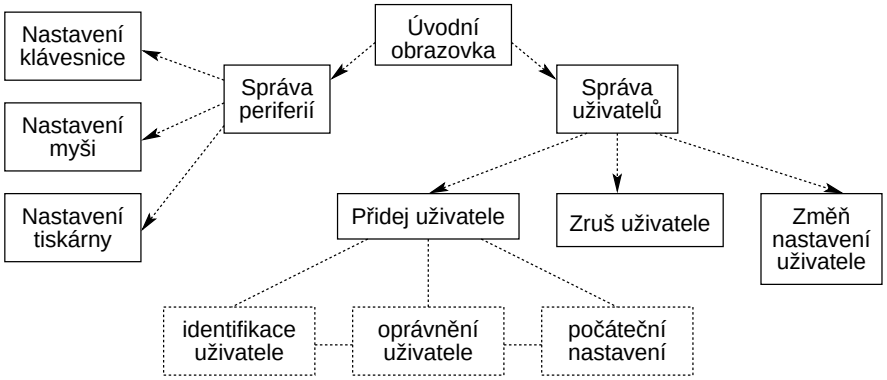
* můžeme vycházet ze scénáře, který se váže k případu použití
* procházíme scénář, identifikujeme hraniční objekty (podstatná jména) a akce (slovesa), týkající se komunikace uživatele se systémem
  - ptáme se: jaké informace bude uživatel potřebovat zadávat, měnit, prohlížet?
  - např. uživatel zadá {jméno} a {heslo}

* výhodné je začít na papíře - čistý papír je pracovní plocha (časem se může stát oknem aplikace, dialogovým oknem apod.)
  - pro každou pracovní plochu si poznamenáme "materiál" (data) a "nástroje" (aktivní prvky pracující s materiálem)
  - pracovní plochu pojmenujeme

* vytvoříme diagram procházení pracovními plochami (navigation map) -
  popisuje, jak budou uživatelé moci přejít z jedné pracovní plochy do jiné
 - pro znázornění můžeme používat následující notaci (Constantine 1998):



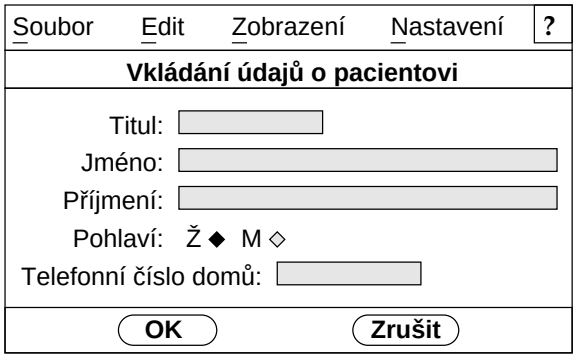
 - například částečný diagram programu pro správu systému by mohl vypadat
   následovně:



Poznámka: pokud byste chtěli ke stejnému účelu použít UML, můžete použít
např. stavový diagram - události popíšeme klávesou, příkazem z menu atd.

* abstraktní prototyp převedeme na vzhled obrazovek
 - typicky se začíná tím, že každou abstraktní komponentu
   implementujeme jako jeden standardní prvek GUI (widget)
 - pak se snažíme zjednodušit způsob práce nebo ušetřit místo na obrazovce

* pro návrh UI běžných systémů můžeme použít diagramy rozvržení oken
  (window layout diagram, viz také [Page-Jones: Základy OO návrhu v UML.
   Grada 2001], str. 168)
 - zobrazuje přibližně návrh okna, neobsahuje všechny kosmetické doplňky
 - můžeme začít hrubým návrhem prototypu pomocí papírového modelu UI,
   nakreslením obrázku na počítači, případně v nástroji pro vytváření oken
 - informace ve formulářích by měla být požadována v logické posloupnosti
   (např. oslovení, jméno, příjmení; nikoli jméno, oslovení, příjmení)



 - při návrhu vzhledu je nejvýhodnější evoluční vývoj (exploratory
   development), kde uživatel návrh vyhodnocuje nebo spoluvytváří

Vyhodnocení UI

* UI je obtížné vyhodnotit bez otestování na skutečných uživatelích
  - dotazníky - co si uživatelé o rozhraní myslí
  - pozorování uživatelů při práci, při pokusu vyřešit úlohu ”myslí nahlas”
  - snímání typického využití systému videokamerou
  - vložení kódu, který sbírá informace o nejpoužívanějších vlastnostech a nejčastějších chybách
* žádný ze způsobů nedetekuje všechny problémy UI

* dotazníky, otázky mohou být (1) jednoduché odpovědi typu ano/ne, (2) číselné odpovědi, (3) subjektivní oznámkování, (4) subjektivní procentuální odpověď; např.
  1. Jsou ikony srozumitelné? Pokud ne, které ikony jsou nejasné?
  2. Jak obtížné bylo naučit se základní práci se systémem? Ohodnoťte obtížnost na stupnici 1 až 5 (kde 5 je nejvyšší obtížnost).
  3. Kolik % fcí systému jste použila? (Všechny = 100%, žádné = 0%)
  4. Jak srozumitelné jsou chybové zprávy? ...

* pozorování a nahrávání, které vlastnosti používají, jaké chyby dělají; analýza nahrané relace dovoluje pozorovat, např. zda UI nevyžaduje příliš mnoho pohybů rukou (problém některých GUI je nutnost opakovaně přesouvat ruce mezi myší a klávesnicí - optimální je, pokud lze aplikaci ovládat pouze pomocí klávesnice)

* sběr statistik - zjištění nejčastějších operací => UI může být upraveno tak, aby tyto operace bylo možné zvolit nejrychleji

Architektonické styly

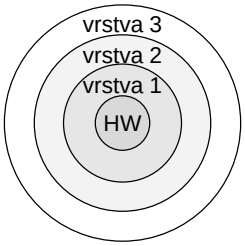
* architektonický styl (angl. architectural style, macro-architectural pattern) je znovupoužitelná abstrakce systému
  - v praxi se vyskytuje opakovaně v různých aplikacích
  - můžeme z něj vycházet při tvorbě vlastních aplikací
* vlastnosti jednotlivých stylů jsou známé z již existujících aplikací daného stylu (škálovatelnost, výkonnost, bezpečnost apod.)
  - styl slouží jako komunikační nástroj a základ pro managerské rozhodování (např. pro rozdělení do týmů, organizaci dokumentace projektu apod.)
  - styl je popsán:
    . množinou podsystémů a jejich typů (např. datové úložiště, proces, UI)
    . topologickým uspořádáním podsystémů
    . množinou sémantických omezení (např. datové úložiště nesmí měnit uložené hodnoty)
    . množinou interakčních mechanismů (volání podprogramu, událost, roura)

V dalším textu si ukážeme nejdůležitější architektonické styly.

Obecné struktury

Vrstvené systémy

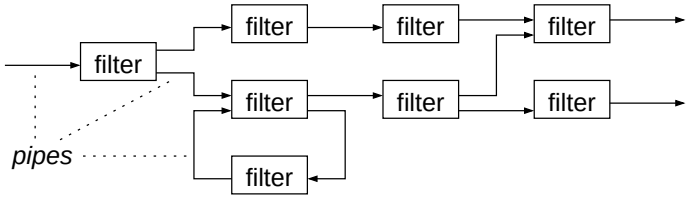
* systém strukturujeme do podsystémů, které tvoří vrstvy, vrstva používá pouze služby nižších vrstev
  - používají se, pokud fčnost může být rozdělena na část specifickou pro aplikaci a generickou část použitelnou pro mnoho aplikací
  - další důvody:
    . pokud části systému mají být nahraditelné
    . pokud je důležitá přenositelnost do různých OS nebo HW
    . pokud můžete využít již existující vrstvu (OS, komunikaci po síti...)
  - používá se v mnoha SW produktech, například některé operační systémy



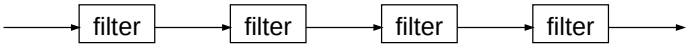
Styl dataflow
....

* používá se, pokud se můžeme na systém dívat jako na posloupnost transformací
  zpracovávajících vstup a produkujících výstup
* výhodou integrovatelnost - relativně jednoduché rozhraní mezi komponentami
* dva hlavní podstyly - ”roury a filtry” a ”dávkově sekvenční” architektura

* roury a filtry (pipe-and-filter architecture)
 - podsystémy nazýváme filtry, jsou propojené rourami, které přenášejí data
 - každý filtr pracuje nezávisle, pouze očekává data v určitém formátu a
   produkuje výstup v určeném formátu



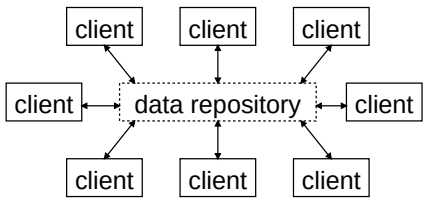
* dávkově sekvenční
 - pokud výše uvedená architektura degeneruje do jedné lineární posloupnosti
   transformací
 - vstupem dávka dat, aplikuje na ní posloupnost sekvenčních komponent (filtrů)



 - příkladem mohou být překladače programovacích jazyků, např. překladač
   jazyka C: zdrojový text nejprve zpracuje preprocesor, poté se provede
   lexikální a syntaktická analýza, optimalizace, nakonec generování kódu

Pasivní datové úložiště: styl repositář (repository)
....

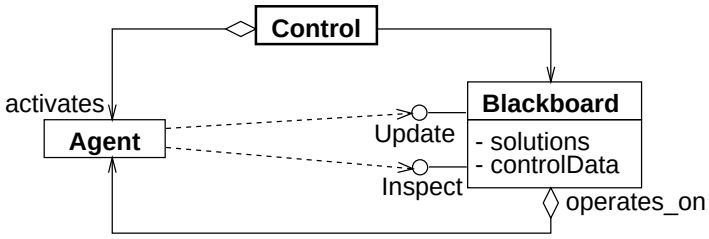
* centrem architektury je datové úložiště (databáze nebo soubor)
* ostatní podsystémy (klienti) k úložišti přistupují a čtou, přidávají, ruší
  nebo modifikují data



* tento styl se používá, pokud je požadováno uchování, výběr a správa velkého
  množství dat a pokud jsou data vhodně strukturovaná
* hlavní cíle:
 - integrovatelnost, klientské podsystémy pracují nezávisle
 - škálovatelnost (scalability) = možnost snadno přidávat nové klienty a data

Aktivní datové úložiště: styl tabule (blackboard)
.....

* množina agentů spolupracuje pomocí datového úložiště
 - úložiště je aktivní, posílá oznámení agentům o změně dat, která je zajímají
 - agent vyhodnotí obsah úložiště, případně vloží výsledek nebo částečné řešení
 - obrázek znázorňuje příklad uspořádání; agenti jsou aktivováni řídícím objektem/modulem, využívají rozhraní tabule pro přístup k datům



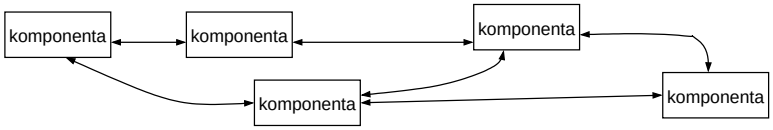
 - například systémy pro umělou inteligenci, rozpoznávání řeči apod.
 - tj. systémy, kde neznáme vhodné uzavřené (algoritmické) řešení problému, ale umíme řešit pomocí agentů (např. znalostních agentů), kteří k řešení přispívají

Distribuované systémy

Pokud systém můžeme strukturovat jako množinu volně vázaných podsystémů, podsystémy mohou běžet na nezávislých strojích propojených sítí.

Peer-to-peer
.....

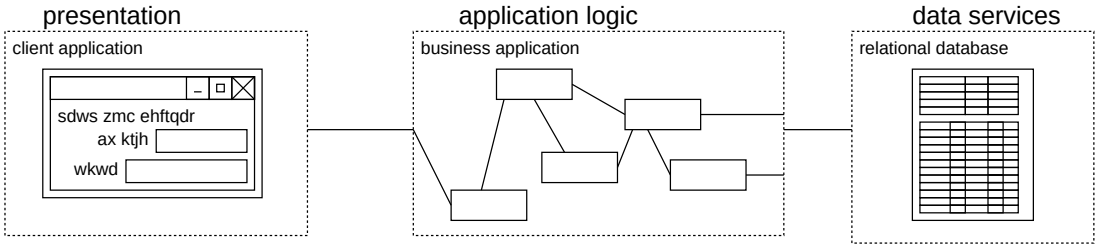
* komponenty spolu mohou navázat komunikaci - vyměňují si informace podle potřeby



Architektura klient/server
.....

* pokud úloha může být rozdělena na tvůrce požadavků (klient) a jejich vykonavatele (server)
* v systému je alespoň jeden klient a alespoň jeden server
* nejčastější podstyly stylu klient-server:
 - třívrstvá architektura (3-tier architecture)
 - tlustý klient (fat client)

* třívrstvá architektura (3-tier architecture)
 - funkčnost se rozděluje do 3 částí:
  . klienti - obsahují prezentační služby, obvykle jeden klient slouží jednomu uživateli
  . datové služby - obvykle jsou implementovány pomocí databázového serveru
  . aplikační logika - informace vytváří a modifikuje pomocí datových služeb, poskytuje informace klientům



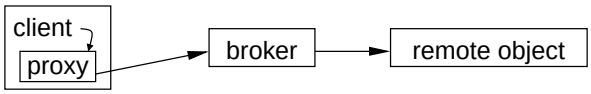
- v současné době nejoblíbenější, ”SW průmysl se neodvratně řítí tímto směrem” (citace z jedné přednášky)
- častá varianta: tenký klient - klientem je např. WWW prohlížeč

* tlustý klient (fat client)
- klient obsahuje prezentaci i aplikační logiku, využívá data z databáze

- oproti třívrstvé architektuře relativně snadný návrh, ale obtížná údržba

Broker

* cílem je, aby distribuovanost byla transparentní = objekt může volat metodu jiného objektu, aniž by věděl, že objekt není lokální

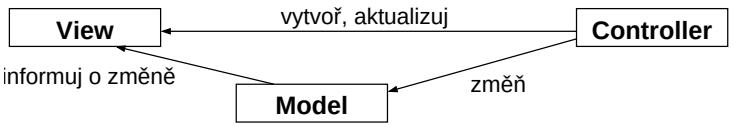


* proxy volá brokera, který zjistí, kde se nachází vzdálený objekt
* příklad: CORBA (Common Object Request Broker Architecture) - je to architektura+infrastruktura pro spolupráci aplikací prostřednictvím sítí

Interaktivní systémy

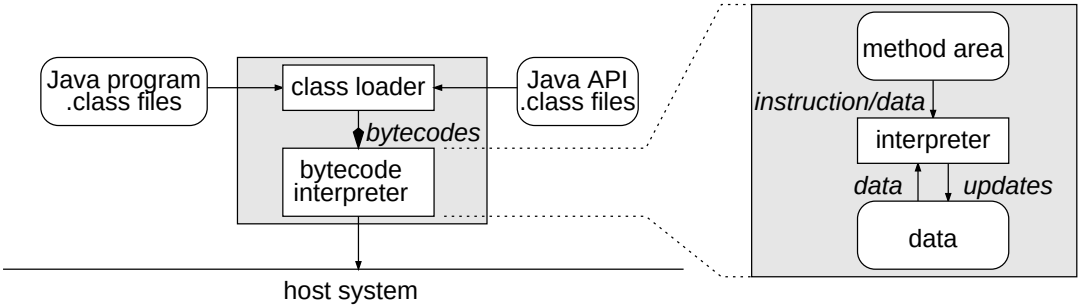
Architektura Model-View-Controller (MVC)

* architektura doporučovaná a široce používaná pro klasické i webové interaktivní aplikace



* odděluje funkční vrstvu aplikace (”Model”) od dvou aspektů uživatelského rozhraní (nazývaných ”View” a ”Controller”)
- Model = objekty, které budeme v aplikaci prohlížet a měnit (např. obchodní data)
- View (náhled) = zobrazení modelu; při změně obsahu modelu je vyvolána také změna zobrazení
- Controller = obsluhuje interakci uživatele s modelem
  . na základě změn modelu a vstupů uživatele (stisku kláves, výběru z menu)

```
    vybírá View, který se má zobrazit
  . typicky jeden Controller pro množinu příbuzných fcí
* architektura MVC byla již popsána (viz str. 79 přednášek)


Ostatní styly
-------------

Virtuální stroj
...............

* pokud je výpočet buď velmi abstraktní, nebo se předpokládá jeho běh na
  různých typech strojů
* příklady:
 - interprety, např. JVM
 - systémy založené na pravidlech, např. expertní systémy
 - procesory příkazových jazyků, např. /bin/sh
```




```
Co se výběru architektonického stylu aplikace týče, nejpříjemnější by bylo
mít katalog stylů (podobně jako existují katalogy pro návrhové vzory), kde by
bylo uvedeno ”pokud je problém podobný X, použijte styl Y”. Na rozdíl od
návrhových vzorů zatím vývoj tak daleko není...


Poznámka (návrh systému shora dolů)

 Alternativa metodičtějších přístupů (např. strukturovaných nebo objektově
 orientovaných metod) je neformální návrh systému shora dolů, kde vycházíme z
 architektonického návrhu:

 * architektonický návrh - identifikujeme a zdokumentujeme podsystémy a
   jejich vztahy
 * abstraktní specifikace podsystémů - pro každý podsystém vytvoříme
   abstraktní specifikaci podsystému a jeho omezení
 * návrh rozhraní podsystémů - pro každý podsystém navrhneme a
   dokumentujeme rozhraní (specifikace rozhraní musí být jednoznačná,
   aby bylo možné podsystém používat bez znalosti jeho vnitřního
   fungování)
 * návrh komponent - služby alokujeme komponentám, navrhneme rozhraní
   komponent
 * podrobně navrhneme a specifikujeme datové struktury
 * podrobně navrhneme a specifikujeme algoritmy
 * po návrhu následuje kódování

[]

Implementace a kódování
=======================

* vstupem implementace je návrh (design) SW systému
 - návrh může být na různé úrovni podrobnosti
 - obvykle popis modulů/tříd a jejich vztahů
 - v některých případech může být kostra programu automaticky vygenerována
   CASE nástrojem

* aktivity a postupy ve fázi implementace:
```

 - návrh podprogramů, algoritmů, datových struktur
 - psaní a dokumentace kódu, strukturované programování
 - testování a ladění modulů, optimalizace

Implementace ”zdola nahoru”

Návrh systému téměř vždy probíhá shora dolů, protože se snažíme velký problém (vytvářený SW systém) rozdělit do tak malých částí, abychom je dokázali postupně vyřešit (tj. nakonec naprogramovat). Tento přístup se často nazývá ”rozděl a panuj”.

