- Mechanismy rozšíření:
    - Omezení – podmínka nebo pravidlo omezující chování předmětu – složené závorky
    - Stereotypy – zavádějí nové elementy modelu založené na existujících elementech
    - Označené hodnoty – umožňují doplňovat elementy modelu o nové vlastnosti (klíčové slovo s novou hodnotou)

Pohled 4+1 na architekturu systému

- Logický pohled – funkce systému a slovník (důraz na množinu tříd a objektů implementující chování systému) – ekvivalence pohledu na zdroje a jejich organizaci a konkrétní vykonavatele v Kapitole 1
- Pohled procesů – procesově orientovaná varianta logického pohledu, modeluje spustitelná vlákna a procesy jako aktivní třídy, výkon, škálovatelnost a propustnost systému – ekvivalence pohledu na procesy v Kapitole 1
- Pohled implementace – modeluje soubory a komponenty utvářející hotový kód, znázornění závislostí mezi komponentami + správa konfigurace, definice verzí systému – ekvivalence definice jednotlivých aktivit; jejich vykonávání a správy meziproduktů
- Pohled nasazení – fyzické nasazení komponent, distribuce komponent – v reálném prostředí
- Předcházející pohledy sjednoceny v pohledu případů užití

Unified Process (UP)

- Pro každý nový projekt nová instance UP
- Řízen rizikem a případy užití
- Soustředění na architekturu
- Iterativní a přírůstkový (inkrementační)
- Každá iterace – požadavky, analýza, návrh, implementace, testování
- Čtyři fáze
    - Zahájení
    - Rozpracování
    - Konstrukce
    - Zavedení

UML Diagramy

- Následující obrázky jsou převzaty převážně z webu Embarcadero Developer Network http://edn.embarcadero.com/article/31863

Diagram případů užití

Případy užití

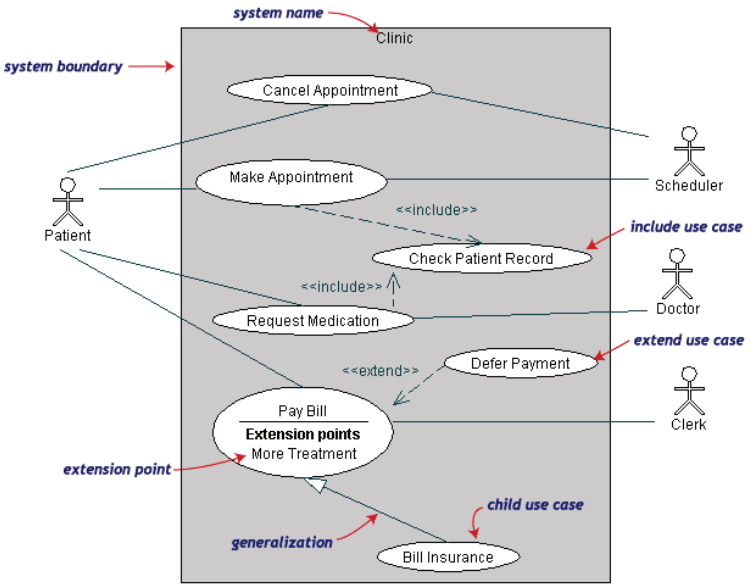
- používají se zejména pro popis kontextu systému a pro popis uživatelských požadavků
- jsou funkce, které systém vykonává jménem jednotlivých účastníků nebo v jejich prospěch
- reprezentují vnější pohled na systém, modelují zamýšlené fce systému a jeho vztah k okolí

Modelování případů užití – diagram případů užití

- Nalezení hranic systému
- Vyhledání účastníků (aktérů)
- Nalezení případů užití
- Specifikace případu užití
- Tvorba scénářů

Model případů užití – 4 komponenty

- Aktéři – role přidělené osobám nebo předmětům používajících daný systém
- Případy užití – činnosti, které mohou aktéři se systémem vykonávat
- Relace – smysluplné vztahy mezi aktéry a případy užití
- Hranice systému – ohraničení kolem případů užití, vymezení modelovaného systému



Obrázek 33 Příklad diagramu případu užití

Hranice systému

- co je a není součást systému
- aktéři vně, případy užití uvnitř
- 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

Aktéři

- uživatelé a další systémy, které mohou se SW systémem interagovat
- jejich znalost nám pomůže určit hranici systému a co má systém dělat
- aktérem nemusí být člověk, může to být i jiný systém nebo čas (např. spouštění zálohování v určitý okamžik)
- definují množinu rolí, kterou uživatelé systému mohou hrát při interakci se SW systémem (nejsou to ani konkrétní osoby, ani konkrétní předměty)
- komunikují bezprostředně se systémem
- jedna osoba nebo předmět může mít ve vztahu k systému více rolí

Případy užití

- co účastník od systému očekává
- jsou vždy iniciovány aktérem a jsou vždy napsány z pohledu aktéra
- popisují akci – slovesná vazba
- 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
- konkrétní popis případu užití je obvykle textový a je svázán odkazem z případu užití v diagramu
- při definici případů užití procházíme jednotlivé aktéry a zvažujeme způsob, jak budou se systémem interagovat – vznikne seznam případů užití

Relace

- účast aktéra na případu užití se nazývá "asociace" a značí se čarou spojující aktéra s případem užití

Slovníček pojmů

- poskytuje definice klíčových doménových pojmů (definice synonym a homonym)

Specifikace případu užití (detail užití)

- obsahuje (viz také kapitola 2 - Specifikace požadavků):
    - Název
    - Jedinečný identifikátor
    - Vstupní podmínky – omezení stavu systému, musí být splněny dříve, než je možné spustit případ užití
    - Tok událostí – jednotlivé kroky v případu užití
    - Následné podmínky – kriteria, která musí být splněna na konci případu užití

Komplexní případ užití

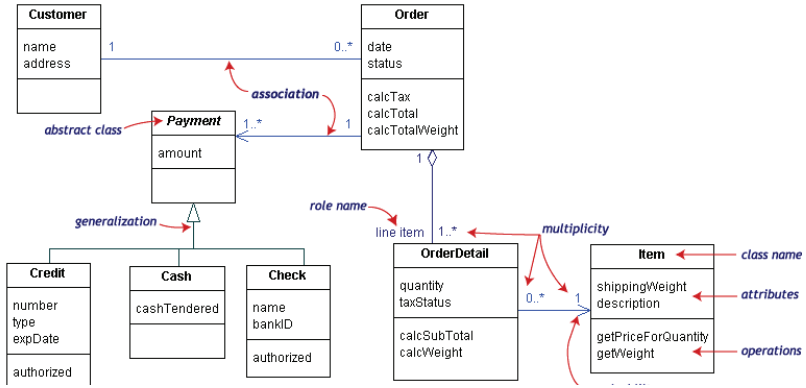
- lze rozložit na několik scénářů (v případě, že model obohatí)
- vždy hlavní scénář (předpokládá, že nedojde k alternativám) a alternativní scénáře (v případě výskytu výjimek, …)
- vedlejší scénáře lze najít zkoumáním hlavních scénářů

Modelování případů užití

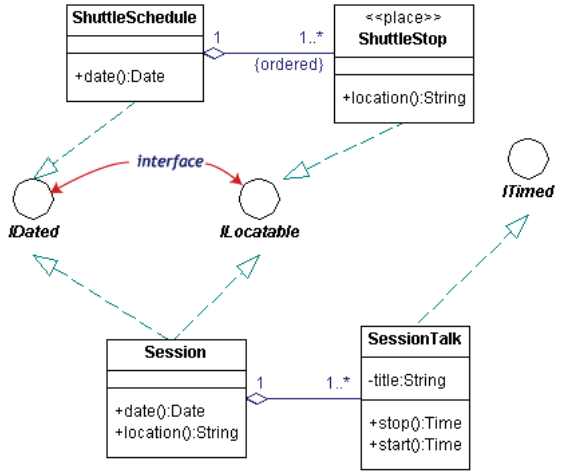
- vhodné u systémů, ve kterých
    - Převládají uživatelské požadavky
    - se vyskytuje mnoho aktérů
    - Je obsaženo mnoho rozhraní k dalším systémům
- nevhodné u systémů, u nichž
    - Převládají parametrické požadavky
    - Se vyskytuje málo aktérů
    - Je obsaženo málo rozhraní

Diagram tříd

- znáte z PT
- ukazuje statickou strukturu tříd v systému, tj. třídy, jejich vztahy (dědičnost, asociace, závislost), atributy a operace
- diagram je považován za statický proto, že struktura popisovaná diagramem platí v jakémkoli okamžiku běhu systému



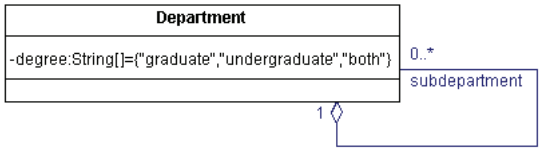
Obrázek 34 Příklad diagramu tříd



Obrázek 35 Příklad digramu tříd a zakreslení interface

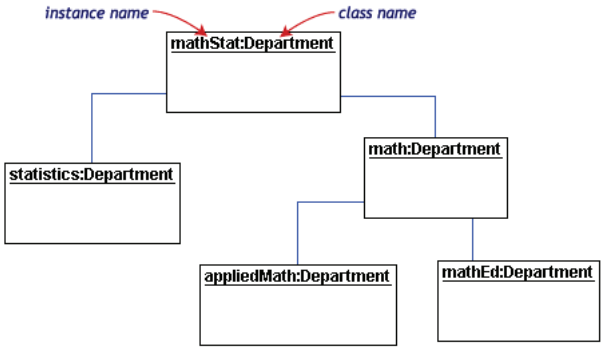
Diagram objektů

- je variantou digramu tříd, diagram objektů ukazuje konkrétní instance a jejich vztahy (linky neboli propojení)
- ukazují možný vztah objektů v nějakém okamžiku běhu systému
- používá se poměrně zřídka, vhodné pro vysvětlení malých částí systému se složitými (zejména rekurzivními) vztahy