Implementaci je naproti tomu většinou vhodné realizovat zdola nahoru:

* začínáme s podprogramy nejnižší úrovně, které ihned testujeme
 - pro testování podprogramu můžeme vytvořit jednoduchý hlavní program, který bude pouze volat testovaný podprogram s příslušnými parametry
* jakmile je podprogram funkční, je třeba ho zdokumentovat
 - v komentáři k podprogramu uvedeme co podprogram dělá, význam vstupních a výstupních argumentů a případné návratové hodnoty
* po vytvoření všech potřebných podprogramů nižší úrovně můžeme napsat podprogram vyšší úrovně a otestovat ho atd.
* nakonec napíšeme hlavní program

Vytvářením systému zdola nahoru postupně zvětšujeme vyjadřovací sílu jazyka, který pro psaní systému používáme. Ve chvíli, kdy nám začnou velké části systému pracovat, stane se to pro nás také motivací k dalšímu programátorskému úsilí.

Poznámka (další vylepšení)

 Výsledkem předcházejícího kroku je téměř funkční SW, který je většinou možné ještě podstatně vylepšit:

 * pro každý podprogram je vhodné se podívat, jakým způsobem je volán; často je výhodné podprogram rozšířit o činnost, která se provádí vždy před a po volání příslušného podprogramu
 * volají-li se dva nebo více podprogramů vždy ve stejném pořadí, je vhodné tyto podprogramy sdružit vytvořením dalšího podprogramu
 * vykonává-li podprogram několik typů činností, může být vhodné ho rozdělit na podprogramy pro jednotlivé činnosti.

[]

Vytváření podprogramů z pseudokódu
---

* vlastní kódování je vysoce individuální záležitost, obvykle nebývá podřízeno nějakému SW procesu

* jako příklad metodiky pro kódování jednotlivých podprogramů uvedu vytváření podprogramů z pseudokódu (PDL-to-code process, McConnell 1993)
 - vstupem je podrobný návrh v pseudokódu - ten převezmeme nebo vytvoříme
 - na základě pseudokódu vytváříme kód, z pseudokódu se stanou komentáře

Vytváření pseudokódu
.....

* pseudokód můžeme získat např. jako výstup strukturované analýzy, například:

```
     PROCES 3.2.1: ”Vydej stvrzenku”

     vytiskni hlavičku stvrzenky
     celková-hotovost = 0
     DO WHILE v tabulce ”peníze” jsou nezpracované záznamy
         do ”platba” načti záznam z tabulky ”peníze”
         vytiskni obsah záznamu ”platba”
         k celkové hotovosti ”celková-hotovost” přičti částku z ”platba”
     vytiskni ”celková-hotovost”
```

* pokud nemáme pseudokód, musíme ho navrhnout - postupujeme od obecného ke konkrétnímu
 - napíšeme komentář popisující účel podprogramu
  . ověříme, zda je činnost podprogramu dobře definována, zda zapadá do architektury a zda alespoň nepřímo odpovídá požadavkům
  . definujeme účel podprogramu - musí být definován tak podrobně, aby podprogram bylo možné implementovat
  . popíšeme vstupy a výstupy (včetně globálních proměnných, které vytvářený podprogram ovlivní), způsob obsluhy chyb
 - pokud je manipulace s daty podstatnou součástí činnosti, navrhneme hlavní datové struktury
 - podprogram pojmenujeme
 - vytvoříme vysokoúrovňový pseudokód podprogramu
  . pseudokód nebude používat prvky cílového jazyka, bude popisovat záměr
 - vytvořený pseudokód zkontrolujeme
  . kontrola a oprava pseudokódu je podstatně snazší než kontrola a oprava výsledného programu!

* je to opět iterativní proces, chyby je třeba opravovat ihned, jak je naleznete

Transformace do kódu
....

* pseudokód transformujeme do kódu takto:
 - pseudokód postupně zjemňujeme až na úroveň, kdy je snadné doplnit skutečný kód
 - pseudokód změníme v komentáře
 - napíšeme deklaraci podprogramu
 - pod každý komentář doplníme kód
 - kód zkontrolujeme, doplníme zbývající části (chybějící deklarace proměnných, ošetření chyb apod.)

* po ”ruční” kontrole kódu podprogram syntakticky zkontrolujeme překladem
 - při překladu povolíme všechna varování překladače, případná varování vyřešíme

Strukturované programování
----

* strukturované programování = používání řídících konstrukcí tak, aby každý blok kódu měl pouze jeden vstupní a jeden výstupní bod
 - jednoduché, hierarchické struktury pro řízení běhu
 - základní řídící konstrukce: sekvence, větvení, iterace

* základní myšlenka v Dijkstrově článku ”Go To Statement Considered Harmful” (1968, viz http://www.acm.org/classics/oct95/)
 - nestrukturované jazyky typu FORTRAN a BASIC používaly řízení pomocí příkazu GOTO (příkaz GOTO spustí provádění instrukcí od udaného návěští)
 - příklad v jazyce BASIC:

```
10 A=10
20 B=5
30 IF A<B THEN GOTO 60
40 PRINT ”A je větší než B”
50 STOP
60 PRINT ”A je menší než B”
70 END
```

 - problém: lze vytvářet libovolně komplikované konstrukce => ze zápisu programu s GOTO není snadno viditelné dynamické chování programu (běh procesu)
 - řešení: strukturované programování, příklad v jazyce Python:

```
a = 10
b = 5
```

```
        if a < b:
             print ”A je menší než B\n”
        else:
             print ”A je větší než B\n”
```

- strukturované programování zlepší produktivitu oproti nestrukturovanému až o 600%, čitelnost kódu zlepší o cca 30%
- strukturované programování podporují všechny současné procedurální programovací jazyky (obsahují strukturované řídící konstrukce typu ”if-then-else”, ”while”, ”repeat-until”, ”for”)

* řešení výjimečných situací pouze strukturovanými konstrukcemi může být zbytečně komplikované
- proto má většina jazyků více způsobů jak opustit řídící konstrukci (např. příkazy ”continue”, ”break”, ”return”, případně ”goto”)
- používat obezřetně pro řešení situací, jako je předčasné opuštění vnořených cyklů při chybě

Moderní jazyky vycházejí z myšlenek strukturovaného programování, ale posouvají je dále (objekty/třídy, výjimky, ...).

❇

KIV/ZSWI 2004/2005
Přednáška 11

Prototypování

* na první přednášce jsem zmiňoval dva druhy prototypů:
 - prototypy, ze kterých vyvineme konečný systém
  . vyvineme relativně jednoduchý systém, implementující nejdůležitější požadavky zákazníka
  . podle dalších požadavků přizpůsobujeme
  . měly by být vyvíjeny se stejnou kvalitou jako ostatní SW
 - throw-away prototypy - účelem je získání nebo ověření požadavků apod.
  . mají krátkou dobu života, je třeba je rychle vytvořit, snadno změnit

V dalším textu se budu zabývat pouze throw-away prototypy, tj. verzemi SW systému, které mají sloužit pro zjištění dalších informací o systému - nejčastěji v souvislosti se sběrem požadavků nebo v souvislosti s hledáním odpovědí na technické otázky (výkonnost apod.).

Tvorba prototypů by tedy v rámci přednášek logicky patřila ke sběru požadavků, ale z praktických důvodů ji uvádím zde.

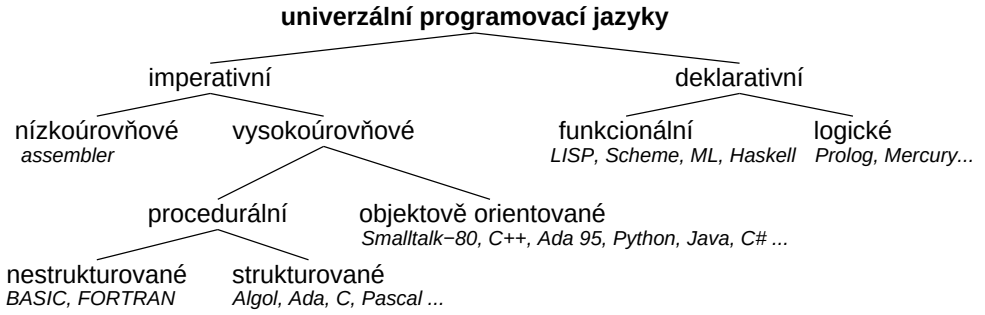
* experimenty ověřily intuitivně zřejmý předpoklad, že prototypy snižují množství problémů se specifikací požadavků (Boehm at al. 1984)

* prototypy se vytvářejí v následujících krocích:
 - definice účelu prototypu - například prototyp uživatelského rozhraní, prototyp demonstrující užitečnost systému zákazníkovi apod.
 - určení funkčnosti prototypu - co bude a co nebude prototyp obsahovat
  . při tvorbě throw-away prototypu obvykle rezignujeme na mimofunkční požadavky, jako je čas odpovědi, paměťová náročnost, spolehlivost (omezená kontrola chyb)
 - vytvoření prototypu
  . throw-away prototypy nemusejí být nutně spustitelné; užitečné (a levné) jsou i papírové modely uživatelského rozhraní apod.
 - vyhodnocení prototypu - nejdůležitější fáze, zde získáváme díky prototypu potřebné informace
  . pro otestování UI je třeba zvolit typického uživatele systému

Rychlé prototypování

* pro rychlé prototypování (rapid prototyping) se používají zejména:
 - dynamické vysokoúrovňové programovací jazyky
 - databázové jazyky
 - komponentově orientované programování

Vysokoúrovňové programovací jazyky



Programovací jazyky můžeme v zásadě rozdělit do následujících kategorií:

* imperativní - posloupnost příkazů mění stav programu, jsou odvozeny od von Neumannova modelu počítače
* funkcionální - výpočet je zapsán pomocí fcí, které vracejí hodnotu

* logické - programy jsou vyjádřeny pomocí fakt a jejich vztahů

* klasické procedurální jazyky jako Pascal, C/C++, Ada, Java atd.
 - vytváření spolehlivých a rychlých programů
 - deklarace datových typů => specializace fcí, neumožňuje ”náhodnou spolupráci”, nutí programátora provádět explicitní volby brzy ve vývojovém procesu

* pro prototypování se používají především dynamické vysokoúrovňové jazyky
 - obsahují silné mechanismy pro manipulaci dat, správa paměti v režii jazyka (tj. programátor nemusí řešit problémy při alokaci a dealokaci paměti, na rozdíl např. od C kde programátor musí paměť alokovat/uvolňovat explicitně pomocí malloc() a free())
 - dynamické typování (typy argumentů nebo proměnných se nedeklarují)
 - předpoklad ”je lepší mít 100 fcí pracujících nad jednou datovou strukturou než mít 10 fcí pracujících nad 10 datovými strukturami”
 - příklady jazyků: awk, Javascript, Lisp/Scheme, Haslell, Perl, Prolog, Python, Ruby, Smalltalk, Tcl, Visual Basic...

* příklad č. 1: Python
 - objektově orientovaný interpretovaný jazyk, možnost procedurálního programování
 - elegantní a snadno naučitelná syntaxe, sdružování příkazů pomocí odsazování

```
    def spocti(a, b):
        if a<b and a*b > b:
            return a
        else:
            return b
```

 - vestavěné vysokoúrovňové typy: seznam, slovník
 - knihovny s mnoha třídami usnadňujícími programování (např. regulární výrazy, komunikace po internetu, XML, GUI...)
 - Jython = v Javě implementovaný Python, dovoluje volání Pythonovského kódu z Javy a naopak (v Javě implementovaná část kódu může volat prototyp vytvořený v Pythonu)
 - nevýhody pro reálné aplikace: slabá správa paměti, pomalý (cca 3x pomalejší než Java)

* příklad č. 2: Haskell98
 - moderní funkcionální jazyk, tj. výpočet je prováděn vyhodnocováním funkcí
 - funkce jsou obvykle definovány množinou rovnic
 - levá strana výrazu obsahuje vzory, které se porovnávají se skutečnými argumenty
 - jako příklad - implementace algoritmu quicksort:

```
qsort []     = []
qsort (x:xs) = qsort less ++ [x] ++ qsort more
     where less = filter (<x)  xs
           more = filter (>=x) xs
```

* další jazyky pro některé typy prototypů:
 - Tcl - často prototypování grafických aplikací (Tk toolkit, který je dnes ale dostupný i z jazyků Python, GUILE atd.)
  . Tcl se většinou používá jako rozšiřovací jazyk pro aplikace v C nebo C++
  . nevýhoda: nemá dobře navržené datové struktury (do verze 8.0 pouze řetězce)
 - awk a Perl - orientovány především na zpracování textových souborů (vestavěné regulární výrazy apod.)
  . nevýhoda: Perl má problematickou syntaxi
 - Lisp (Common Lisp, Scheme) - funkcionální jazyk
  . hlavní datovou strukturou je seznam
  . minimální syntaxe
  . často se používá jako rozšiřovací jazyk aplikací (AutoLisp pro AutoCAD, GUILE pro volně šířené programy, elisp pro Emacs)
 - Prolog - logické programování, někdy simulace databází

* někdy se různé části prototypu vytvářejí v různých jazycích (pro danou část se volí nejvhodnější jazyk)

Databázové programování

* mezi aplikacemi zpracovávajícími data je velká podobnost
  - pro vstup a výstup obvykle množina formulářů nebo tabulek, zadaná data uložena do databáze
  - výběr dat z databáze, vytvoření výstupních sestav

* proto vznikly specializované jazyky pro manipulaci s databází, s nimi související nástroje pro definici UI
* pro nástroje + prostředí se používá pojem ”jazyky čtvrté generace” (fourth-generation languages, 4GLs)
* v 4GL prostředí typicky:
  - databázový dotazovací jazyk, dnes obvykle SQL
    . dotazy obvykle vygenerovány automaticky z formulářů vyplněných uživatelem
  - generátor UI
    . interaktivní definice formulářů pro vstup nebo zobrazování dat, jejich propojení, definice dovolených rozsahů vstupních hodnot
    . většina dnešních 4GL podporuje WWW formuláře
  - generátor výstupních sestav
    . pro definici a vytváření (tiskových) výstupů z informace obsažené v databázi

* nevýhoda: 4GL nejsou zatím nijak standardizované

Komponentově orientované prototypování

* komponentově orientované prototypování má obdobné výhody a nevýhody jako komponentově orientované programování:
  - nemusíme-li některé části prototypu navrhnout a implementovat, snížíme tím čas vývoje
  - na druhou stranu je často nutné přizpůsobit specifikaci tomu, jaké komponenty máme k dispozici

* extrémním případem je využití celých aplikací jako komponent
  - prototyp může být realizován např. jako objekt složeného dokumentu (text, část tabulky, zvukové soubory), které jsou udržovány různými aplikacemi (editor, spreadsheet, program pro přehrávání zvukových souborů)
  - nejpoužívanější mechanismus Microsoft OLE (Object Linking and Embedding)
  - výhoda: prototyp je vytvořen rychle
  - nevýhoda: pokud uživatelé nemají zkušenosti s použitými aplikacemi, může pro ně být matoucí funkčnost, která pro prototyp není zapotřebí

* prostředí pro vytváření prototypů obecně obsahuje:
  - komponenty
  - rámec pro sestavování komponent - poskytuje mechanismus řízení + mechanismus pro komunikaci
    . jeden příklad je prostředí tzv. skriptovacích jazyků (scripting languages), mezi které patří jazyky příkazových interpretů, Python, Tcl apod.
    . dalším příkladem jsou obecné rámce pro integraci komponent (CORBA, DCOM, JavaBeans)

Volba programátorských konvencí

Implementaci by měly předcházet minimálně následující kroky:

1. pochopení řešeného problému
2. návrh architektury systému
3. výběr vhodného programovacího jazyka
4. výběr programátorského prostředí, které poskytuje vhodné nástroje
5. volba programátorských konvencí (pokud se nejedná o jednorázový program)

Body (1) a (2) už byly popsány v předchozích přednáškách. Výběr programovacího jazyka a prostředí byl zmíněn v souvislosti s prototypováním.

Proto se budeme ještě zabývat bodem (5).

Motivace

Jedno ze základních pravidel zní: Zdrojový text programu musí být
srozumitelný pro lidi (ostatní členy týmu).

Jak poznamenává Fowler ve své knížce ”Refactoring”:

 Any fool can write code that a computer can understand. Good programmers
 write code that humans can understand.

* srozumitelnost důležitá zejména pro údržbu - co když je v programu nalezena
  chyba a původní programátor není k dispozici?
* pokud programátor nerozumí cizímu programu, může pro něj být jednodušší
  nesrozumitelný kód napsat znovu než ho převzít => snižuje produktivitu

Poznámka (kód pište pro ”průměrného programátora”)

 Při čtení vašeho kódu by se měl programátor cítit jako při čtení
 nejnudnějšího románu na světě. Funkce každého řádku by měla být zřejmá.
 Pokud kód bude nějak zacházet s proměnnou, měl by si čtenář říci: ”Mně bylo
 předem jasné, že uděláš přesně tohle!”

[]

* problém - pokud čtete kód vytvořený jinými programátory, je často obtížně
  srozumitelný kvůli jejich programátorskému stylu (formátování atd.)
 - programátorský styl (coding style) = soubor pravidel pro psaní zdrojových
   textů, týká se formátování, tvorby názvů, komentářů, předepsaného chování
   SW v určitých situacích (např. při chybě) atd.
 - čtení kódu vytvořeného ve stylu, na který nejste zvyklí, trvá vždy déle, než
   pokud styl znáte
 - nesprávný či nekonzistentní styl => chybná interpretace
 - proto je styl kódu, který budou číst další lidé, podstatný
   => tým nebo týmy by se měly shodnout na souboru pravidel pro psaní
      zdrojových textů sdíleném celým projektem
  . nedokonalý systém je lepší než žádný
  . vzájemná srozumitelnost přednější než preference jednoho autora
  . na druhou stranu existují studie porovnávající čitelnost některých stylů
 - pozitivní důsledek jednotného stylu: kód bude pro všechny zúčastněné čitelnější

* programátorský styl se týká především
 - odsazování bloků
 - zalamování řádků, mezer a závorek
 - jmenných konvencí
 - komentářů

Poznámka (standardní konvence pro jazyk Java)

 Pokud je to možné, měl by být jednotný styl týmu založený na standardních
 konvencích daného programovacího jazyka. Např. pro jazyk Java jsou
 standardní konvence autorů jazyka zveřejněné na

  http://java.sun.com/docs/codeconv/

 Konvence pro jazyk Java oproti námi uváděným oblastem navíc pokrývají
 pojmenování souborů a jejich organizaci.