Obrázek 36 Diagram objektů - příklad

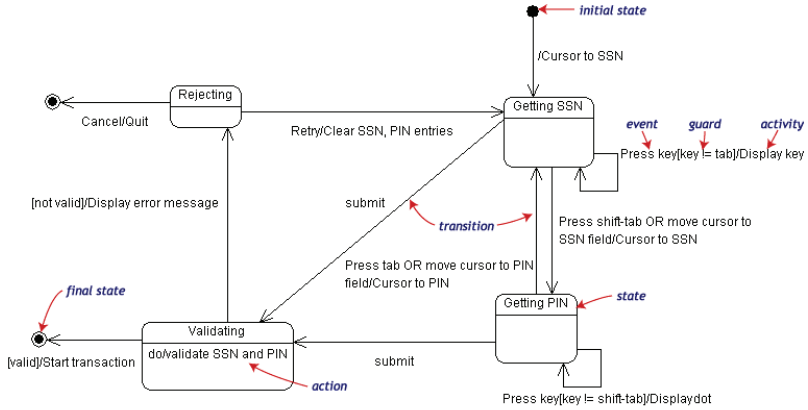
- link (propojení) je instancí asociace
- link je n-tice (nejčastěji dvojice) odkazů na objekty
- znázorňuje se čarou (podobně jako asociace)



Obrázek 37 Diagram objektů - příklad

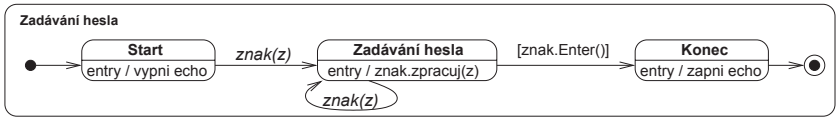
Stavový diagram

- typicky se používají pro popis chování instance třídy, někdy také pro případy užití nebo pro celý systém nebo podsystém
- ukazuje, jakými vztahy mohou instance třídy procházet během svého životního cyklu a jaké události mohou způsobit přechod mezi stavy
- událost může být způsobena zasláním zprávy objektu, např. pokud uplyne specifikovaná doba nebo může přechod nastat po splnění nějaké podmínky
- s přechodem může být spojena nějaká činnost objektu
- proti klasickým "plochým" stavovým diagramům umožňují stavové diagramy v UML strukturování
    - 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



Obrázek 38 Stavový diagram - příklad

- 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
2. orientované hrany grafu = přechody mezi stavy
3. 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)
4. počáteční stav - znázorněn jako černé kolečko
5. koncový stav - znázorněn jako černé kolečko v kroužku ("býčí oko")
6. činnost v daném stavu lze popsat opět pomocí stavového diagramu – vnoření - počáteční a koncový pseudostav je znázorněn stejně jako počáteční a koncový stav



Obrázek 39 Vnořený stavový diagram - příklad

7. v UML2 existují dva typy stavového diagramu:
    - stavový diagram chování (behavioral state machine) - popisuje životní cyklus třídy
    - stavový diagram protokolu (protocol state machine) - pravidla pro implementaci rozhraní nebo portů; usnadňuje komponentově orientované programování tím, že popisuje pravidla, která musejí ostatní systémy dodržet při komunikaci s asociovanou třídou

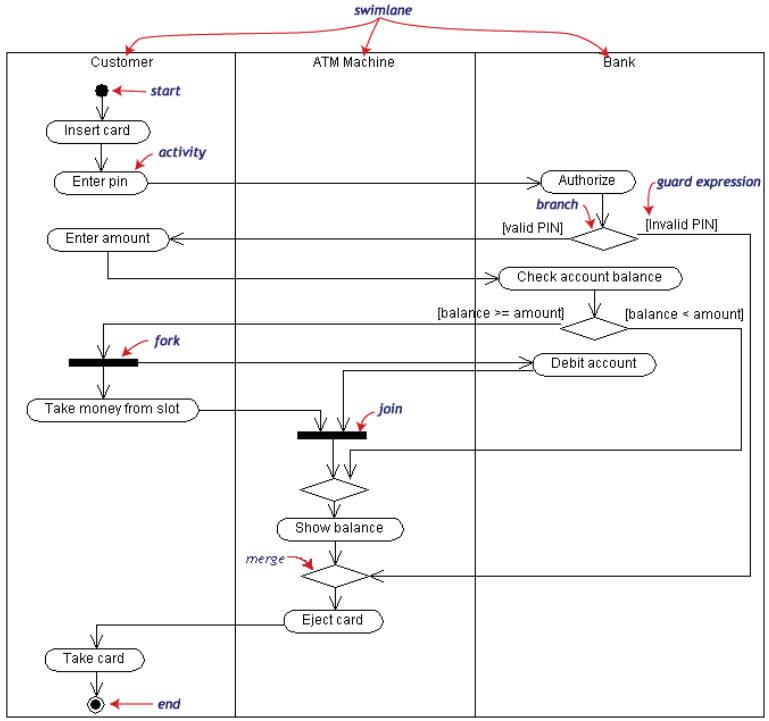
Diagram aktivit

- založen na notaci Petriho sítí
- znázorňuje sekvenční tok aktivit v rámci procesu

- umožňují modelovat paralelní chování, zpracování výjimek atd.

- při modelování se obvykle používají pro různé účely
    - pro popis akcí, které se budou vykonávat v průběhu operace, tj. popis interní činnosti objektu
    - pro popis příbuzných akcí a jakým způsobem ovlivní okolní objekty
    - pro znázornění instance případu užití
    - pro znázornění chodu firmy, tj. aktérů, organizace práce a objektů
- zóny – rozdělují zodpovědnost za aktivity mezi objekty
- aktivity = obdélníky s oblými rohy
- přechod mezi aktivitami – čára se šipkou spojující obdélníky
- hodnocení přechodu a alternativní cesty aktivit - kosočtverec, podmínka v []
- rozvětvení (fork) – rozvětvení aktivity do souběžných toků – všechny výstupní přechody se vykonávají současně
- spojení (join) – místo synchronizace souběžných toků

- stavový diagram a diagram aktivit se doplňují (stavový diagram popisuje objekt a změnu jeho stavů v rámci probíhání procesů, diagram aktivit popisuje tok aktivit a jejich vzájemnou závislost v rámci procesu)



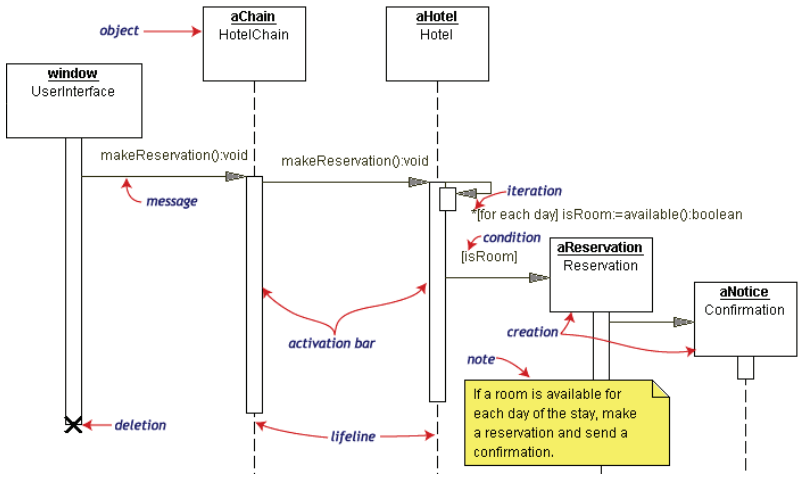
Obrázek 40 Příklad diagramu aktivit - výběr peněz z bankomatu

Sekvenční diagram

- ukazuje dynamickou spolupráci mezi množinou objektů
- má časovou osu - plyne shora dolů v diagramu
- diagram ukazuje posloupnost zpráv zasílaných mezi objekty
    - s objektem spojena svislá čára života
    - zprávy jako šipky mezi čarami života
    - na okraj diagramu lze zapisovat komentáře
- v některých metodikách (např. OOSE) se k případům užití vytvářejí sekvenční diagramy místo diagramů spolupráce

- 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
    - asynchronní komunikace - šipka tvořená čarami
    - návrat z procedury - přerušovaná šipka, pokud řízení toku procedurální (viz "návrh mechanismu řízení"), můžeme vynechat šipku pro návrat z procedury

- 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



Obrázek 41 Příklad sekvenčního diagramu

- 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

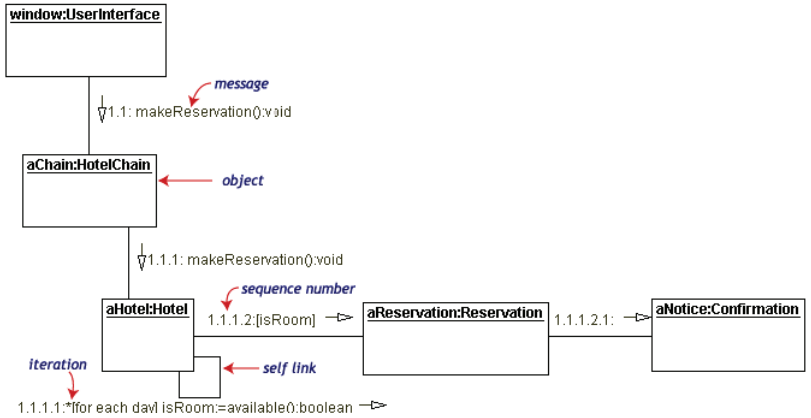
Komunikační diagram (Diagram spolupráce)

- podobně jako sekvenční diagram ukazuje výměnu zpráv
- na rozdíl od sekvenčního diagramu nemá časovou osu, zato ukazuje vztahy mezi objekty (linky); interakci mezi objekty znázorňují oba diagramy, tj. pokud je důležitá časová posloupnost, používá se sekvenční diagram, a pokud je třeba znázornit kontext, použijeme komunikační diagram