[]

Odsazování bloků
....

* správné odsazení musí ukazovat logiku programu
 - cíl: samodokumentující kód
 - důsledky nesprávného či nekonzistentního odsazení: chybná interpretace,

obtížně udržovatelný kód; například následující kód bude jinak interpretovat člověk a jinak počítač:

```
for (int i = 1; i <= 10; i++)
   leftboot = left[i];
   left[i]  = right[i];
   right[i] = leftboot;
```

* doporučuje se psát pouze jeden příkaz na řádku
* odsazování bloků tak, aby bylo vidět, které příkazy jsou v bloku
 - čisté odsazování: lze v Adě, protože každá řídící struktura má svůj ukončovač:

```
začátek_bloku         while Color = Red loop
   příkaz1               příkaz1;
   příkaz2               příkaz2;
konec_bloku           end loop;
```

 - simulované čisté odsazování: jako kdyby ”begin” a ”end” byly součástí řídící struktury
  . styl ”Kernighan & Ritchie” v C
  . de facto standard v C, C++ a Javě, v Pascalu se příliš nepoužívá

```
xxxxxx begin           while (c) {
   příkaz1                příkaz1;
   příkaz2                příkaz2;
end                    }
```

 - begin-end hranice: za hranici bloku považujeme ”begin” a ”end”
  . podle toho zda ”begin” a ”end” považujeme za součást bloku 3 varianty
  . varianta 1 se používá v C i Pascalu, varianta 2 v C (styl GNU), varianta 3 v Pascalu

```
              1.               2.               3.
xxxxxx        while (!done)    while (!done)    while not done
begin         {                  {                begin
   příkaz1       příkaz1;           příkaz1;         příkaz1;
   příkaz2       příkaz2;           příkaz2;         příkaz1;
end           }                  }                end
```

* formátování jednopříkazových bloků
 - mělo by být konzistentní s formátováním delších bloků
 - v zásadě následující možnosti, každá má své výhody i nevýhody:

```
1.             2.               3.             4.
if (expr)      if (expr) {      if (expr)      if (expr) cmd;
   cmd;           cmd;         {
               }                   cmd;
                                }
```

 - ve skupinových projektech se doporučuje styl 2, protože konzistentní s odsazováním podle K&R, a pokud budete přidávat příkazy za if, nemůžete zapomenout přidat ”{” a ”}”

* někdy máte potřebu použít ”speciální” formátování; téměř vždy je to příznak špatně navržené metody/podprogramu nebo rozhraní
* například rozhraní pro jazyk C++ pro práci s Okny:

```
   HRESULT hr = NějakéJménoFunkce(
             0,                  // Rezervováno, parametr musí být 0
             0,                  // Nedělej něco divného
             THIS|THAT|ANOTHER,  // Nastav nějaké speciální příznaky
             BSTR(0),            // Rezervováno, musí vypadat takhle
             &something);        // Objekt, který má být vyplněn hodnotou
```

 - problémem výše uvedeného rozhraní je příliš mnoho parametrů

Poznámka pro zajímavost (automatické formátování)

 * některé týmy nechají programátory používat styl odsazování, jaký kdo chce, a použijí např. ”indent -kr -i8 -l80” před přidáním kódu do repositáře

```
   projektu
 * automatické formátování ale občas vede k méně přehlednému kódu, než je kód
   formátovaný ručně

 * program indent(1) v Linuxu
  - formátuje zdrojové texty jazyka C - mezery, odsazení, umístění
    programových závorek, komentáře
  - předdefinované styly:
   . Kernighan & Ritchie (-kr)
   . Berkeley (-orig)
   . GNU (-gnu)
  - nejdůležitější parametry:
    -iN         ... odsazení příkazů v bloku o N mezer
    -lN         ... délka řádku bude N znaků
    -br         ... složené závorky budou na stejné řádce jako ”if”, ”while” atd.
    -bli0 -bliN ... složené závorky na další řádce odsazené o N znaků
    -diN        ... odsazení jmen proměnných od typu v deklaracích na sloupec N

 * obdobné programy existují i pro jiné jazyky, např. astyle pro C, C++ a
   Javu, Jalopy pro jazyk Java

[]


Zalamování a vkládání prázdných řádků
.....................................

* řádek by neměl být delší než je obvyklá šířka obrazovky (např. 80 znaků pro
  znakový terminál)
* dlouhé řádky je nutné zalomit na logickém místě
 - související věci ponechat na stejném řádku
 - na pokračovacím řádku odsazení podle úrovně ”vnoření” zalamovaného místa

  fd = open(name, O_WRONLY|O_CREAT|O_TRUNC|O_APPEND,
            S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH);

 - některé konvence doporučují zalamovat před operátorem (např. konvence pro
   jazyk Java, GNU konvence), jiné za ním (např. Delphi; je třeba se řídit
   vybranou konvencí)

  if (queue == NULL && foo_this_is_long && bar > win (x, y, z)
      && remaining_condition) ...

  if (queue == NULL && foo_this_is_long && bar > win (x, y, z) &&
      remaining_condition) ...

* prázdným řádkem je vhodné od sebe oddělit logické celky:
 - jednotlivé sekce v programu, podprogramu, třídě nebo metodě (např. lokální
   proměnné od prvního příkazu apod.)
 - skupinu souvisejících příkazů
 - jednotlivé podprogramy nebo metody
 - komentáře

Poznámka (k délce podprogramů)

 Již dlouhá léta se traduje, že podprogramy by neměly přesahovat cca jednu až
 dvě obrazovky (cca 50 řádků). Studie však prokázaly, že do cca 200 řádek
 kódu samotná délka podprogramu neovlivňuje negativně chybovost ani
 srozumitelnost. Podprogram by tedy měl být dlouhý přesně tak, jak je
 zapotřebí.

[]


Používání mezer a závorek
.........................

* K&R doporučují kolem operátorů obvykle zapsat mezeru, např.
```

```
  x = x * (y + 1);

* uvnitř výrazů se doporučuje vkládat závorky a mezery pro lepší
  srozumitelnost
 - tedy nikoli: x = a + b % c * d / e;
     ale např.: x = a + (((b % c) * d) / e);
        nikoli: z = x / 2 + 3 * y;
     ale např.: z = x/2 + 3*y;

* za čárkou a středníkem má následovat mezera, např.

  foobar (x, y, z);  // nikoli: foobar (x,y,z);


Jmenné konvence
...............

* dobré názvy jsou nejdůležitější složkou programátorského stylu
 - příklad chybného pojmenování: x = x - fee(x1, x) + tt;  // co to asi může znamenat?
 - příklad lepšího pojmenování: kredit = kredit - poplatek(zakaznik, kredit) + urok;

* názvy mají dodávat kódu význam
 - z čím větší části programu je název viditelný, tím pečlivěji ho musíme
   zvolit
 - proměnnou, metodu, třídu atd. bychom měli označit srozumitelným názvem,
   který popisuje význam entity, kterou reprezentuje
 - například pocetSedadel, pocet_mist_k_stani, jmenoOlympijskehoTymu apod.
  . výše uvedené názvy jsou samy o sobě srozumitelné
  . některá jména jsou ale příliš dlouhá na to, aby byla praktická (z výše
    uvedených poslední dvě)
  . výzkum ukázal, že psychologické optimum je cca 8-20 znaků
 - názvy delší než 20 znaků je vhodné konzistentně zkrátit, např. použít
   srozumitelné prefixy/postfixy (jako jsou anglické Sum, Max, Min, Ptr)

* existují další konvence pro pojmenování řídících proměnných cyklů,
  logických proměnných, konstant, tříd a metod apod.
 - řídící proměnné cyklů - pokud jsou cykly krátké, používají se často
   jednoznakové názvy jako i, j, k; např. v jazyce C:

     for (i = 0; i < BUFFER_SIZE; i++) ...

 - pokud je smyčka delší než několik řádek, má i zde smysl použít popisné
   jméno, např. v Pascalu:

    for TeamIndex := 1 to TeamCount do begin
       for EventIndex := 1 to EventCount [ TeamIndex ] do ...

* logické proměnné - měly by mít pozitivní jméno podmínky
 - např. česky chyba, konec, nalezeno atd.
 - nebo anglicky done, error, found, success (případně isDone, isError,
   isFound, isSuccess)

* výčty a pojmenované konstanty - často velkými písmeny, např.
  VELIKOST_BUFFERU nebo BUFFER_SIZE (v C, Javě), případně
  Okraj.VYSTŘEDIT nebo BorderLayout.CENTER (v Javě)

* dočasné proměnné (temporary variables, často názvy ”tmp”, ”tem”)
 - dočasná proměnná = lokální proměnná, která se uvnitř jednoho podprogramu
   používá postupně pro několik různých účelů
 - jejich výskyt je často varující příznak toho, že programátor problému
   ještě zcela nerozumí
 - dočasným proměnným bychom se měli spíše vyhýbat, pro každý účel bychom
   měli vytvořit samostatnou lokální proměnnou se smysluplným názvem
 - kromě zvýšení čitelnosti to usnadní optimalizaci dobrým překladačům

* v objektově orientovaných jazycích, které rozlišují malá a velká písmena, se
  často používá konvence pocházející z jazyka Smalltalk:
 - jméno třídy a jméno konstruktoru začíná velkým písmenem, např. Point,
   Rectangle, Image, ImagePanel apod.
```

- jméno metody, proměnné atd. malým písmenem, např. metody addMouseListener(), paintComponent(), proměnné point, rectWidth, imageHeight apod.

Poznámka (neformální jazykově závislé konvence - C)

Pro konkrétní programovací jazyky vznikly další konvence. Pokud budete programovat v jazyce C, brzy zjistíte, že (až na výjimky) jsou názvy proměnných a funkcí tvořeny malými písmeny a podtržítkem (”pocet_sedadel”, ”pocet_mist_k_stani” apod.), pro lokální proměnné se používají krátké názvy (”c” a ”ch” pro znaky, ”p” pro ukazatel, ”s” pro řetězec) apod.

[]

Poznámka pro zajímavost (Maďarská notace pro pojmenování identifikátorů)

* maďarská notace - vznik ve firmě Microsoft (Simonyi asi 1984), dnešní rozšíření zejména díky rozhraní MS Windows
* viz závěr poznámky (mínusů je více než plusů, tj. maďarskou notaci NEDOPORUČUJI používat, pokud nemusíte)

* k identifikátoru přidává prefix popisující funkční typ identifikátoru
* název ”maďarská notace” jednak protože identifikátor vypadá na první pohled nesrozumitelně a také protože Simonyi pochází z Maďarska

* základní myšlenka pojmenovat hodnoty jejich funkčním typem, aby programátor nemusel název proměnné a fce dlouho vymýšlet
  - ”funkční typ” dvou proměnných je stejný, pokud je nad oběma možné provést stejné operace (tj. nebere se v úvahu pouze reprezentace, ale také význam)
  - například pokud je operace setPosition(x, y) v pořádku, zatímco setPosition(y, x) je nesmysl, nemají ”celá čísla” x a y stejný funkční typ
  - funkční typy jsou pojmenovány krátkými indikátory, které si volí programátor; neměly by to být obecné názvy, protože s nimi jsou potíže (např. ”color” je obecný název, ale v aplikaci můžeme mít víc funkčních typů pro uchovávání barev; proto raději názvy jako ”co”, ”cl”, ”kl” apod.)
  - například funkční typy pro textový procesor by mohly být: ”wn” (okno), ”row” (řádek textu), ”fon” (font), ”f” (boolovská hodnota - flag), ”ch” (znak - character) apod.
* ze základních funkčních typů můžeme konstruovat další typy pomocí prefixů
  - standardní prefixy:

prefix	*anglicky*	*význam*
a	array	pole
c	count	počet, např. počet znaků, záznamů apod.
d	difference	rozdíl mezi dvěma proměnnými stejného typu
e	element of an array	prvek pole
g	global variable	globální proměnná
h	handle	popisovač, např. popisovač souboru apod.
i	index	index do pole
m	module-level	proměnná modulu
p	pointer	ukazatel

  - např. ”arow” = pole prvků typu ”row”
  - datové struktury mají vlastní typy (název typu by neměl být odvozen z prvků datové struktury, protože reprezentace typu se může snadno změnit, aniž by se změnil jeho význam)
  - datové typy jsou pojmenovány stejnými zkratkami jako funkční typy, tj. v programu bychom našli deklarace jako:

```
WN wnMain;
ROW rowFirst;
```

* pravidla pro pojmenování hodnot (proměnných): funkční typ volitelně následovaný kvalifikátorem
  - např. v názvu ”rowFirst” je ”row” typ a ”First” je kvalifikátor; typ by měl být od kvalifikátoru vhodným způsobem oddělen, např. v C velké písmeno

```
  - příklady identifikátorů v maďarské notaci:

        ch          datová struktura pro znak
        cch         počet znaků
        ach         pole znaků
        achInsert   pole znaků pro vložení
        echInsert   prvek pole znaků pro vložení
        hwn         popisovač okna (typ "okno" jsme si pojmenovali "wn", viz výše)

 * výhody
  - standardní konvence (to je užitečné samo o sobě)
  - snadná tvorba názvů

 * nevýhody:
  - hlavní nevýhoda - vytvořená jména nejsou vždy informativní (např. "hwn"
    neříká, o jaký typ okna se jedná - nevím, zda je to hlavní okno, help, menu apod.)
  - spojuje význam dat s jejich reprezentací
  - mnoho uživatelů maďarské notace používá místo funkčních typů základní
    typy programovacích jazyků (int, long apod.) - což je zbytečné (překladač
    základní typ dat zná)

[]



Komentáře
.........

* doplňují, co v kódu není vidět (sémantika, odkazy, zdroje informací,
  záměr/účel, nestandardní operace)
 - komentování modulů, proměnných, podprogramů, bloků, řádek kódu

* komentování modulů
 - každý modul by měl začínat komentářem, který stručně popíše účel modulu
   (případně jiného zdrojového souboru aplikace)
 - další část komentáře obvykle popisuje copyright apod.
 - např.:

  /* farm.c -- obsahuje funkce pro zakládání a rušení farem.
   *
   * Copyright 2003  Lukáš Petrlík  <luki@kiv.zcu.cz>
   */

* komentování deklarací proměnných, zejména globálních; např.

   wrkstate *worker;         /* Seznam obsahující stav všech pracovníků. */
   farminfo *farm;           /* Seznam obsahující popis všech farem. */
   int nfarms;               /* Celkový počet farem spravovaných aplikací. */

* komentování podprogramů
 - činnost podprogramu
 - argumenty, význam hodnot, které mohou nabývat, účel argumentů
 - význam případné návratové hodnoty

* komentování bloků kódu apod.
 - komentář by měl být odsazen stejně jako komentovaný kód
 - před nebo za komentářem je vhodné vynechat prázdný řádek, abychom komentář
   snadno našli

        foobar(buff);

        /*
         * POSIX.1 specifies that if the O_APPEND flag is set, no intervening
         * file modification operation is allowed (see POSIX.1:1996, section
         * 6.4.2.2, lines 226-228).
         */
        write(fd, buff, strlen(buff));

* komentáře nepřehánět, vysvětlit především co a proč program dělá
```

Poznámka (nástroje pro kontrolu programátorských konvencí)

Pro kontrolu programátorských konvencí existují nástroje. Například pro programy v Javě existuje nástroj Checkstyle, který umí kontrolovat všechny zde uvedené konvence, včetně jmenných konvencí.

[]

Nástroje pro dokumentaci: javadoc
---

* javadoc je nástroj pro jazyk Java
 - generuje HTML dokumentaci včetně křížových referencí na základě speciálně formátovaných komentářů (”doc comments”)
 - komentář začíná /** a končí */ (pokud více řádků, mohou začínat hvězdičkou)
 - komentáře musí být umístěny před dokumentovanou třídou, metodou apod.
 - v komentářích možno použít HTML (kromě H1-H3)
 - klíčová slova pro odkazy apod. na úrovni modulu a třídy

* na úrovni modulu a třídy:

```
   @author jméno       // bere se v úvahu pouze pokud zadán parametr -author
   @version verze      // bere se v úvahu pouze při zadaném parametru -version
   @see JménoTřídy     // generuje ”See Also: JménoTřídy”
```

* na úrovni atributů a metod

```
   @param název popis                // popis parametru
   @return popis_hodnoty             // popis návratové hodnoty
   @exception modul.JménoTřídy popis // totéž co @throws - generuje ”Throws”
   @deprecated náhradní_řešení       // lze také na úrovni třídy nebo rozhraní
```

* příklad:

```
  /**
   * Toto je komentář pro třídu <code>Hello</code>.
   *
   * @author  Lukáš Petrlík
   * @version 1.0 (25.4.2003)
   * @see     java.lang.Object
   */
  public class Hello {
      /** Příklad komentáře atributu. */
      int x = 0;

      /** Popis metody <code>main</code>. */
      public static void main(String[] args) {
          System.out.println(”Hello World!”);
      }
  }
```

* popis javadoc naleznete na http://java.sun.com/products/jdk/javadoc/
* podobné nástroje jsou dostupné i pro další jazyky nebo jsou jazykově nezávislé, např. nástroje Doc++, RoboDoc, atd.
 - přehled najdete na http://www.codeassets.com/doc_tools.htm

Poznámka (nástroj Checkstyle)

Výše zmíněný nástroj Checkstyle umí kontrolovat i Javadoc komentáře.

[]

Optimalizace programu
---

* program by měl být tak efektivní, jak je od něj požadováno, nikoli tak, jak je to technicky možné
 - přílišné zaměření na výkonnost zhoršuje čitelnost a udržovatelnost kódu

```
 - optimalizace je drahá činnost, tj. je třeba důkladně zvážit, zda je nutná
 - při optimalizaci je riziko zanesení chyb do funkčního kódu

* na výkonnost se můžeme zaměřit na dvou úrovních: strategické a taktické
 - strategie:
  . nejde změnit/vyladit design?
  . můžeme použít jiný algoritmus, změnit datové struktury?
 - taktika - optimalizace kódu:
  . cca 20% programu konzumuje 80% času (Boehm 1987, podobně Bentley 1988)
  . nutné optimalizovat pouze kritická místa programu
  . kritická místa dnes není možné určit bez měření, protože moderní
    překladače provádějí poměrně agresivní optimalizaci

* nástroje - profilery - zjištění času stráveného v jednotlivých částech
 - např. volně šířené nástroje gcc a gprof:

   $ gcc -pg program.c      # přeloží program.c a vloží kód pro profilování
   $ ./a.out                # přeložený program spustíme, vytváří gmon.out
   $ gprof a.out gmon.out   # gprof vypíše statistiky (doby strávené ve fcích apod.)

* nejčastější zdroje neefektivity:
 - přístup k souborům
 - podprogramy pro formátovaný tisk
 - operace v pohyblivé řádové čárce
 - stránkování (bude popsáno v předmětu KIV/ZOS)
 - volání služeb operačního systému (taktéž viz KIV/ZOS).
```

❇

KIV/ZSWI 2004/2005
Přednáška 12

Verifikace a validace

* verifikace = ověření, zda produkt dané fáze vývoje SW odpovídá konceptuálnímu modelu (např. zda kód odpovídá návrhu apod.)
 - tj. odpověď na otázku: vytvářím produkt správně?
* validace = vyhodnocení SW na konci procesu vývoje SW, abychom zajistili splnění požadavků na SW
 - tj. odpověď na otázku: vytvářím správný produkt?