- popisují komunikaci mezi objekty nebo jejich rolemi
- mohou být připojeny např. k případu užití jako jeho popis
  - popisují, které objekty nabízejí chování popisované případem užití
  - jak objekty případ užití 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
  - zprávy (stimuly) - kreslí se jako šipka blízko čáry - šipka 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)

- 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]
- zprávy je možné také "číslovat" identifikátorem



Obrázek 42 Příklad diagramu spolupráce

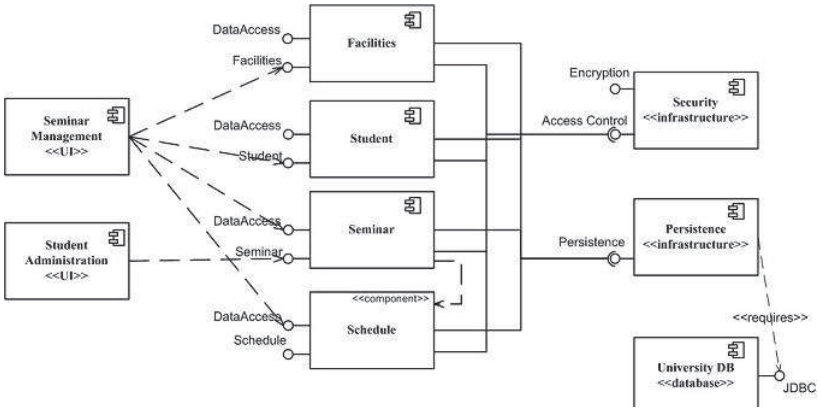
Diagram přehledu interakce

- umožňuje nahlížet na tok interakcí na vyšší úrovni
- je to v podstatě diagram aktivit, kde jsou hlavní uzly nahrazené fragmenty interakce nebo fragmenty sekvenčního diagramu
- nový diagram v UML2

Diagram komponent

- ukazuje organizaci a závislosti mezi komponentami nebo mezi komponentami a rozhraními komponent
- komponenty reprezentují dobře zapouzdřené prvky logické architektury
- 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. - tyto jsou modelovány jako artefakty

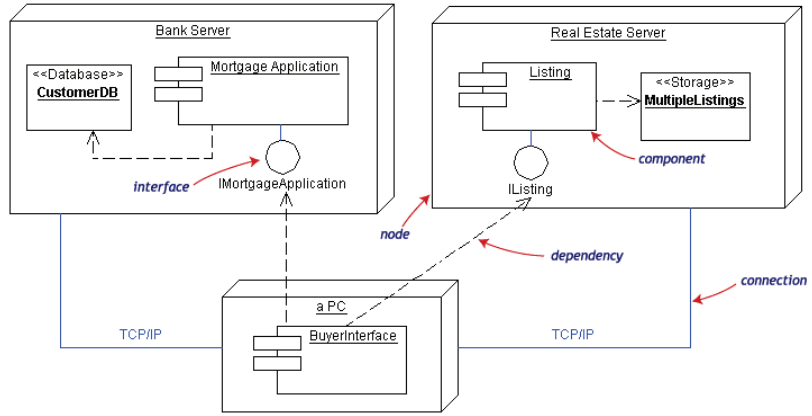
- komponenta se někdy znázorňuje jako obdélník se stereotypem <<component>>
- častěji bývá zobrazena ikona komponenty – obdélník, ze kterého po straně vyčnívají dva menší obdélníčky
- v UML 1.x se komponenty směly používat pouze pro strukturování modelu na fyzické architektuře; v UML2 se používají pro strukturování všech částí modelu (z toho důvodu už není zapotřebí speciální prvek "podsystém")



Obrázek 43 Příklad diagramu komponent

Diagram nasazení (Deployment diagram)

- ukazuje fyzickou architekturu hardwaru a SW v systému, např. počítače, zařízení, jak jsou propojeny, jaké artefakty budou umístěny na kterých počítačích
- zobrazuje uzly, komunikační cesty a nasazené artefakty

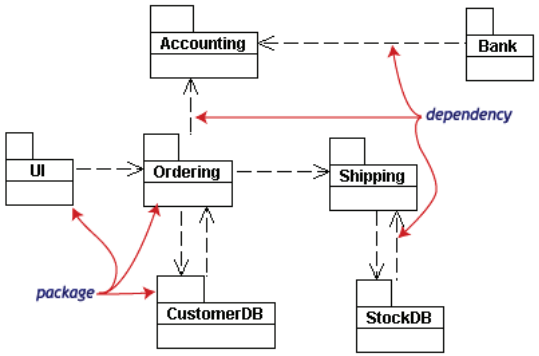


Obrázek 44 Příklad diagramu nasazení (Deployment diagramu)

Balíčky (packages)

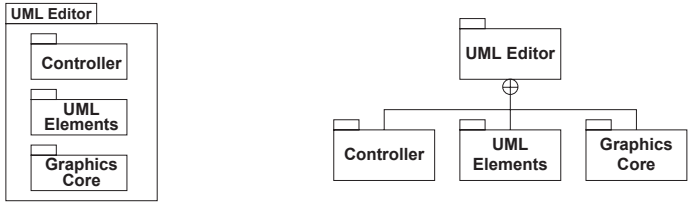
- slouží pro zjednodušení složitých diagramů
- sdružují množinu libovolných prvků diagramu (uvnitř balíčků mohou být další balíčky)
- každý prvek může být nejvýše v jednom balíčku