Verifikace a validace je široké téma, my se budeme zabývat pouze následujícími třemi oblastmi:

* automatická statická analýza - používá se nejčastěji pro kontrolu zdrojových textů SW systému, případně kontrola modelů apod.
* inspekce - ruční kontrola artefaktů SW procesu, typicky prováděná skupinou 3 až 5 lidí
* testování - spouštění programu s takovými daty, abychom v programu odhalili defekty

Základní termíny, které budu dále používat:

* omyl (error) - chybná úvaha nebo překlep vývojáře, vede k jednomu nebo více defektům
* defekt (fault, bug, defect) - rozdíl mezi chybným programem a jeho správnou verzí
* symptom (symptom, failure, run-time fault) - pozorovatelné chybné chování programu; defekt se při konkrétním běhu může projevit žádným, jedním nebo více symptomy

Automatická statická analýza

* používají se programy pro automatickou kontrolu modelů nebo zdrojových textů
 - například překladač jazyka Java obsahuje silnou typovou kontrolu
 - pro slabě typované jazyky (např. C) se používají statické analyzátory (code checkers)
  . nejstarší ze známých statických analyzátorů je program lint(1), který byl součástí historických systémů UNIX
  . v současnosti se často používá volně šířený lclint(1)
  . detekuje neinicializované proměnné, odchylky od standardů apod.
  . měl by se použít vždy před inspekcemi

* další kontroly - mnoho programů používá pro detekci podezřelých míst heuristiky
 - nevýhoda: pokud zdrojový text neodpovídá heuristikám zabudovaným v programu, mohou tyto programy produkovat falešná chybová hlášení
 - příklad program: Jlint pro jazyk Java

Inspekce a procházení programů

* používají se při přezkoumávání (review) DSP, detailního návrhu, kódu (inspekce kódu se provádějí před testováním programu, inspekcím by měla předcházet statická analýza)
* zahrnují čtení dokumentu nebo programu týmem např. 3 nebo 4 lidí (jeden z nich autor) s cílem nalézt defekty (nikoli jejich řešení)
* budu popisovat pro testování kódu, testování ostatních artefaktů SW procesu je analogické

* výhody:
 - bývají poměrně efektivní (typicky najdou 30% až 70% defektů detailního návrhu a kódování)
 - úsilí bývá přibližně poloviční oproti ekvivalentnímu otestování na

počítači (na druhou stranu - pokud máme testy již připravené, mohou běžet automaticky)
- cena opravy defektu bývá nižší než při testování na počítači (protože je známá přesná příčina defektu, zatímco testování na počítači najde pouze symptomy)
- nalézá jiné typy defektů než klasické testování, tj. je s ním komplementární (je vhodné provádět obojí)

* nevýhody:
- pro maximální efektivitu je třeba, aby s nimi tým získal zkušenost

Faganovské inspekce kódu
.....

* zahrnují procedury a techniky pro skupinové čtení kódu (Fagan 1976)
* poprvé využity ve firmě IBM

* tým provádějící inspekci se obvykle skládá ze 4 osob
- moderátor - jeho úkolem je rozdistribuovat materiály pro schůzku, naplánovat schůzku, vést jí, zaznamenávat defekty, zajistit aby byly opraveny
  . moderátor by měl být schopný programátor, ale ne autor programu; nemusí mít detailní znalosti programu, jehož inspekce se provádí
- programátor - autor programu
- dalšími členy obvykle návrhář (pokud je odlišný od programátora) a specialista na testování

* před schůzkou moderátor (např. několik dní předem) rozdistribuuje program a specifikaci návrhu programu, účastníci se mají s materiálem seznámit

* inspekce probíhají podle následujícího scénáře:
- optimální doba inspekce je asi 90 až 120 min, měla by probíhat bez přerušení
1. na schůzce je programátor požádán o vysvětlení logiky programu příkaz po příkazu
   - moderátor je zodpovědný za to, že se účastníci zaměří na vyhledávání defektů, nikoli na jejich řešení
   - během proslovu vznikají otázky, usilující o určení, zda se v kódu nacházejí defekty
   - zkušenost ukazuje, že velkou část defektů najde programátor během výkladu (jinými slovy: samotné čtení kódu před posluchači je efektivní technika pro hledání defektů)
2. program je analyzován vzhledem k seznamu obvyklých programátorských chyb (seznam se vytváří v průběhu předchozích inspekcí)
3. po skončení schůzky dostane programátor seznam defektů
   - pokud je nalezeno více defektů nebo pokud některý defekt vyžaduje podstatný zásah do programu, může se domluvit nová inspekce po opravě programu
   - defekty jsou analyzovány a kategorizovány, použijeme je pro zpřesnění seznamu obvyklých programátorských chyb použitých v bodě (2)

* při většině inspekcí se projde cca 150 příkazů za hodinu
* kromě nalezení defektů je vedlejším efektem zpětná vazba týkající se programátorského stylu, výběru algoritmů a programovacích technik
* identifikace částí, které obsahují více defektů
- defekty se vyskytují ve shlucích, pravděpodobnost existence dalších defektů v dané sekci programu (např. podprogramu) je přímo úměrná počtu defektů v příslušné sekci již nalezených
- pokud jsou v některé sekci nalezeny defekty, měli bychom se na ní více zaměřit, např. při testování na počítači

Poznámka pro zajímavost (Internet Explorer)

V souvislosti v výše uvedeným je zajímavé si přečíst následující (cit. z http://www.zive.cz, článek z 11.2.2004):

Podle britského šéfa bezpečnosti společnosti Microsoft je Internet Explorer nejbezpečnější dostupný internetový prohlížeč. Toto tvrzení je založeno na počtu chyb, které již byly odstraněny. K prohlášení došlo po vydání bezpečnostní záplaty z minulého pondělí, která však přinesla problémy. ...

[]

* aby inspekce fungovaly, musí k nim mít všichni účastníci správný přístup
 - pokud programátor chápe inspekci jako útok na svou osobu a brání se, bude inspekce nutně neefektivní
 - naopak funguje, pokud bude chápat jako pomoc ke zlepšení kvality svého kódu

Příklad obvyklých programátorských chyb (pro jazyk C):

* data
 - je proměnná inicializována?
 - jsou odkazy do pole v rámci definovaných mezí pole?
 - nenastává při indexování pole chyba off-by-one?
 - ukazuje ukazatel na alokovanou paměť?
 - pokud čteme záznam ze souboru, má proměnná správný typ?

* chyby výpočtu
 - jsou v kódu výpočty se smíšenými typy (např. sčítání float a int)?
  . často zdrojem chyb
 - je do kratší hodnoty (např. int) přiřazována hodnota s delší reprezentací?
 - je možné přetečení nebo podtečení během výpočtu?
 - může nastat případ, že dělitel je 0?
 - jaké jsou důsledky nepřesností reálné aritmetiky?
 - atd.

* řízení toku
* rozhraní
* vstup a výstup
* ostatní

Procházení kódu (walkthroughs)
....

* podobně jako inspekce je technika detekce defektů pomocí skupinového čtení, v podrobnostech se liší

* schůzka 3 až 5 lidí, trvá 1 až 2 hodiny, také nemá být přerušena
 - role moderátora - podobná jako v případě inspekcí
 - sekretář - zaznamenává všechny nalezené defekty
 - tester
 - programátor - autor kódu
 - role ostatních členů týmu není ustálená, doporučuje se např. zkušený programátor, začínající programátor (má zatím nezkalený pohled), osoba, která bude provádět údržbu apod.

* stejně jako v případě inspekcí by měli dostat materiál několik dní předem
 - členové týmu si hrají na počítač: tester přijde na schůzku s malým počtem papírových testovacích případů - vstupy a očekávané výstupy programu nebo podprogramu
 - tým provádí testovací případ, stav programu (hodnota proměnných) zaznamenává na tabuli nebo na papír
 - v případě nejasností se ptá programátora na logiku programu a na předpoklady
  . většina defektů je nalezena otázkami, nikoli testovacími případy
 - následuje obdoba bodu 3. z popisu inspekcí - tj. programátor dostane seznam defektů, opraví...

* podobně jako v případě inspekcí je podstatný přístup
 - tým by měl hodnotit program, nikoli toho, kdo program napsal
 - na defekty nehledět jako na neschopnost programátora, ale chápat je jako nutný důsledek doposud nedokonalých metod programování

Testování
=========

* testování = spouštění programu se záměrem najít v něm defekty (tj. snažíme se, aby se projevily symptomy případných defektů)

Black box a white box testování

* existují dva základní způsoby testování - ”black box” a ”white box” testování

* black box testování (používají se také názvy: functional, data-driven, input/output driven testing)
  - tester na program pohlíží jako na černou skříňku s danou specifikací, vnitřní struktura a vnitřní funkce programu ho nezajímají
  - hledá případy, ve kterých se program nechová podle specifikace
  - pro nalezení všech defektů by bylo nutné otestovat program se všemi možnými vstupy (platnými i neplatnými), což je prakticky nemožné
    . např. překladač jazyka C bychom museli otestovat se všemi platnými i neplatnými programy
  - víme, že úplné otestování programu je nemožné - jak ale maximalizovat počet defektů nalezený konečným počtem testovacích případů?
    . k programu už nemůžeme přistupovat čistě jako k černé skříňce, ale musíme učinit nějaké rozumné předpoklady o jeho vnitřním chování

* white box testování (také: glass-box, clear-box, logic-driven testing)
  - testovací data se odvozují z programové logiky
  - pro úplné otestování programu bychom potřebovali pomocí testovacích případů otestovat všechny možné logické cesty v programu (analogie otestování programu se všemi možnými vstupy, viz výše)

  - má dva zásadní problémy:
    . počet logických cest je i v malých programech příliš velký - například fragment programu

```
for (i=0; i<100; i++) {
        if (podmínka)
                příkaz1;
        else
                příkaz2;
}
```

      má za předpokladu nezávislosti podmínek 2^100 logických cest, kterými může být vykonán (ve skutečnosti podmínky nebudou nezávislé, takže cest bude méně)
    . i po otestování všech logických cest mohou v programu zůstat nenalezené defekty, protože některé logické cesty mohou chybět a protože nemusejí být nalezeny defekty citlivé na data, např.

```
if ((a - b) < epsilon) ...  // místo: if (abs(a - b) < epsilon) ...
```

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

Návrh testovacích případů

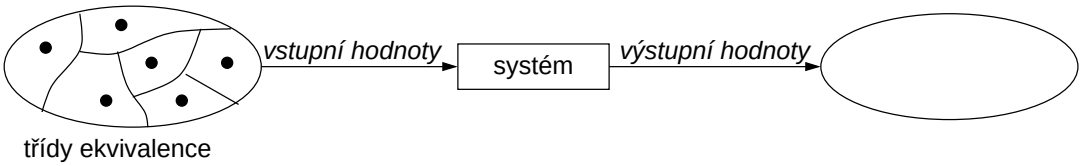
* už jsme viděli, že úplné otestování programu není možné
  - proto je pro testování velmi podstatný návrh efektivních testovacích případů
  - tj. klademe si otázku - jaká podmnožina všech testovacích případů má největší pravděpodobnost nalézt většinu defektů?

* první nápad - náhodně vybraná podmnožina všech možných vstupů
  - pravděpodobně jedna z nejhorších možností, protože má malou pravděpodobnost být optimální podmnožinou nebo být alespoň blízká optimální podmnožině

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

Rozdělení vstupů do ekvivalentních tříd
.....

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



* z těchto úvah je odvozena metodologie pro black-box testování známá jako equivalence partitioning = rozdělení do tříd ekvivalence
 - nejprve identifikujeme třídy ekvivalence
 - pak definujeme testovací případy

* identifikace tříd ekvivalence
 - vezmeme každou vstupní podmínku (často větu nebo frázi z DSP) a podle ní rozdělíme množinu všech vstupních hodnot do dvou nebo více podmnožin
  . existují dva typy tříd ekvivalence - platné (reprezentující platné vstupy) a neplatné (reprezentující chybné vstupní hodnoty)
 - rozdělení do tříd ekvivalence je heuristický proces, můžeme využít následujících doporučení:
  . pokud vstupní podmínka specifikuje interval hodnot (např. rok může být mezi 2000 a 2100), pak máme jednu platnou třídu ekvivalence (hodnoty 2000 až 2100) a dvě neplatné třídy ekvivalence (hodnoty < 2000, hodnoty > 2100)
  . pokud vstupní podmínka specifikuje množinu vstupních hodnot a pokud lze předpokládat, že každá z nich bude obsluhována jinak (např. ”vlak”, ”autobus”), bude jedna platná třída ekvivalence pro každý prvek množiny; přidáme jednu neplatnou třídu ekvivalence pro další prvek množiny (např. ”letadlo”)
  . pokud vstupní podmínka specifikuje situaci, která ”musí nastat”, např. první znak identifikátoru musí být písmeno, bude jedna platná třída ekvivalence (je písmeno) a jedna neplatná třída ekvivalence (není písmeno)
  . pokud je důvod předpokládat, že prvky nějaké třídy nejsou obsluhovány stejně, rozdělte třídu do menších tříd ekvivalence

* definice testovacích případů
 - pro každou neplatnou třídu ekvivalence vytvoříme samostatný testovací případ (to je nutné, abychom otestovali každou podmínku kontrolující neplatný vstup); testovací případy budou typicky obsahovat:
  . příliš málo dat nebo žádná data
  . příliš mnoho dat
  . neplatná data (např. negativní počet zaměstnanců)
 - dokud jsou platné třídy ekvivalence nepokryté testovacími případy, vytvoříme testovací případ pokrývající co nejvíce platných tříd ekvivalence; testovací případy budou typicky obsahovat:
  . nominální případy = běžné nebo očekávané hodnoty
  . minimální normální konfiguraci (např. jediný zaměstnanec)
  . maximální normální konfiguraci (pokud ji umíme určit)

* je výhodné testovat hraniční hodnoty tříd ekvivalence vstupních hodnot,
 - například pokud je platný vstup -1.0 až +1.0, pak vytvoříme testovací případy pro vstupy -1.0, +1.0, -1.0001, +1.0001
 - stejně otestovat hranice výstupních hodnot

Příklad (bankomat)

Například SW bankomatu bychom mohli otestovat pomocí následujících testovacích případů:

1. Vadná karta, konec.
2. Platná karta, chybné PIN, konec.
3. Platná karta, platné PIN, konec.
4. Výběr platné částky, dotaz na zůstatek.
5. Výběr neplatné částky.
6. Neplatný dotaz na zůstatek.

[]

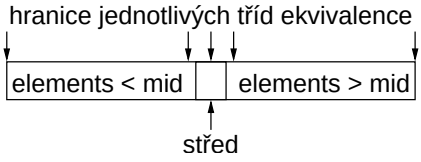
White-box testování
---

* white-box testování = využíváme znalost implementace
  - obvykle se používá pro testování relativně malých částí programu, jako jsou podprogramy v modulu, metody třídy - tj. testování jednotek
  - pokud jsou moduly integrovány do systému, složitost narůstá tak, že jsou strukturální techniky prakticky neproveditelné; některé proveditelné techniky uvedeme dále
  - se zvyšováním rozsahu projektu testy jednotek zabírají menší podíl na celkovém času vývoje (podle literatury od 35% pro malé systémy až po cca 8% pro velké systémy)

* pokud známe strukturu implementace, můžeme pro testování použít opět třídy ekvivalence a testovat běžné a hraniční podmínky se znalostí kódu
* ilustrovat budu na základě binárního vyhledávání v jazyce Java:

```
class BinSearch {

    // binární vyhledávání
    //  key       - klíč
    //  elemArray - seřazené pole prvků, ve kterém se vyhledává
    //  r         - výsledek, obsahuje r.index a boolovskou hodnotu
    //              r.found, která je true pokud byl klíč v poli

    public static void search(int key, int elemArray[], Result r) {
        int bottom = 0;
        int top = elemArray.length - 1;
        int mid;

        r.found = false;                        // n1
        r.index = -1;                           // n2
        while (bottom <= top) {                 // n3
            mid = (top + bottom) / 2;           // n4
            if (elemArray[mid] == key) {        // n5
                r.index = mid;                  // n6
                r.found = true;                 // n7
                return;                         // n8
            } else {                            // n9
                if (elemArray[mid] < key)       // n10
                    bottom = mid + 1;           // n11
                else                            // n12
                    top = mid - 1;              // n13
            } // konec ”if”                     // n14
        } // konec ”while”                      // n15
    } // konec ”search()”

} // konec ”BinSearch”
```

- z kódu můžeme zjistit, že binární vyhledávání rozděluje vyhledávací prostor do třech částí (prostřední prvek pole elemArray[mid], začátek pole, konec pole), každá část tvoří třídu ekvivalence



- pro testování bychom mohli použít například následující testovací případy:

vstupní pole (elemArray) (elemArray)	klíč (key)	výstup (found, index)	třída ekvivalence (pole, prvek)
17	17	true, 0	jedna hodnota, výskyt v poli
17	0	false, -1	jedna hodnota, není v poli
17, 21, 23, 29	17	true, 0	více hodnot, první prvek
9, 16, 18, 30, 31, 41, 45	45	true, 6	více hodnot, poslední prvek
17, 18, 21, 23, 29, 38, 41	23	true, 3	více hodnot, prostřední prvek
17, 18, 21, 23, 29, 33, 38	21	true, 2	více hodnot, soused prostředního
12, 18, 21, 23, 32	23	true, 3	více hodnot, soused prostředního
21, 23, 29, 33, 38	25	false, -1	více hodnot, není v poli

- testování bychom provedli vytvořením ovladače (např. metody s názvem ”test”, případně ”main”), který bude volat testovaný podprogram nebo metodu
- zadané vstupy (případně i výstupy) mohou být zakódovány v ovladači, převzaty z příkazového řádku, zadány uživatelem, přečteny ze souboru apod.
- například:

```
public static void main(String[] args)
{
    int[] values = { 21, 23, 29, 33, 38 }; // inicializace
    Result r = new Result();

    search(25, values, r);                 // vlastní test
    if (r.index != -1 || r.found != false) // odpovídá předpokladu?
        System.out.println(”Test 8 FAILED”);
}
```

* vytváření testů si vynucuje dobrý návrh - má-li být kód testovatelný, je třeba, aby moduly/třídy byly volně vázané, nevyžadovaly složitou inicializaci apod.