- základní rozdíl mezi balíčkem a komponentou: balíček je čistě koncepční mechanismus pro organizaci modelů v UML, zatímco komponenty existují skutečně
- balíčky se zakreslují jako obdélníky s malým držátkem na levé horní straně
-



Obrázek 45 Balíčky a jejich vzájemná závislost

- 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")¨



Obrázek 46 Balíčky, strom balíčků

UML nástroje

- **Nástroj Together Designer -** http://www.kiv.zcu.cz/~brada/tmp/sw/
- **Nástroj MagicDraw UML -** http://www.magicdraw.com
- **Poseidon for UML, Apollo for Eclipse -** http://gentleware.com/
- **Eclipse – Omondo -** http://www.omondo.de/

- **Eclipse UML plug-ins – přehled -** http://eclipse-plugins.2y.net/eclipse/plugins.jsp?category=UML&sort=hits24h

Literatura a použité zdroje

- Arlow J., Neustadt I., UML a unifikovaný process vývoje aplikací
- UML homepage - http://www.uml.org/
- UML tutorial- http://edn.embarcadero.com/article/31863
- Agile Modeling - http://www.agilemodeling.com/artifacts/

Objektově orientovaná analýza

- Navazuje (a dále souběžně probíhá) na definici sw produktu – specifikaci požadavků; neexistuje jasný předěl mezi specifikací požadavků a analýzou, představme si je spíše jako doplňující se a překrývající se pracovní postupy
- V UP konec fáze zahájení a fáze rozpracování
- Vycházíme především z případů užití a podnikatelských pravidel (znalostí domény)
- Cíl: analytický model (co má systém dělat) – zachycuje podstatné požadavky a charakteristické rysy požadovaného systému; můžeme jej modelovat v UML
- Hranice mezi analýzou a návrhem také není ostrá
- Výstup analýzy (analytický model)
- Analytický model
    - Vždy v jazyku domény
    - Zachycuje problém z určité perspektivy (nebezpečí zabývání se přílišnými detaily)
    - Obsahuje artefakty modelující problémovou doménu
        - Analytické třídy – klíčové pojmy v doméně, vyjadřují velmi přesně definované zobecnění entit problémové domény
        - Realizace případů užití – vzájemná komunikace instancí analytických tříd
    - Vypráví příběh o požadovaném systému (používáme vhodné diagramy)
    - Je užitečný pro maximální počet zainteresovaných osob
- Analýza v metodice UP zahrnuje architektonickou analýzu, analýzu případů užití, analýzu třídy a balíčku
- Analytický model a odhady
    - Středně velký systém – 50-100 tříd
    - Součástí pouze třídy, které modelují pojmy problémové domény (nenajdeme tu např. návrhové třídy)
    - Soustředíme se na třídy a jejich asociace, počet vazeb minimalizujeme (co nejjednodušší model)
    - Dědičnost použijeme pouze tam, kde existuje přirozená hierarchie abstrakcí

Analytická třída

- Obsahuje množinu hlavních kandidátů na atributy a množinu hlavních operací
- Z názvu je jasný její účel
- Modeluje jeden specifický element problémové domény
- Má definovanou malou a jasnou množinu (ideální počet je 3-5) odpovědností (odpovědnost je dohoda-závazek vůči klientům, který představuje sémanticky soudržná množina operací)
- Je vysoce soudržná (má jeden jedinečný účel)
- Má málo vazeb na okolní třídy
- Špatně navržená analytická třída
    - Má pouze jedinou operaci – tzv. funktoid
    - Má více než 5 odpovědností
    - Je všeobjímající (jmenuje se např. systém, software, apod.)
    - Obsahuje široce rozvětvený strom dědičnosti
    - Je málo soudržná

- Má úzké vazby na příliš mnoho dalších tříd

Metody hledání analytických tříd

- Shromažďování a následná analýza podstatných jmen (kandidáti na třídy a atributy) a sloves (kandidáti na odpovědnosti a operace)
- CRC karty (Class/Responsibilities/Collaborators) – spontánní diskuse + hledání nápadů (forma brainstormingu) nebo procházení scénářů případů užití
    - pro třídy vytvoříme kartičky, nahoru napíšeme jméno třídy, vlevo zodpovědnost třídy, vpravo spolupracující třídy
    - proces:
        - pojmenování doménových konceptů, uvedení odpovědností jednotlivých konceptů, označení spolupracujících tříd, vše – nejprve forma brainstormingu, poté analýza sebraných informací
        - NEBO (bez fáze brainstormingu): procházíme scénáře případů užití, přidáváme třídy a nové odpovědnosti existujícím třídám; pokud neumíme, rozdělíme existující třídu na dvě nebo založíme novou
    - nevýhoda - vazby mezi třídami nejsou znázorněny graficky (snažte se proto minimálně např. lepit lístečky na tabuli a kreslit mezi nimi spojovací čáry)
- přemýšlení o dalších typických zdrojích (fyzické objekty – auto, budova, výtah,…), kancelářské a obchodní záležitosti (faktura, objednávka, smlouva,…), známá rozhraní k vnějšímu světu (monitor, klávesnice, port,…) – důležité zejména u embedded systémů, koncepční entity, které nejsou uvedeny jako konkrétní fyzické předměty (věrnostní program)