Poznámka (nástroje JUnit a JUnitDoclet)

Jako pomůcka pro testování se v jazyce Java často používá knihovna JUnit, která poskytuje nástroje pro spouštění testovacích případů a umí graficky i textově zobrazit výsledky testů.

Další pomocný nástroj JUnitDoclet (je implementován jako plug-in do nástroje JavaDoc) umí vygenerovat kostru testovacích případů pro JUnit.

[]

* z kódu můžeme určit také další typ hraniční podmínky, který nastává, pokud dochází ke kombinaci vstupních hodnot
  - například pokud podprogram hodnoty násobí, testovací případy mohou zahrnovat dvě velká kladná čísla, dvě velká záporná čísla apod.

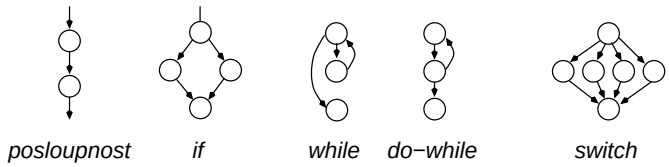
Pokrytí kódu testovacími případy
.....

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

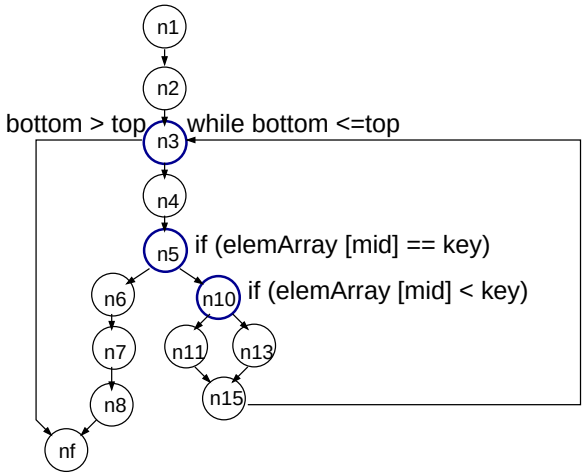
* je dobrým kritériem alespoň jedno vykonání každého příkazu? (statement coverage, pokrytí všech příkazů)

- například mějme příkaz: if (a>1) and (b=0) then x = x/a;
- oba příkazy (if a přiřazovací příkaz) by bylo možné pokrýt jediným testovacím případem (a=2, b=0)
- příklady defektů, které by testem zůstaly neodhaleny: místo ”and” má být ”or”, místo ”a>1” má být ”a>=1” nebo ”a>0” apod.
- podmínka pokrytí všech příkazů je nutná, ale nikoli postačující

* kritérium pokrytí zesílíme - je dobrým kritériem pokrytí všech rozhodovacích příkazů tak, aby byly vykonány všechny jejich větve? (branch coverage)
- musíme vytvořit tolik testovacích případů, aby se v každém příkazu ”if” vykonala alespoň jednou větev při podmínce ”false” a alespoň jednou větev při podmínce ”true” atd.
  . pro složitější podprogramy se vyplatí modelovat cestu podprogramem pomocí orientovaného grafu, popisujícího možný tok řízení v podprogramu



  . uzly reprezentují příkaz nebo část příkazu (přiřazovací příkazy, volání podprogramů, případně podmínky rozhodovacích příkazů)
  . každý pár uzlů, pro který je možný přenos řízení, je spojen hranou
  . příklad - graf toku řízení pro dříve uvedený příklad na binární vyhledávání v Javě:



  . nezávislá cesta = musí procházet alespoň jednu novou hranu v grafu, tj. provést jednu nebo více nových podmínek
- kvůli ”patologickým případům” (podprogram neobsahuje rozhodování, má více vstupních bodů apod.) připojujeme ještě podmínku pokrytí všech příkazů
- jako kritérium by postačovalo, pokud bychom měli vždy jedinou podmínku v rozhodovacím příkazu
- pro kód ”if (a>1) and (b=0) then x = x/a” je stále ještě slabé kritérium: pokud otestujeme pomocí (a=2, b=0) a (a=2, b=1), neodhaleny by zůstaly defekty jako místo ”a>1” má být ”a>=1”

* rozumným kritériem je pokrytí všech kombinací podmínek v rozhodovacím příkazu (multiple condition coverage)
- oproti branch coverage navíc vyžaduje, aby byly otestovány všechny kombinace hodnot logických operandů v podmínce
- kód ”if (a>1) and (b=0) then x = x/a” můžeme otestovat čtyřmi testovacími případy:
  . (a=2, b=0) => obě podmínky jsou true, větev ”then” se provede
  . (a=2, b=1) => true, false, větev ”then” se neprovede
  . (a=1, b=0) => false, true, větev ”then” se neprovede
  . (a=1, b=1) => false, false, větev ”then” se neprovede
- testovací případy není vhodné generovat strojově z kódu, ale můžeme si

pomoci nástrojem generujícím ”nápady na testovací případy” z výrazů (např. pro Javu nástroj ”multi”)

* další obvyklé kritérium je pokrytí smyček - vyžaduje 3 testovací případy:
 - tělo smyčky se nevykoná, tj. při prvním vyhodnocení bude test ”false”
 - tělo smyčky se vykoná právě jednou, tj. při prvním vyhodnocení bude test ”true”, poté ”false”
 - tělo smyčky se vykoná více než jednou, tj. test bude ”true” nejméně dvakrát

* další možné kritérium - all-du-path
 - jedna cesta pro každou definici-použití:
  . pokud je proměnná definována v jednom příkazu a použita v jiném, cesta by měla procházet oběma příkazy

* pro určení pokrytí velkých sad testů spouštěných na celé systémy jsou užitečné tyto podmínky:
 - pokrytí podprogramů - zda byl podprogram vyvolán alespoň jednou
 - pokrytí volání - zda každý podprogram volal všechny podprogramy, které může volat

* testy prováděné bez měřené pokrytí kódu typicky pokrývají pouze 55% kódu (Grady 1993)
 - pro zjištění, které cesty byly vykonány, se používají dynamické analyzátory programu
  . při překladu je ke každému příkazu připojen kód, který počítá, kolikrát byl daný příkaz vykonán (tzv. instrumentace)
  . po běhu můžeme zjistit, které části programu nebyly pokryty příslušným testovacím případem
  . příklad nástroje: GCT (Generic Coverage Tool) volně šířený nástroj pro jazyk C, viz ftp://ftp.cs.uiuc.edu/pub/testing/gct.files; příklad části výstupu: ”program.c”, line 9: loop one time: 10, many times: 2.

Strategie testování

* testování by mělo být předem naplánováno spolu s celým SW procesem
 - pro zvýšení efektivity testování je vhodné provádět také inspekce kódu
 - testování by mělo začínat na úrovni jednotek (procedur, tříd - funkce každé jednotky se ověří samostatně) a postupovat směrem k větším celkům (podsystémům, celému systému)
 - testování jednotek provádí obvykle ten, kdo danou část napsal; testování větších celků provádí nebo alespoň řídí specialista - tester (rozsáhlejší software testuje nezávislá testovací skupina)

Poznámka (psychologický problém testování)

Pro většinu programátorů je obtížné testovat vlastní programy, protože jejich zájem je spíše ukázat, že jejich program neobsahuje defekty a pracuje podle požadavků zákazníka.

Jinými slovy, protože testování je destruktivní proces, pro programátora může být velmi obtížné přepnout se z kódování programu (proces konstrukce) na testování (destrukci). Program také může obsahovat chyby způsobené nepochopením specifikace, které programátor nemůže sám odhalit.

Proto je vhodné přenechat testování sestaveného programu někomu jinému, např. nezávislému testovacímu týmu. Programátor je však vždycky zodpovědný za otestování jednotek, které vytvořil (procedur, funkcí, tříd).

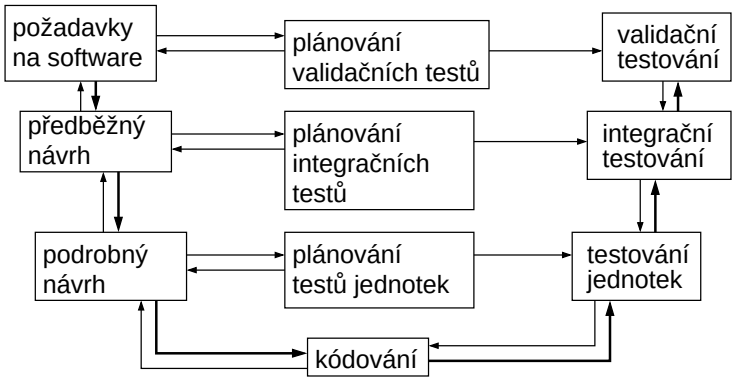
[]

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

* testování jednotek (unit testing) - testujeme nejmenší jednotky návrhu, např. procedury nebo funkce; pro testování můžeme používat white-box techniky
* integrační testování (integration testing) - sestavujeme software, spolu s

  tím testujeme defekty týkající se rozhraní mezi jednotlivými částmi
* validační testování (validation testing) - po integraci se zaměřujeme na
  funkce viditelné uživatelem
* testování systému (system testing) - pokud SW je pouze jednou součástí
  většího celku, účelem je otestovat celek; např. zátěžové testování,
  zotavení po závadě apod.

Testování by mělo být plánováno v souvislosti s aktivitami konstrukce SW:



Poznámka pro zajímavost (název ”testování jednotek”)

 Název ”testování jednotek” (unit testing) pochází z výroby hardwaru, kde
 znamená testování (měření) jednotlivých součástek (jednotek) před jejich
 sestavením do komponent, které se otestují a sestaví se z nich celý
 přístroj, který se opět otestuje.

[]

Testování jednotek
----

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

* pro objektově-orientovaný software se za jednotku považuje třída
 - třídy jako samostatné komponenty jsou obvykle rozsáhlejší než samostatné
   podprogramy
* při testování třídy bychom měli provést:
 - samostatné otestování každé metody
  . některé metody lze testovat až po předchozím vyvolání jiných metod, např.
    po inicializaci objektu
  . pokud používáme dědičnost, musíme testovat i všechny zděděné operace
    (mohou obsahovat předpoklady o dalších operacích a atributech, které ale
    mohly být potomkem změněny; obdobně musíme znovu otestovat potomky při
    změně rodiče)
 - testovat nastavení všech atributů objektu, dotaz na všechny atributy
 - testovat průchod všemi stavy objektu, případně simulace všech událostí,
   které způsobují změnu stavu objektu
  . pokud jsme vytvořili stavový diagram objektu, můžeme z něj určit
    posloupnost přechodů, které chceme testovat, a najít posloupnost
    událostí, které ji způsobí

Integrační testování
----

* po otestování individuálních komponent musíme komponenty integrovat =
  sestavit do částečného nebo úplného systému
* výsledek musíme otestovat na problémy, které vznikají interakcí komponent

* ”big bang” (velký třesk) - po otestování jednotlivých modulů je z nich v jediném kroku sestavena aplikace
 - použitelné pouze pro malé programy
 - pro větší systémy nejméně efektivní způsob integrace = vysoká pravděpodobnost neúspěchu

* hlavním problémem je lokalizace defektů, protože vztahy mezi komponentami mohou být značně složité
 - proto se často pro integraci a testování používá inkrementální přístup
 - nejprve integrujeme minimální konfiguraci systému, otestujeme
 - k systému přidáváme inkrementy, po každém přidání systém otestujeme
 - pokud nastaly problémy, budou pravděpodobně (ale ne nutně) způsobeny přidáním posledního inkrementu

* ve skutečnosti nebude tak jednoduché, protože některé vlastnosti budou rozptýleny do několika komponent, vlastnost můžeme otestovat až po integraci těchto komponent
  => při plánování testů je třeba počítat s časovým plánem na dokončení modulů
* pokud má důležitý modul neočekávané problémy, může se tím zdržet celá integrace (programátor řeší problém, zatímco všichni ostatní na něj čekají)

Testování rozhraní
....

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

* uvedu pouze obecná pravidla (volně podle Sommerville 2001):
 - v testovaném kódu najděte všechna volání externích komponent
 - navrhněte množinu testů tak, aby externí komponenty byly volány s parametry, které jsou extrémy jejich rozsahu (např. prázdný řetězec, dlouhý řetězec, který by mohl způsobit přetečení apod.)
 - navrhněte testy, které by měly způsobit neúspěch externí komponenty (zde jsou často chybné předpoklady)
 - v systémech s předáváním zpráv použijte zátěžové testování (viz dále)
 - pokud spolu komponenty komunikují prostřednictvím databáze, sdílené paměti apod., navrhněte testy, ve kterých se bude lišit pořadí aktivace komponent; testy mohou odhalit implicitní předpoklady programátora o pořadí, v jakém pořadí budou sdílená data produkována a konzumována

* mnoho defektů rozhraní odhalí statické testy
 - např. silná typová kontrola v překladačích jazyka Java
 - pro slabě typované jazyky (např. C) by se před integračním testováním měly použít statické analyzátory (code checkers)

* některé inspekce se mohou zaměřit také na rozhraní komponent a jejich předpokládané chování

Validační testování
----

* začíná tam, kde končí integrační testování
* testujeme, zda SW splňuje požadavky uživatele

* akceptační testování = zadavatel určí, zda produkt splňuje zadání
 - testuje se na reálných datech

* pro generické produkty není většinou možné vykonat akceptační testování u každého zákazníka, proto alfa a beta testování
 - alfa testy: na pracovišti, kde se SW vyvíjí (známé prostředí)
  . testuje uživatel, vývojoví pracovníci ho sledují a zaznamenávají problémy
 - beta testy: testují vybraní uživatelé ve svém prostředí (vývojářům neznámém)
  . defekty ohlášené uživateli jsou opraveny => finální produkt

❇

KIV/ZSWI 2004/2005
Přednáška 13

Testování systému

* účelem otestovat celý systém, jehož je SW součástí
* mívají různý účel, např. otestovat vlastnosti jako je výkonnost, kompatibilita, bezpečnost, instalovatelnost, spolehlivost apod.

Testování výkonnosti

* v mnoha typech systémů je nepřípustné, aby SW nesplňoval požadavky na výkonnost (zejména v řídících systémech)
* v takovém případě by se výkonnost měla testovat ve všech krocích včetně jednotkových testů
* pomocné procedury monitorují dobu vykonání apod.

Zátěžové testování

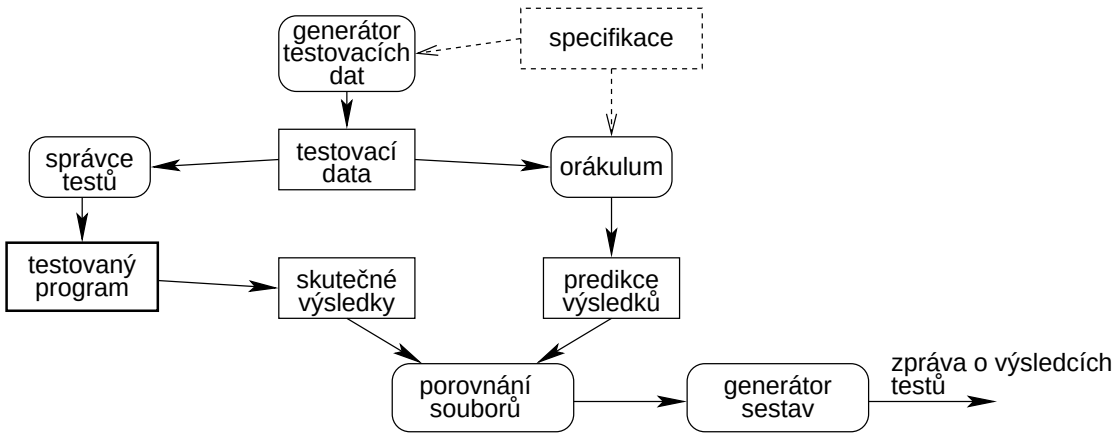
* obvykle se používají testy, kde se zátěž postupně zvyšuje, dokud není výkonnost systému neakceptovatelná nebo dokud systém nehavaruje
  - zátěž = množství dat, frekvence požadavků, data, která jsou extrémně náročná na zpracování
  - je vhodné určit části kódu, které mohou být problematické při velké zátěži, zátěžové testy navrhnout tak, aby pokrývaly především tyto části kódu
  - ověření, zda havárie systému nepoškodí data apod.
  - může odhalit některé defekty, které se normálně neprojeví
  - důležité zejména u internetových aplikací, v distribuovaných systémech apod., kde se vysoce zatížené systémy mohou zahltit, protože si vyměňují čím dál více koordinačních dat, čímž se opět zvyšuje zátěž systému atd.

Testování zotavení systému po havárii

* mnoho systémů se musí být schopno zotavit po havárii v předepsaném čase, jinak hrozí značné finanční ztráty
  - při testování zotavení systému způsobujeme různé havárie systému a ověřujeme, zda se systém zotavil správně a v časovém limitu

Nástroje pro testování SW

* v rozsáhlých systémech může cena testování dosáhnout 50% ceny vývoje, mohou existovat stovky až tisíce testovacích případů
  - proto potřeba automatizace testů
  - automatizace testů také snižuje cenu změn - programátoři se nemusejí tolik obávat, že změnou zanesou do kódu defekt, protože testy by defekt (s určitou pravděpodobností) odhalily
  - obvyklé je následující uspořádání:



* jednotlivé programy mají následující fce:
 - správce testů (test manager) - řídí běh testů
 - generátor testovacích dat (test data generator) - generuje testovací vstupní data pro testovaný SW
 - orákulum (oracle) - generuje předpokládané výstupní hodnoty
  . orákulum lze vyvinout jako nový program (obsahující podmnožinu funkčnosti, co nejméně pracná implementace)
  . často lze využít prototyp SW, předchozí verze SW, SW vytvořený konkurencí apod.
  . pokud se jako orákulum používá předchozí verze SW, používá se název regresivní testování (regression tests = porovnáváme výsledky staré a nové verze, rozdíly znamenají potenciální problém v nové verzi)
 - program pro porovnání souborů (file comparator) - porovná výsledky skutečného běhu s předpokládanými hodnotami vygenerovanými orákulem; často lze použít univerzální programy jako cmp(1) a diff(1) v UNIXu apod.
 - generátor zpráv (report generator) - umožňuje definovat a generovat zprávy o výsledcích testů

Poznámka (regresivní vs. progresivní testování)

 Proces testování má svou progresivní i regresivní fázi. Progresivní fáze testování přidává a testuje nové fce (nově přidané nebo modifikované moduly a jejich rozhraní s již integrovanými moduly). Regresivní fáze testuje důsledky změn v již integrovaných částech.

[]

* jako jednoduchou ukázku uvedu část příkazového souboru pro příkazový interpret (shell) v UNIXu, který provádí testování aplikace:

```
test-mkdata > data.test                  # Vytvoří testovací data ”data.test”.
test-oracle < data.test > result1.test   # Orákulum předpoví výsledky ”result1.test”.
aplikace    < data.test > result2.test   # Testujeme aplikaci.
if cmp result1.test result2.test; then   # Porovnáme skutečné a předpovězené výsledky.
   echo Test 1: PASSED                   # Soubory stejné -> test prošel.
else
   echo Test 1: FAILED                   # Soubory rozdílné -> test neprošel.
fi
```

* nástroje pro zachycení a pozdější přehrání testů (capture-replay tool)
 - informace protékají SW systémem - na vhodné místo toku dat aplikací vložíme nástroj, který tekoucí data zaznamená
 - tester spustí/vykoná a zaznamená testovací případy
 - později může zaznamenané testovací případy spustit znovu (regresivní testování)
 - existují komerční capture-replay nástroje pro zachycení/přehrání stisků kláves a pohybů myši, obsahující i nástroje pro zachycení a porovnání výstupů na obrazovku - používané pro GUI aplikace
  . pro většinu ostatních účelů (testování zapouzdřených systémů apod.) je většinou nutné vytvořit si vlastní SW pro podporu testování

* kolik defektů lze očekávat a kolik z nich lze najít testováním?
 - podle kvality vývoje lze očekávat asi 10 až 50 defektů na 1000 řádek kódu před testováním
  . např. Microsoft Application Division 10-20 defektů/1000 řádek kódu (Moore 1992)
 - testy najdou obvykle méně než 60% defektů, proto je vhodné je kombinovat s inspekcemi
 - někteří autoři uvádějí, že defekty jsou častěji v testovacích případech než v testovaném kódu (McConnell 1993)
 - pro kritické systémy je vhodné používat kombinaci formálních metod vývoje, inspekcí a testování (např. pro Cleanroom metodiku v průměru 2.3 defektů/1000 řádek kódu, pro některé systémy až 0 defektů)

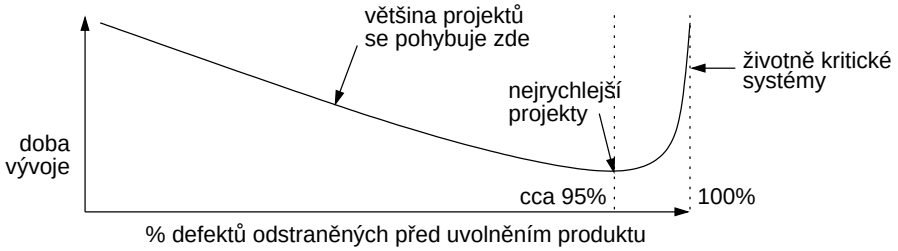
Poznámka (kvalita SW a rychlost vývoje)

 Většina managerů se snaží zkrátit dobu vývoje omezením času věnovanému inspekcím návrhu a kódu, testování apod. Zejména testování bývá často

obětováno, protože je na konci vývojového cyklu (tj. blízko deadline).

Podle dostupných studií (shrnutých např. v DeMarco a Lister 1987, McConnell 1996 aj.) tento chybný přístup celkovou dobu vývoje naopak prodlužuje. Jinými slovy, kvalitnější produkt bude dříve dokončen. Pokud obětujeme kvalitu, vývoj se tím prodlouží a prodraží.

Ve skutečnosti vypadá vztah mezi dobou vývoje a počtem defektů přibližně následovně:



[]

Ladění

* testováním nalezneme defekty, následuje proces ladění (angl. debugging)
* ladění = identifikace příčiny defektu (90% času) a její oprava (10% času)
* mělo by probíhat v 5 krocích - (1) stabilizace symptomu, (2) nalezení příčiny, (3) oprava, (4) otestování opravy defektu, (5) vyhledání obdobných defektů

* stabilizace symptomu - potřebujeme, aby se defekt projevoval spolehlivě
  - proto nejprve hledáme testovací případ, který symptom reprodukuje
  - testovací případ co nejvíce zjednodušíme, aby se defekt ještě projevoval
  - při zjednodušování testovacího případu často již můžeme vytvářet hypotézu, proč defekt nastává
  - existují defekty, které se neprojevují spolehlivě, například neinicializované proměnné, neplatný ukazatel, chyby časového souběhu
    . některé z těchto defektů můžeme zviditelnit; například neinicializované proměnné - před spuštěním programu paměť zaplnit náhodnými hodnotami apod.
    . chyby časového souběhu lze zviditelnit pomocí vložení yield(), případně náhodného prokládání (o chybách časového souběhu - viz předmět ZOS)

* nalezení příčiny symptomu
  - např. hledání zúžením podezřelé části kódu: systematicky vynecháváme kód (i na úkor funkčnosti), testujeme zda se symptom ještě vyskytuje
    . adaptace metody binárního vyhledávání - vynechá se přibližně polovina kódu, pokud se symptom projevuje opět rozdělíme na poloviny atd.
    . vynechávání volání podprogramu
  - použití ladícího programu nebo ladících výpisů - sledujeme, kde nastane symptom
    . přeskakujeme ty části programu, které nejsou relevantní, můžeme použít obdobné metody jako výše (aniž bychom vynechávali kód)
  - někdy pomůže se vyspat (podvědomí pracuje za nás)

* oprava defektu
  - pokud jsme defekt nalezli, bývá jeho oprava poměrně jednoduchá, ale je vysoké nebezpečí zanesení dalšího defektu (podle některých studií více než 50%)
  - podle studie z 1986 mají větší šanci provést opravu správně programátoři s celkovou znalostí programu (oproti programátorům, kteří se seznámí pouze s opravovanou částí programu)
  - proto je před opravou třeba rozumět jak problému, tak opravovanému programu

* opravu je třeba otestovat
  - defekty je nutné opravovat po jednom, opravy po jedné otestovat
  - poté program otestovat jako celek, nejlépe regresivními testy, aby se ukázaly případné vedlejší efekty opravy

- pro případ defektu v opravě je vhodné mít uchovánu předchozí verzi (ručně, SCM), porovnat obě verze (např. v UNIXu programem diff(1)), z toho je často možné zjistit problém.

* hledáme obdobné defekty

Defenzivní programování

* defenzivní programování = zabezpečení definovaného chování při chybných vstupech, zamezení propagace chyb z podprogramu ven

* nástroje:
 - kontroly vstupních parametrů
 - makro assert() v C, aserce v Javě), preconditions/postconditions v Eiffelu (programming by contract)
  . pokud nejsou, snadno si naprogramujeme jejich ekvivalent, např. v Pascalu:

```
procedure Assert(Condition: boolean; Message: string);
begin
     if (not Condition) then
     begin
          writeln('Assertion ', Message, ' failed. Aborting the program');
          halt;
     end
end;
```

  . použití např.:

```
Assert(delitel <> 0, 'delitel <> 0');  (* dělitel nesmí být roven 0 *)
```

 - výjimky v Javě, C++, Delphi, Pythonu apod. - konstrukce typu try-catch a try-catch-finally:

```
try {
    foo();                                  // zde může nastat výjimka
} catch (SomethingWentWrongException e) {   // pokud nastane výjimka,
    System.out.println("some error");       //  provede se toto
} finally {                                 // nakonec se vždy provede
    dispose();                              //  toto
}
```

 - program může při spuštění ověřovat své datové struktury, volání fcí apod.

* reakce na chybu
 - fce vrátí speciální návratový kód
 - programová výjimka, způsob propagace - podle stanovených konvencí
 - nastavení defaultní hodnoty vstupu nebo defaultního stavu
* o způsobu reakce na chybu se by mělo rozhodnout na úrovni architektury, aplikace by se měla ve všech svých částech chovat konzistentně

Poznámka (rozdíl mezi vývojovou verzí produktu a dodanou verzí)

Zatímco během vývoje potřebujeme, aby defekty byly co nejlépe viditelné, ve výsledném produktu se naopak snažíme, aby defekty co nejméně rušily. Ve výsledném produktu proto:

* ponecháváme kód, který kontroluje významné defekty; pokud aplikace hlásí interní chybu, měla by také oznámit, jakým způsobem jí uživatel může ohlásit
* zrušíme kód testující nepodstatné defekty,
* zrušíme kód, který způsobuje havárie
* ponecháme kód, který umožní přijatelné ukončení aplikace při chybě (např. s uložením dat)

Rušení kódu neprovádíme fyzicky (při ladění ho budeme opět potřebovat), ale např. pomocí preprocesoru vynecháme tělo procedury Assert, využijeme verzování apod.

```
 Např. v jazyce C definicí makra NDEBUG změníme všechny výskyty assert()
 na prázdný příkaz.

[]


Údržba SW systémů
=================

* údržba SW = aktivity, které jsou prováděny po uvolnění programu, resp. po
  jeho dodání zákazníkovi

* údržba zahrnuje především tyto tři typy aktivit:
 - oprava chyb SW (corrective maintenance)
 - přizpůsobení SW změnám prostředí, ve kterém běží - OS, periferie apod.
   (adaptive maintenance)
 - přidávání nové funkčnosti nebo změna funkčnosti na základě požadavků
   uživatele (perfective maintenance)

* v průměru cca 17% údržby se týká opravy chyb, 18% přizpůsobení SW změnám
  prostředí a 65% přidávání nebo změny funkčnosti
* údržba tvoří cca 50% až 75% vývoje, pro obtížně změnitelné systémy (jako
  jsou zapouzdřené systémy reálného času) až 80%
 - studie ukazují, že cena údržby systému postupně stoupá
 - proto je efektivní systém navrhnout a implementovat tak, aby se cena
   údržby snížila

* proces údržby je spuštěn množinou požadavků na změny od uživatelů systému,
  managementu nebo od zákazníka
 - provedeme vyhodnocení ceny a dopadu změn
 - navržené změny jsou schváleny nebo neschváleny; některé jsou odloženy
  . rozhodnutí o schválení/neschválení změn je do určité míry ovlivněno
    udržovatelností komponenty, ve které změnu provádíme
 - změny jsou implementovány a ověřeny
  . v ideálním případě: změna specifikace systému, návrhu, implementace,
    otestování systému
 - je dodána nová verze
* už bylo probíráno, viz ”Správa požadavků” na třetí přednášce

Nestrukturovaná údržba
......................

* výše uvedené by ovšem platilo v ideálním případě
* pracnost a cenu údržby ve skutečnosti zvyšují tyto faktory:
 - po dodání produktu je tým obvykle rozpuštěn, lidé jsou přiřazeni jiným
   projektům; údržba je přenechána jinému týmu nebo jednotlivcům, kteří
   systém neznají => většina jejich úsilí musí být věnována pochopení
   existujícího systému
 - smlouva bývá obvykle pouze na dodání systému, o údržbě se nemluví => tým
   nemá motivaci vytvářet udržovatelný SW (zvláště pokud se předpokládá, že
   údržbu bude provádět někdo jiný)
 - údržba je obvykle přenechána nejméně zkušeným programátorům, navíc systémy
   mohou být vytvořeny v zastaralých programovacích jazycích (Cobol), které
   se tým provádějící údržbu musí teprve naučit
 - změnami se snižuje strukturovanost kódu - tím roste ”entropie” SW,
   dokumentace starých systémů může být ztracena nebo může být
   nekonzistentní atd.

* v mnoha případech je jediným dostupným prvkem SW konfigurace zdrojový text
 - proto proces údržby začíná (obtížným) procesem vyhodnocení kódu
  . kód obsahuje pouze implementaci, nikoli záměr => obtížná interpretace
  . pokud neexistují testy, není možné regresivní testování
 - takovému případu říkáme nestrukturovaná údržba (unstructured
   maintenance), produktivita klesá na 40:1 oproti vývoji (podle Boehma)
 - tomuto stavu se snažíme předejít, pokud je to možné

* v zásadě dvě možnosti, jak se k problému postavit:
 - předpokládat vodopádový model - systém vyvinout, udržovat dokud se údržba
   stane ekonomicky neúnosná, pak nahradit novým systémem
```

- evoluce systému - předpokládat např. spirálový model
  . systém by měl být navržen tak, aby ho bylo možné přizpůsobovat novým požadavkům
  . pokud systém nevyhovuje (= nízká udržovatelnost), musíme systém přepracovat (přestrukturovat apod.)

* udržovatelnost systému lze měřit následujícími metrikami:
- počet požadavků na opravy chyb
  . pokud počet požadavků na opravy stoupá, může to znamenat, že do programu je procesem údržby vnášeno více chyb, než je jich odstraňováno údržbou, tj. indikuje snížení udržovatelnosti
- průměrná doba potřebná pro vyhodnocení dopadu změny
  . odráží rozsah (např. počet komponent), které jsou ovlivněny požadavkem na změnu
  . pokud naroste, udržovatelnost se snižuje
- průměrná doba implementace požadavku na změnu
- počet nevyřízených požadavků na změnu

* pokud je program špatně udržovatelný, jsou možné následující kroky pro zlepšení:
- konverze SW do moderního programovacího jazyka (nebo do modernější varianty použitého programovacího jazyka); například z Fortranu do jazyka Java nebo C#
- zpětné inženýrství = analýza programu s cílem najít specifikaci a návrh SW
  . obvykle na základě zdrojových textů, v některých případech jsou ztraceny a je nutné vycházet ze spustitelného kódu
- vylepšení struktury programu s cílem zlepšit jeho srozumitelnost
  . pro automatickou transformaci nestrukturovaného kódu na strukturovaný existují nástroje (při transformaci ale ztratíme původní komentáře)
- modularizace programu = sdružení souvisejících částí programu, v některých případech včetně transformace architektury SW
  . stejná motivace jako při vytváření modulů při vývoji, tj. abstrakce dat, abstrakce řízení HW, sdružení příbuzných fcí
- přizpůsobení zpracovávaných dat změněnému programu

O postupech při přepracovávání existujících systémů hovoří kniha Martina Fowlera ”Refactoring: Zlepšení existujícího kódu. Grada, Praha 2003.”

Metriky

* metrika (metrics) = jakékoli měření atributů SW produktu nebo SW procesu
- např. počet řádků kódu, počet defektů, defekty na 1000 řádek kódu apod.
- jsou potřebné, abyste věděli, co se s projektem opravdu děje, pro zlepšování SW nebo SW procesu, pro odhady o budoucích projektech, pro informaci, kam zaměřit testování atd.
- např. před zavedením testovacího nástroje můžeme změřit počet defektů nalezených za časovou jednotku, totéž po zavedení nástroje
- jiný příklad - nejvíce defektů bývá v procedurách/metodách s vysokou cyklomatickou složitostí - tam bychom měli zaměřit své úsilí při testování

* nejdůležitější metriky se týkají následujících oblastí:
- metriky analytického modelu (např. kvalita specifikace)
- metriky návrhu (např. složitost komponent)
- metriky zdrojových textů (např. níže uvedené LOC a cyklomatická složitost)
- metriky pro testování (např. pokrytí logiky programu, efektivita testování)

* nejjednodušší a nejčastější metrika zdrojových textů - počet řádek kódu (Lines of Code, LOC)
- otázka - co máme počítat jako řádek kódu?
- pokud se snažíme měřit množství práce do programu vložené, nebudeme započítávat komentáře, prázdné řádky a automaticky generovaný kód
- někdy se nazývá NCLOC (Non-comment LOC) nebo ELOC (Effective LOC)
- je základ metodiky COCOMO pro odhad ceny SW

* McCabeho metrika ”cyklomatická složitost” počítá rozhodovací body v podprogramu
- vysoká cyklomatická složitost má korelaci s chybovostí, s obtížností číst a testovat podprogram (jak už bylo řečeno, prostá délka podprogramu

nemá korelaci s chybovostí)
- cyklomatickou složitost spočteme následovně:

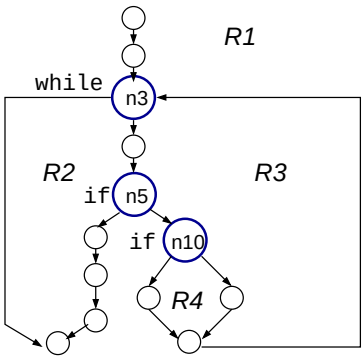
  1. Pro přímou cestu podprogramem započteme 1.
  2. Pro každé z následujících klíčových slov nebo jejich ekvivalentů přičteme jedničku: if, while, do-while, for, a pro ”and a ”or” ve složených podmínkách.
  3. Pro každý případ v switch/case přičteme 1.

- pokud je >10, je příznak, že je vhodné podprogram zjednodušit (např. část podprogramu vložit do dalšího podprogramu volaného z původního)

Poznámka pro zajímavost (cyklomatická složitost a graf toku řízení)

Pokud v rozhodovacích příkazech nejsou složené podmínky, odpovídá cyklomatická složitost počtu regionů v grafu toku řízení podprogramu. Pojmem region se myslí oblast ohraničená hranami a uzly grafu; oblast mimo graf se počítá jako samostatný region.

Jako příklad uvádím graf toku řízení pro příklad binárního vyhledávání, uvedený na minulé přednášce:



Pro kontrolu si můžeme spočíst cyklomatickou složitost pomocí výše uvedené metody: započteme přímou cestu, while, if a if, tj. výsledek je opět 4.

[]

Některé užitečné metriky
.....