- První analytický model- výsledek vzájemného porovnání výsledků jednotlivých metod

Používání relací mezi modelovanými elementy

- Relace je významová vazba (mezi třídami, objekty,…) – způsob spojování předmětů a abstrakcí dohromady
- Vazba mezi objekty je spojení (jeden objekt obsahuje odkaz na jiný objekt), příslušná vazba mezi třídami je asociace (spojení je instance asociace)
- Objektové diagramy ukazují objekty a jejich vzájemná spojení v určitém okamžiku
- Objekty mohou vůči sobě hrát různé role – role objektu při spojení definuje sémantiku jeho úlohy při spolupráci
- Asociace mezi dvěma třídami je totéž, jako kdyby měla jedna třída pseudoatribut, který nese odkaz na objekt druhé třídy (asociace používejte vždy, když je na druhém konci asociace důležitá třída, jejíž přítomnost chcete zdůraznit)
- Asociační třída (modelování vazby M:N) je asociací, která je rovněž třídou (může mít atributy, operace, relace) – je použitelná všude tam, kde v daném okamžiku mezi libovolnými dvěma objekty existuje jedno jedinečné spojení; pokud je třeba, aby mezi dvěma objekty bylo v jednom okamžiku více spojení, je třeba nahradit relaci normální třídou
- Závislosti jsou relace, kde se změna v dodavateli automaticky projeví rovněž v klientovi (tečkovaná šipka od klienta k dodavateli); univerzální stereotyp <<use>> - klient používá dodavatele jako argument, návratovou hodnotu nebo jako element vlastní implementace (např. vytvoření dočasného objektu)

Balíčky

- mechanismus jazyka UML pro seskupování předmětů
- vytváří např. sémantické hranice uvnitř modelu a poskytují zapouzdřené jmenné prostory, v nichž musí být všechny názvy jedinečné
- vytvářejí hierarchie
- analytické balíčky obsahují případy užití, analytické třídy, realizace případů užiti
- elementům balíčků je možné určit viditelnost
- existují stereotypy pro vyjádření relace závislosti mezi balíčky

Architektonická analýza

- Dělí soudržné množiny analytických tříd do analytických balíčků, které rozvrstvuje podle sémantiky
- Minimalizuje vzájemné vazby mezi balíčky (minimalizuje závislosti, počet veřejných a chráněných atributů, maximalizuje počet soukromých atributů)
- Hledá analytické balíčky
    - Průzkumem analytický tříd – tj. hledáním soudržných skupin úzce souvisejících tříd (zkoumají se hierarchie dědičnosti, kompoziční, agregační a závislostní vazby)
    - Průzkumem případů užití (hledání skupin případů užití zabývajících se konkrétním podnikatelským procesem nebo účastníkem, případy užití však často prochází přes více analytických balíčků
    - Maximalizací soudržnosti uvnitř balíčků (přesouvání tříd mezi balíčky, přidání nových balíčků, odebrání nepotřebných balíčků, odstraňování cyklických závislostí sloučením balíčků nebo vyčleněním určitých elementů do jiného balíčku)
    - Obsahuje ideálně 5-10 analytických tříd

Realizace případů užití

- Explicitní znázornění spolupráce skupin objektů pro dosažení požadovaného chování systému -> diagramy interakce (diagram spolupráce a sekvenční diagram) ukazují, jak třídy a objekty plní požadavky specifikované v případech užití
- Dynamický pohled na systém
- Spolupráce instancí analytických tříd
- Každá realizace zachycuje právě jeden případ užití
- Skládá se z následujících součástí
    - Diagram analytických tříd, které „vypravují“ příběh o případu (případech) užití
    - Diagram interakcí – spolupráce skupin objektů pro dosažení chování případu užití
    - Speciální požadavky odhalené v průběhu realizace případu užití
    - Upřesňování (změna) případu užití
- Digramy obecné interakce ukazují role klasifikátorů a asociací, zprávy a tok zpráv
- Diagramy konkrétní interakce ukazují instance klasifikátorů, spojení, tvorbu a uvolnění instancí a spojení, zprávy a tok zpráv, iterace a větvení
- Diagram spolupráce (zdůrazňují statické relace mezi instancemi a role jednotlivých instancí) a sekvenční diagram (popisuje chronologicky uspořádanou posloupnost zpráv předávaných mezi instancemi) jsou považovány za téměř zaměnitelné
- Modelování všech typů procesů – diagram aktivit – zachycuje jeden specifický aspekt chování systému – je to objektově orientovaný vývojový diagram

Příklad zadání – Bankomat (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.