* velikost zdrojových textů
 - celkový počet řádek (LOC)
 - celkový počet komentářových řádek (CLOC)
 - počet deklarací dat
 - počet prázdných řádek

* produktivita
 - E-faktor (environmental factor) = počet nepřerušených hodin / celková pracovní doba; optimální je cca 0.4
 - pracovní doba strávená na projektu
 - pracovní doba strávená na každém podprogramu
 - počet změn v podprogramu
 - náklady projektu
 - náklady projektu na řádek kódu
 - náklady na opravu defektu

* sledování defektů
 - vážnost defektu
 - místo defektu
 - způsob opravy defektu
 - osoba zodpovědná za defekt
 - počet řádek změněných při opravě defektu
 - pracovní doba strávená opravou defektu
 - průměrná doba potřebná na nalezení defektu
 - průměrná doba potřebná na opravu defektu
 - počet pokusů opravit defekt
 - počet nových chyb zanesených opravou defektu

* celková kvalita

- celkový počet defektů
- počet defektů v podprogramu
- průměrný počet defektů na 1000 řádek kódu
- střední doba mezi haváriemi
- chyby detekované překladačem

* udržovatelnost
- počet parametrů předávaných každému podprogramu
- počet lokálních proměnných použitých každým podprogramem
- počet podprogramů volaných každým podprogramem
- počet rozhodovacích bodů v každém podprogramu
- složitost řídících konstrukcí v každém podprogramu
- počet řádek kódu v každém podprogramu
- počet komentářů v každém podprogramu
- počet deklarací dat v každém podprogramu
- počet prázdných řádek v každém podprogramu
- počet příkazů goto v každém podprogramu
- počet vstupně/výstupních příkazů v každém podprogramu

* objektově orientované metriky
- vážená složitost třídy
- hloubka stromu dědičnosti
- počet přímých potomků třídy
- počet operací předefinovaných v každé podtřídě
- stupeň provázanosti mezi třídami

Práce v týmech

* většina profesionálních programátorů pracuje v týmech, týmy od 2 do několika set lidí
- pokud je tým velký, je zřejmá potřeba, aby byl nějak strukturován
  . rozdělení do skupin, každá skupina je zodpovědná za podprojekt
  . skupiny by neměly mít více než 8 členů
  . malý počet = zmenšení komunikačních problémů

Constantine (1993) popisuje čtyři paradigmata pro organizaci týmů:

* uzavřené paradigma - tým má pevně stanovenou hierarchii
- příkladem je na první přednášce zmíněný ”chirurgický tým”
- tyto týmy pracují dobře, pokud je SW podobný předchozím projektům, obvykle bývají méně inovativní

* náhodné paradigma - tým má volnou strukturu, role závisejí na iniciativě jednotlivých členů týmu (”tvořivá nezávislost”)
- může mít vysokou výkonnost, pokud si členové mohou vzájemně důvěřovat, jednotliví členové mají přiměřené dovednosti a pokud neobsahuje rebely
- vhodné pro inovativní projekty, často problémy pokud je vyžadována ”běžná práce”

* otevřené paradigma - tým založený na spolupráci, typicky značná komunikace a rozhodování založené na konsensu
- vhodné pro řešení obtížných problémů, obvykle nebývá tak efektivní jako jiné typy týmů

* synchronní paradigma - závisí na možnosti rozdělit problém na nezávislé části
- členové týmu pracují na jednotlivých podproblémech, členové mezi sebou nemusejí příliš komunikovat

Nezávisle na typu organizace týmu ovlivňují práci v týmu především 4 faktory:

* složení týmu: má tým vyvážené technické schopnosti, zkušenosti, osobnostní charakteristiky?
* koheze týmu: je tým množinou jednotlivců nebo má ”skupinového ducha” (skupina o sobě uvažuje jako o týmu)
* skupinová komunikace: dokáží spolu členové efektivně komunikovat?
* organizace týmu: má každý přiměřenou roli v týmu?

Složení týmu
.....

* v psychologické studii motivace (Bass & Dunteman) se ukázalo, že profesionály lze v zásadě rozdělit podle jejich motivace do tří kategorií:
 - úkolově orientovaní - motivací je jim práce, kterou vykonávají
  . při vytváření SW je motivací intelektuální výzva vytvořit SW
  . platí pro velkou část vývojářů
 - orientovaní na sebe - v zásadě motivováni osobním úspěchem a uznáním
  . vývoj SW je jim prostředkem k dosažení vlastních cílů
 - orientovaní na interakci - jsou motivovaní přítomností a činností spolupracovníků

* lidé orientovaní na interakci pracují raději ve skupinách, zatímco úkolově orientovaní a na sebe orientovaní obvykle raději pracují sami
* u žen je pravděpodobnější orientace na interakci než u mužů, často jsou efektivnější komunikátoři

* motivace jednotlivce se skládá ze všech tří kategorií, jedna z nich ale většinou převažuje
* osobnosti nejsou statické, motivace se může měnit
 . například pokud má úkolově orientovaný člen týmu pocit, že není přiměřeně odměňován, může se jeho motivace změnit na ”orientovaný na sebe”

* pro skupinu je dobré, pokud obsahuje doplňující se typy osobností:
 - úkolově orientovaní bývají obvykle nejsilnější technicky
 - na sebe orientovaní obvykle tlačí tým na dokončení práce (výsledky)
 - orientovaní na interakci napomáhají komunikaci uvnitř skupiny

* Proč optimalizovat složení týmu [převzato od P. Brady]

 Dobře vyvážený tým je velmi silná zbraň s enormní kapacitou k tvořivé práci - a proto je také drahý a musí být zatěžován odpovídajícími úkoly. Optimalizovat složení je tedy vhodné při vytváření nových skupin za účelem ambiciózních a náročných úkolů, a také pro týmy, které musí obstát v prostředí velkých změn, silné konkurence, potřeby rychlé inovace a akčnosti.

* Kdy složení týmu není kritické [převzato od P. Brady]

 Není vždy nutné snažit se optimalizovat složení týmu. Optimalizace není na místě v případech rutinních operací a úloh pod intelektuální úrovní ideálního týmu - takový tým by pro daný účel byl velmi drahý nehledě na to, že by práce neposkytovala jeho členům motivaci a uspokojení.

 Někdy ji nelze aplikovat z praktických či logistických důvodů - ne vždy je možnost vybrat lidi s požadovanými vlastnostmi, aniž by bylo potřeba nabírat nové zaměstnance; je také možné, že jsou k dispozici lidé s potenciálem, ale bez technických znalostí.

 Je vždy lepší se o vyvážení složení týmu pokusit částečně než vůbec. Pokud přesto není vyhovující, mohou se členové jeho nedostatky snažit nahradit bděním nad slabými aspekty s použitím ”nejlepších ze všech špatných” lidí a postupnou změnou.

* důležitá role je vedoucí skupiny
 - obvykle technické směřování a administrace projektu
 - sledují práci týmu, efektivitu

* vedoucí obvykle určeni managementem
 - problém - určení vedoucí nemusejí být vůdci skupiny po technické stránce
 - ve skutečnosti si skupina může najít ve svém středu např. technicky nejschopnějšího, nejlepšího motivátora
 - někdy je proto výhodné oddělit technické vedení od administrace projektu

Koheze skupiny
.....

* kohezivní skupina = pro členy jsou jejich individuální zájmy méně důležité než zájmy skupiny

- tj. členové se cítí být členy skupiny, snaží se skupinu chránit, pomáhat si navzájem apod.
- lze podpořit poskytováním informací a důvěry skupině

* výhody kohezivní skupiny:
- konsensem mohou vzniknout standardy kvality
- členové skupiny těsně spolupracují - mohou se od sebe navzájem učit apod.
- členové znají navzájem svojí práci (výhoda pokud např. člen skupiny onemocní)
- programy mohou být chápány jako skupinové vlastnictví (egoless programming) - usnadňuje provádění inspekcí, přijímání kritiky a vylepšování programu skupinou

* kohezivní skupiny mají náchylnost ke dvěma problémům:
- iracionální rezistence ke změně vedoucího - pokud je vedoucí nahrazen někým mimo skupinu, skupina se může sjednotit proti novému vedoucímu => snížení produktivity
  . pokud je to možné, měl by být vedoucí zvolen z členů skupiny
- tzv. skupinové myšlení - kritické myšlení je potlačeno ve prospěch loajality vůči skupině (resp. skupinovým normám, skupinovým rozhodnutím)

Komunikace uvnitř skupiny

* dobrá komunikace mezi členy skupiny je podstatná
* podstatné faktory:
- velikost skupiny
- struktura skupiny
- složení
- fyzické pracovní prostředí

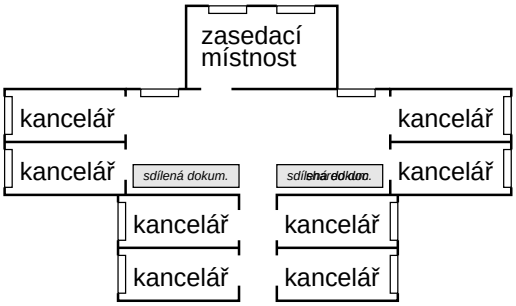
* klíčovým faktorem je velikost skupiny
- počet možných komunikačních cest je n * (n-1)
- tj. pro sedmičlennou skupinu (42 cest) je pravděpodobné, že někteří členové spolu budou komunikovat velmi zřídka

* dalším faktorem struktura skupiny
- v neformálně strukturovaných skupinách efektivnější komunikace než ve formálně (hierarchicky) strukturovaných skupinách
- v hierarchicky strukturovaných skupinách mají informace tendenci putovat nahoru a dolů v hierarchii, ale lidé na některé úrovni spolu nekomunikují - problém velkých skupin

* složení skupiny
- pokud se skupina skládá z příliš mnoha lidí stejného osobnostního typu, nastávají konflikty a komunikace uvázne
- komunikace je obvykle lepší ve skupinách, kde jsou muži i ženy, než ve skupinách složených pouze z mužů nebo pouze z žen
- ženy jsou častěji interakčně orientované => mohou být prostředníci

* fyzické pracovní prostředí
- velmi podstatné pro chování a výkonnost skupiny
- podle (DeMarco & Lister 1985) rozdíly ve výkonnosti týmů až 1:11
- pro výkonnost nejdůležitější vlastnosti:
  . soukromí - potřeba prostoru, kde se mohou soustředit na práci bez vyrušování
  . přirozené světlo, viditelnost vnějšího prostředí (= okna)
  . možnost individuálních úprav prostředí podle způsobu práce

* pro komunikaci je podstatné, aby skupina mohla diskutovat projekt formálně i neformálně
- (Weinberg 1971) cituje případ, kdy organizace chtěla zabránit programátorům ”ztrácet čas” povídáním u automatu na kafe; po odstranění automatu se okamžitě dramaticky zvýšil počet formálních požadavků na výpomoc
- vyplatí se mít neformální místo pro setkávání, stejně jako konferenční místnost pro formální sezení



Softwarové profese
....

V rámci týmů vykonávají různí členové různou práci; všichni jsou stejně potřební, někteří ale nesou větší zodpovědnost.

Jako příklad uvedu rozdělení na profese podle (Paleta 2003):

* zadavatel nebo manager produktu - formálně není součástí týmu
 - sestavuje požadavky na vytvářenou aplikaci, zodpovídá otázky týmu, přebírá aplikaci
 - u systémů vytvářených na zakázku je zástupcem zadavatele

* vedoucí projektu - řídí vlastní vývoj, je zodpovědný za splnění požadavků, termínu a rozpočtu
 - komunikuje se zadavatelem nebo managerem produktu

* technický leader nebo architekt - ostatní se na něj obracejí, pokud mají technický dotaz
 - navrhuje celkovou strukturu aplikace, vybírá technologie a vývojové prostředky

* databázový specialista - návrh databáze
 - příprava výkonnostních testů a na základě jejich výsledků optimalizace databáze

* analytik - rozpracovává specifikaci, vytváří konceptuální model aplikace
 - navrhuje posloupnost obrazovek apod.
 - role může být kombinována s pozicí programátora nebo tvůrce dokumentace

* návrhář uživatelského rozhraní - navrhuje obrazovky aplikace tak, aby byly přehledné a snadno použitelné

* programátor - vytváří kód na základě specifikace a konceptuálního modelu

* tester - řídí nebo provádí testování

* tvůrce dokumentace - zpracovává technickou dokumentaci vytvářenou ostatními členy týmu, vytváří uživatelskou dokumentaci (manuály, nápověda)

Konfigurační management
=======================

* v SW projektech se mění požadavky na systém, design systému, kód, dokumentace systému atd.
 - v průběhu času vědí všichni zúčastnění více (o tom co potřebují, jak by se to nejlépe udělalo, atd.)
 - požadavky na změny budou přicházet ve všech fázích tvorby SW
 - pro řízení změn v projektech byly vyvinuty procesy, nazývané souhrnně konfigurační management (angl. software configuration management, SCM)
 - úkolem SCM definovat procedury pro provádění změn, eliminuje některé problémy vznikající zejména pokud je mnoho vývojářů a mnoho verzí SW

* představte si tým vyvíjející SW
 - úspěšný SW => tisíce požadavků na opravy a vylepšení
 - kód ve sdíleném adresáři - co když dva programátoři provádějí změnu ve

```
   stejném modulu?
 - oprava chyby v produkční verzi, měla by se promítnout zároveň ve vývojové
   verzi, do které jsou ale mezitím přidávány další vlastnosti
 - vylepšení zavleklo chyby - jak se vrátit ke staré verzi?
 - jak zjistit, z čeho se která verze skládá?
 - obvykle kombinace těchto + dalších problémů
* pro úspěch projektu podstatná schopnost řídit změny tak, aby si systém mohl
  zachovat integritu v čase

Tradiční SCM proces
-------------------

* v tradičním procesu vývoje SW založeném na vodopádovém modelu je SW předán
  SCM týmu po dokončení vývoje a po otestování jednotlivých komponent
 - SCM tým přebírá odpovědnost za sestavení úplného systému a za vedení testů
 - chyby objevené při testu systému jsou předány zpět k opravě vývojovému týmu
 - tento přístup ovlivnil tvorbu standardů; např. IEEE Std. 828 nevysloveně
   předpokládá vodopádový model, tj. obtížně se přizpůsobují např.
   inkrementálnímu vývoji
 - proto také SCM patří k nejhůře zpracovaným tématům v literatuře o SW
   inženýrství
 - napřed popíšu tradiční SCM proces (tak jak ho popisují standardy), pak
   zmíním přizpůsobení SCM pro inkrementální model SW procesu

* tradiční SCM definuje 4 procedury, které musejí být pro SW projekt
  definovány, pokud má být definován dobrý SCM proces
 - identifikace konfigurace
 - řízení konfigurace
 - vytváření záznamů o stavu konfigurace
 - autentizace konfigurace

Identifikace konfigurace
........................

* informace, které jsou výstupem jednotlivých fází SW procesu, můžeme rozdělit
  do tří velkých kategorií: (1) programy (jak zdrojové texty tak
  spustitelné), (2) dokumentace programů, (3) data
 - na počátku máme specifikaci systému, z ní vznikne DSP, později design atd.
 - informace vytvořená v důsledku SW procesu a reprezentující určitou podobu
   daného SW systému se nazývá konfigurace SW

* konfigurace sestává z tzv. ”konfigurovatelných položek” (configurable item,
  CI), které jsou atomické z hlediska změn a označování verzí
 - CI bude např. DSP, ERA model, jeden .java soubor, jedna .dll knihovna,
   množina testovacích případů apod.
 - mezi CI existují závislosti (kompozice, generování, master-dependent, ...)
 - každá CI je jednoznačně identifikovatelná (typ CI, identifikátor projektu,
   identifikátor změny nebo verze)

* konzistentní konfigurace = SW konfigurace, jejíž prvky jsou navzájem
  bezrozporné
 - obsahuje např. zdrojové texty, makefiles, konfigurační soubory,
   dokumentace, testů atd. v příslušných verzích
 - bezrozporná = např. zdrojové soubory lze přeložit, knihovny přilinkovat

* baseline = konzistentní konfigurace tvořící stabilní základ pro produkční
  verzi nebo další vývoj (startovací bod pro řízenou evoluci)
 - příklad: milník beta verze aplikace stabilní: vytvořená, otestovaná a
   schválená managementem
 - pro baseline předpokládáme následující vlastnosti:
  . dokumentovaná funkčnost, tj. vlastnosti SW pro každou baseline jsou dobře
    známé
  . známá kvalita: např. známé chyby budou dokumentovány, SW prošel
    testováním před tím, než je definován jako baseline
  . baseline je nezměnitelná a znovu vytvořitelná: po definici nemůže být
    baseline změněna, všechny CI tvořící baseline můžeme kdykoli znovu
    vytvořit
 - změny prvků baseline jen podle schváleného postupu
 - při problémech návrat k baseline
```

```
 - každá nová baseline je předchozí baseline + souhrn schválených změn CI

* proces identifikace konfigurace definuje baseline, z jakých CI se skládá

* další užitečné pojmy:
 - delta = množina změn CI mezi dvěma po sobě následujícími verzemi
  . v některých systémech jednoznačně identifikovatelná
 - repository (úložiště, databáze projektu) = centrální místo, kde jsou
   uloženy všechny CI projektu
  . řízený přístup (udržení konzistence)
 - workspace (pracovní prostor) = soukromý datový prostor, v němž je možno
   provádět změny prvků konfigurace, aniž by byla ovlivněna jejich podoba v
   repository
  . akce ”zkopírování CI z repository” a ”uložení CI do repository”

Řízení konfigurace
..................

* problém ve všech fázích životního cyklu:
 - jak zvládat množství požadavků na úpravy produktu (opravy, vylepšení)?
 - jak poznat, kdy už jsou vyřešeny?

* nutný striktní postup akcí: klasifikace a vyhodnocení navrhovaných změn,
  jejich schválení nebo neschválení, koordinace schválených změn,
  implementace změn na příslušnou baseline, dokumentace a ověření

* požadavek obsahuje: název a verzi produktu / subsystému, kterého se týká;
  popis chyby či požadované změny (co nejpřesnější) + indikaci priority;
  pro chybu: jak vznikla, jak je možné ji znovu reprodukovat;
  informace o použitém software (konfigurace, OS, knihovny)
 - požadavek prochází stavy: nový -> převzatý -> přiřazený ->
   vyřešený/zrušený/duplicitní -> uzavřený

* postup zpracování požadavku
 - přijetí požadavku
  . přidělení ID
  . nastavení závažnosti, priority (kritická chyba - problém - vada na kráse
    - vylepšení)
 - v tradičním procesu schválení, neschválení, odložení změny řídí ”komise
   pro řízení konfigurace” (angl. configuration control board, CCB)
  . chyba -> nutno ověřit, že chyba je reálná
 - zpracování požadavku
  . pověřený člověk (dle zodpovědnosti za části systému)
  . lokalizace změn v produktech procesu
  . oprava v lokálním workspace, testování a validace
  . schválení, nová baseline

* SW podpora - systémy pro správu změn (bug tracking systems)
 - evidence a archivace požadavků, sledování stavu požadavku, případně
   statistiky
 - často emailové, webové
 - např. Gnats + Gnatsweb, Bugzilla, JitterBug apod.

Vytváření záznamů o stavu konfigurace
.....................................

* zajištění sledovatelnosti změn SW
* zaznamenávání informací o každé verzi SW a o změnách oproti předchozí
  baseline, které k této verzi vedly
* záznam pomůže zodpovědět otázky jako
 - ”Byla chyba XYZ opravena?”
 - ”Kdo je zodpovědný za tuto modifikaci?”
 - ”Čím přesně se tato verze liší od baseline?”
* konkrétní nástroje uvedeme později

Autentizace konfigurace
.......................

* proces, který zajišťuje
```

- aby v nové baseline byly zahrnuty všechny plánované a schválené změny
- aby součástí dodaného systému byly všechny požadované programy, dokumentace a data

SCM pro inkrementální vývoj

* příklad procesu:
 - celý systém se sestavuje často (např. denně)
 - organizace určí čas, do kdy musí vývojáři doručit své komponenty (např. 14h)
 - komponenty mohou být neúplné, ale musejí poskytovat základní funkčnost, která může být otestována

* z komponent se sestaví nová verze systému
 - systém je předán testovacímu týmu, který provede předdefinované testy systému
 - vývojáři zatím dále pracují na svých komponentách, přidávají funkčnost a opravují chyby objevené v předchozích testech
 - testovací tým zdokumentuje objevené chyby, předá je vývojářům - vývojáři chybu opraví v další verzi komponenty

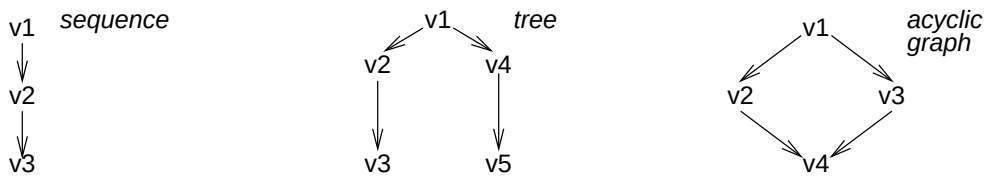
* hlavní výhodou denního sestavování je brzké nalezení problémů vzniklých interakcí komponent
* vývojáři pociťují tlak, aby jejich komponenty nezpůsobily havárii systému - důsledkem je lepší testování jednotek

* SCM proces musí někdo řídit, musí ustanovit podrobné procedury, musí zajistit, aby všechny změny proběhly správně
* příklad velkého projektu - jádro OS Linux:
 - většinu skutečného vývoje jiní vývojáři, ale jaké vlastnosti bude obsahovat jádro určuje jeden člověk - Linus Torvalds
 - všichni vývojáři mu posílají změny, které mají být začleněny do jádra
 - hraje roli SCM procesu:
  . řízení konfigurace (začleňování/nezačleňování změn ostatních vývojářů)
  . vytváření záznamů o stavu konfigurace (ChangeLog)
  . autentizace konfigurace (zaručuje, že jádro má všechny části)

Verzování

* účel: udržení přehledu o podobách CI
 - verze popisuje stav CI, nebo postup jeho změn
 - extenzionální verzování: každá verze má jednoznačné ID
   např. 1.5.1 = základní verze pro DOS, 1.5.2 pro UNIX,
   1.6.1 oprava pro DOS, 2.1.1 = nový release pro DOS
  . přístup v často používaných nástrojích (rcs/cvs, SourceSafe)
  . jednoduchá implementace
  . nepoužitelné při větším počtu verzí
 - intenzionální verzování: verze je popsána souborem atributů
  . např. OS=DOS and UmiPostscript=YES
  . nutné pro větší prostory verzí
  . potřeba vhodných nástrojů (Adele, částečně cpp)

* prostor verzí je často reprezentován grafem
 - uzly = verze, hrany = vazby mezi verzemi, nejčastěji relace následnictví
 - větvení (branch) - často nahrazuje varianty (rcs/cvs)
 - operace vytvoření větve (branch-off, split) a spojení (merge)



* nástroje pro verzování
 - ruční verzování = dohody a konvence o značení verzí v názvech (dokument,

```
   soubor, adresář), baseline pomocí zálohy všech souborů v daném čase
 - základní - správa verzí souborů
  . obvykle extenzionální verzování modulů
  . ukládání všech verzí v zapouzdřené úsporné formě
  . příklady nástrojů: rcs, cvs
 - pokročilé - integrované do CASE
  . obvykle kombinace extenzionálního a intenzionálního verzování
  . automatická podpora pro ci/co prvků z repository do nástrojů
  . příklad nástroje: ClearCase, Adele

rcs: Revision Control System
............................

* správa verzí pro textové soubory; UNIX, Windows, DOS
 - extenzionální stavové verzování komponent
 - části systému - utility spouštěné z příkazového řádku:
  . ci, co, rcs, rlog, rcsdiff, rcsmerge
 - ukládá (do foo.c,v souboru)
  . historii všech změn v textu souboru
  . informace o autorovi a času změn
  . textový popis změny zadaný uživatelem
  . další informace (stav prvku, symbolická jména)
 - používá diff(1) pro úsporu místa
  . poslední revize uložena celá
  . předchozí pomocí delta vygenerované programem diff(1)
 - funkce
  . zamykání souborů, poskytování R/O kopií
  . symbolická jména revizí, návrat k předchozím verzím
  . možnost větvení
  . informace o souboru a verzi lze včlenit do textu pomocí klíčových slov
    $Author$, $Date$, $Revision$, $State$, $Log$ (popis poslední změny
    zadaný uživatelem), $Id$ (kombinace filename revision datum author state)
  . typické použití v C: static char rcsid[] = "$Id$"
    při "co" expanduje na "$Id: soubor.c,v 1.1 2003/05/16 03:17:16 luki Exp $"

CVS (Concurrent Versioning System)
..................................

* nadstavba nad rcs => umí vše, co umí rcs (zejm. klíčová slova)
* optimistický přístup ke kontrole paralelního přístupu
 - místo zamkni-modifikuj-odemkni (RCS) pracuje systémem zkopíruj-modifikuj-sluč
* práce s celými konfiguracemi (projekty) najednou
* sdílené repository + soukromé pracovní prostory
* repository lokální nebo vzdálená (rsh, p-server)
* možnost definovat obsah a strukturu konfigurace
* zjišťování stavu prvků, rozdílů oproti repository
* příkazová řádka, grafické nadstavby (UNIX, Windows, web)

* rcs i cvs jsou volně šířené
* množství informací on-line, viz např. http://www.loria.fr/~molli/cvs-index.html
* existují další podobné nástroje, např. PRCS, Subversion apod.

cpp: Realizace variant
......................

* cpp = C preprocessor, umožňuje intenzionální stavové verzování
 - např. chceme variantu foo.c pro případ OS=DOS and UmiPostscript=YES
 - definice atributů pro popis variant
 - hlavičkový soubor (centrální místo def. varianty celého systému)
   parametry příkazové řádky gcc -DOS_DOS (např. v Makefile)


Automatizace překladu a sestavení projektu: make
------------------------------------------------

* program "make" pochází ze systémů UNIX, původně souvislost s jazykem C
* účelem automatizace překladu a sestavení projektu, minimalizace času
  spotřebovaného vytvářením aktuálních verzí objektových a dalších "strojově
  vytvářených" souborů
```

* spustitelný program a objektové soubory se typicky vytvářejí ze zdrojových textů
* popis pravidel překladu umístíme do souboru ”Makefile” (případně ”makefile”)
* překlad spustíme příkazem ”make”

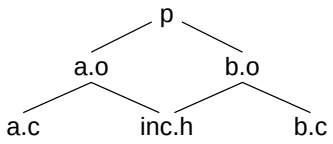
Příklad (projekt v jazyce C v systému UNIX)

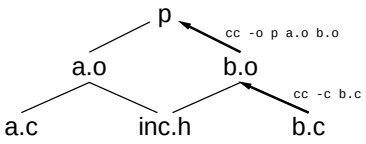
 * program p bude sestaven ze dvou objektových souborů a.o a b.o
 * ty se vytvářejí překladem z odpovídajících zdrojových textů a.c a b.c, oba používají společný hlavičkový soubor inc.h:

 * pro překlad projektu vytvoříme soubor Makefile, obsahující tři pravidla:

```
p: a.o b.o
        cc -o p a.o b.o

a.o: a.c inc.h
        cc -c a.c

b.o: b.c inc.h
        cc -c b.c
```

 * poslední pravidlo znamená:
  - soubor ”b.o” závisí na souborech ”b.c” a ”inc.h”
   . pokud bude soubor ”b.c” nebo soubor ”inc.h” mladší než ”b.o”, je třeba ”b.o” znovu vytvořit pomocí příkazu ”cc -c b.c”
   . b.c bude mladší, např. pokud v něm provedu změnu textovým editorem
  - ostatní pravidla obdobná, tj. pokud dojde ke změně, provede se minimální počet příkazů, která zajistí, aby výsledek byl aktuální





[]

* pravidla mají tvar

```
      cíl: prerekvizity
              příkaz1
              příkaz2
```

 - cíl (cílový soubor, co se má vytvořit)
 - volitelně prerekvizity (soubory, ze kterých se cíl vytváří)
 - volitelně příkazy, které se spustí, pokud je některá z prerekvizit mladší než cíl (cíl je ”zastaralý”, měl by se vytvořit z prerekvizit; pro zjištění stáří se používá čas modifikace souboru)
  . pozor, v UNIXovém ”make” musí být příkazy uvozeny tabulátorem

* provedení souboru Makefile - spustíme příkaz ”make”
 - make najde první pravidlo v souboru Makefile
 - před provedením pravidla rekurzivně zajistí, aby jeho prerekvizity nebyly zastaralé
  . každou prerekvizitu bude považovat za cíl, najde příslušné pravidlo
  . pokud je cíl zastaralý, vytvoří ho znovu pomocí příkazů pravidla
 - chyba (nenulová návratová hodnota příkazu) způsobí ukončení programu make (lze potlačit uvedením ”-”, tj. ignoruje případnou chybu)

* falešné (phony) cíle
 - cíl je ve skutečnosti ”návěští podprogramu”, nikoli vytvářený soubor
 - pravidlo nemá prerekvizity
 - používá se např. pro automatizaci úklidových akcí, instalaci, provedení testů, vytváření distribučních archivů apod. (phony cíle clean, install, test apod.)

```
       clean:
               -rm *.o core
```

* proměnné (v terminologii programu make nazývané makra)
 - definice: jméno=řetězec
 - použití: $(jméno)
 - příklad:

```
      CC=gcc  # kterým překladačem jazyka C budeme překládat

      p: a.c
              $(CC) -o p a.c
```

* další vlastnosti: vestavěná pravidla, inferenční pravidla

* soubory Makefile najdete ve většině volně šířených programů (jádro OS Linux apod.)
* protože ”make” je velmi používáno, mnoho firem apod. má vlastní rozšíření (podpora paralelního běhu, ”include” a ”ifdef” podobně jako v C, apod.)
* gcc -M umí vytvořit závislosti pro objektové soubory, jikes pro .class

* existují další nástroje se obdobným účelem, např. Ant pro automatizaci překladu projektů v jazyce Java

Etické a právní aspekty tvorby SW

* stejně jako v jiných oborech i v SW inženýrství existují určitá etická pravidla, jejichž nedodržování je považováno za neslušné a neprofesionální
* profesionální organizace jako ACM a IEEE definovaly ”etické zásady SW inženýra” (CS Code of Ethics), viz http://www.acm.org/serving/se/code.htm

* některé základní zásady:
 - při vytváření SW máte možnost být někomu prospěšní nebo mu způsobit škodu (nebo dát možnost jiným, aby byli prospěšní nebo ublížili)
  . například pokud jste zodpovědní za vývoj systému kritického pro bezpečnost lidí a čas vás tlačí - bylo by neetické prohlásit systém za otestovaný, pokud není
  . pokud je pravděpodobná účast na vojenských, nukleárních nebo jiných projektech, na které existují různé pohledy z etického hlediska, je třeba to předem mezi zaměstnavatelem a zaměstnancem vyjasnit
 - důvěrnost - měli byste respektovat důvěrnost informací o klientech nebo zaměstnavateli
 - způsobilost - měli byste si být vědomi své úrovně a nepřijímat vědomě práci, která je nad vaše schopnosti
 - dodržovat autorská práva, patenty apod. - porušováním můžete uvést do potíží nejen sebe
 - nezneužívat cizí počítače např. pro provozování programů, se kterými by vlastník nesouhlasil

Autorský zákon

* pokud se živíte SW, je životně nutné znát autorský zákon (121/2000 Sb. ”Zákon o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů”) viz http://www.nkp.cz/o_knihovnach/00-121.htm
 - z § 2 vyplývá, že počítačový program (”je-li původní v tom smyslu, že je autorovým vlastním duševním výtvorem”) i jeho jednotlivé vývojové fáze a části jsou předmětem ochrany podle AZ; podle § 65 je chráněn jako dílo literární
 - § 5, § 11 a § 26: autorem je fyzická osoba, která dílo vytvořila, autorství nelze převést nebo se ho vzdát
* s AZ souvisí § 152 trestního zákona: ”Kdo neoprávněně zasáhne do zákonem chráněných práv k autorskému dílu ... bude potrestán odnětím svobody až na dvě léta nebo peněžitým trestem nebo propadnutím věci.”

* licence
 - § 12 a § 46: autor má právo své dílo užít a udělit jiné osobě smlouvou

licenci k jednotlivým způsobům nebo ke všem způsobům užití (užití podle AZ = rozmnožování, rozšiřování atd.; rozmnožování je podle § 66 i ”vytvoření rozmnoženiny (nezbytné) k zavedení ... programu do paměti počítače”)
. § 49: licence může být výhradní nebo nevýhradní (výhradní = autor nesmí poskytnout licenci třetí osobě)
. § 49: povinnou náležitostí licence je výše odměny nebo způsob jejího určení
. § 50: není-li v licenci řečeno jinak, platí pouze na území České republiky
. § 50: není-li určeno jinak, platí max. jeden rok (!!!)
- § 58: zaměstnavatel vykonává svým jménem a na svůj účet autorova majetková práva k dílu, které autor vytvořil ke splnění svých povinností k zaměstnavateli (není-li sjednáno jinak)
. není-li sjednáno jinak, zaměstnavatel může dílo zveřejnit pod svým jménem, upravovat atd.
. počítačové programy se považují za zaměstnanecká díla i tehdy, byla-li vytvořena na objednávku

* licence platná v právním řádu jiné země nemusí být platnou licencí podle českého AZ a naopak (zejména pokud v licenci chybí některé ze zákona povinné ustanovení)

* podle českého AZ vzniká právo autorské k dílu v okamžiku, ”kdy je dílo vyjádřeno v jakékoli objektivně vnímatelné podobě”
- naproti tomu v jiných jurisdikcích je požadováno uvést informaci o copyrightu ve tvaru: Copyright <rok zveřejnění> <držitel autorských práv>

* příklad licence platné v ČR:

Copyright 2004 Západočeská univerzita v Plzni

Západočeská univerzita v Plzni tímto poskytuje nabyvateli rozmnoženiny tohoto počítačového programu a jeho dokumentace (dále jen ”Software”) bezúplatně celosvětovou a časově neomezenou nevýhradní licenci ke všem způsobům užití Software včetně zejména práva Software rozmnožovat a rozšiřovat, s právem upravovat Software a měnit jeho název, spojovat Software s jinými díly a zařazovat Software do děl souborných.

❇

Použitá literatura:

1. Scott W. Ambler, Larry L. Constantine: *The Unified Process Elaboration Phase.* CMP Books, 2000.
2. Kent Beck: *Extreme Programming Explained.* Addison-Wesley, 2000.
3. Barry W. Boehm: *A Spiral Model of Software Development and Enhancement.* In IEEE Computer 21(1988), pp. 61–72.
4. Barry Boehm, Prasanta Bose: *A Collaborative Spiral Process Model Based on Theory W.* In Proceedings, 3rd International Conference on the Software Process, Applying the Software Process, IEEE 1994.
5. Barry W. Boehm, Tony Ross: *Theory-W Software Project Management: Principles and Examples.* In: IEEE Transactions on Software Engineering, vol. 15 no. 7, July 1989.
6. Grady Booch, James Rumbaugh, Ivar Jacobson: *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.
7. Frederick P. Brooks, jr.: *The Mythical Man-Month.* Essays on Software Engineering, Anniversary Edition. Addison-Wesley, 1995.
8. Brad J. Cox: *Object Oriented Programming.* An Evolutionary Approach. Addison-Wesley, 1987.
9. Tom DeMarco, Timothy Lister: *Peopleware.* Productive Projects and Teams, 2nd ed. Dorset House Publishing, 1999.
10. Norman E. Fenton: *Software Metrics.* Chapman & Hall, 1991.
11. Gail Freeman-Bell, James Balkwill: *Management in Engineering.* Prentice Hall, 1996
12. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
13. James Gosling, Henry McGilton: *The Java Language Environment.* A White Paper. Sun Microsystems, 1996
14. Ralph E. Johnson: *Frameworks = Components + Patterns.* CACM October 1997, Vol. 40, No. 10., pp. 39-42.
15. Steve McConnell: Code Complete. *A Practical Handbook of Software Construction.* Microsoft Press, 1993.
16. *OMG Unified Modeling Language Specification, Version 1.5.* OMG, 2003.
17. Meilir Page-Jones: *Základy objektově orientovaného návrhu v UML.* Grada 2001.
18. Petr Paleta: Co programátory ve škole neučí. Computer Press, Brno 2003.
19. Roger S. Pressman: *Software Engineering: A Practitioner's Approach, 6th edition.* McGraw-Hill, 2005.
20. Karel Richta, Jiří Sochor: *Softwarové inženýrství I.* Vydavatelství ČVUT, Praha 1996 (dotisk 1998).
21. James Rumbaugh, Nichael Blaha, William Premerlani, Frederick Eddy, William Lorensen: *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.
22. Václav Řepa: *Programování ve velkém.* Softwarové noviny 5/2003, str. 18-27.
23. Joseph Schmuller: *Myslíme v jazyku UML.* Grada, Praha 2001.
24. Stefan Sigfried: *Understanding Object-Oriented Software Engineering.* IEEE Press, 1996.
25. Ian Sommerville: *Software Engineering, 6th ed.* Addison-Wesley, 2001
26. SWEBOOK: *Guide to the Software Engineering Body of Knowledge, Trial Version.* IEEE 2004.