Řešení:

- provedeme doménovou analýzu, cílem maximální porozumění doméně aplikace (seznámíme se s činností bankomatů, jejich zabezpečením apod.)
- navrhneme základní doménové třídy – doménový model, tj. třídy reprezentující objekty relevantní v aplikační doméně (často vytváříme zároveň s případy užití)
    - sledujeme podstatná jména v definici problému, věci a místa v aplikační doméně, pro každé vytvoříme předběžnou třídu; slovesa zaznamenáme, aby časem mohla být operacemi
    - vytvoříme třídy, jejich názvy, významné atributy, odpovědnosti, asociace mezi třídami a jejich násobnost
    - použijeme i další metody hledání analytických tříd (např. CRC kartičky)
- doménový model posléze rozvineme na model analytický (zároveň s detailním rozborem případů užití, přechod od specifikace k analýze)
    - 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 analytickými třídami se nesmí objevovat návrhové třídy a implementační konstrukce (např. seznam, pole, strom, tabulka apod.)

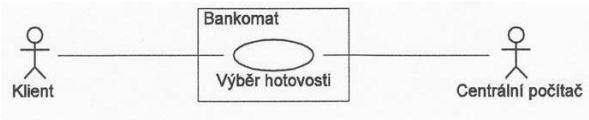
- předběžné třídy vyplývající z definice problému: software, bankomat, centrální počítač, banka, transakce, účet, záznam o činnosti, zabezpečení
- předběžné třídy vyplývající z aplikační domény: klient, platební karta, stvrzenka, výplata
- eliminujeme vágní třídy: software, zabezpečení
- CRC kartička

třída: Centrální počítač

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

- v našem případě nebudeme vytvářet balíčky a provádět s nimi architektonickou analýzu

- provedeme realizaci případů užití
- k dispozici máme
    - stručný popis účelu případu užití (cca odstavec)
    - rozložení případu užití na kroky (-> detailní popisy PU, přechod k analýze)
        - základní posloupnost aktivit, tj. kroky aktéra a odpovědi systému
        - alternativní posloupnosti aktivit
    - případy užití, 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



Obrázek 47 Diagram případu užití

Případ užití: PU1 - *Výběr hotovosti 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.

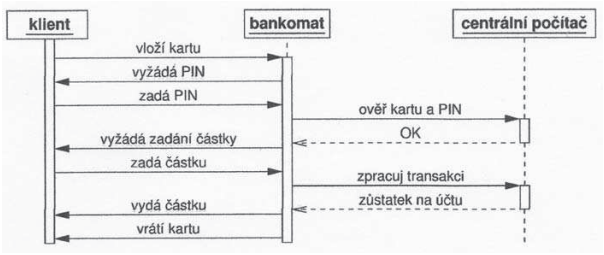
A7. Bankomat vytiskne stvrzenku a vrátí kartu.

- případy užití zjemníme dynamickými modely – realizace případů užití (diagramy interakce, diagram aktivit)
- vytvoříme diagram analytických tříd

Dynamické modely UML

- rozebereme základní sekvenci z případu užití
- alternativní sekvence většinou popisuje reakci na chyby, začleníme později
- opravujeme seznam tříd

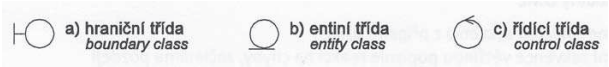
Příklad (sekvenční diagram pro případ použití "Výběr hotovosti z bankomatu")



Obrázek 48 Příklad sekvenčního diagramu

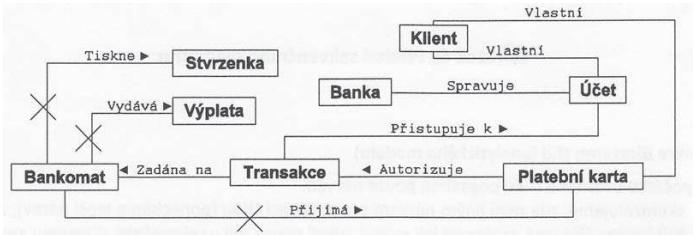
Konstrukce diagramu tříd (analytického modelu)

- na počátku uvažujme třídu popsanou pouze názvem
  - zkontrolujeme, zda není jiným názvem pro existující třídu (ponecháme lepší název), zda není rolí (jméno třídy má popisovat její esenci, nikoli pouze roli v sekvenčním diagramu apod.)
- ke třídě najdeme atributy = informace, která se nebude používat samostatně, ale je silně svázána s objektem (např. jméno, věk, adresa budou atributy nějaké Osoby)
- třídy můžeme rozdělit např. podle následujících stereotypů (jistá náhrada balíčků)
  - hraniční třídy
    - všechno, s čím aktéři přímo komunikují, např. formuláře, komunikační protokoly, rozhraní pro tiskárnu
    - v UML můžeme označovat stereotypem <<boundary>> nebo níže uvedenou značkou (případně oběma způsoby zároveň)
  - entitní třídy
    - informace, kterou systém udržuje delší dobu -> entitní objekty (odpovídají objektům reálného světa, např.: Student, Předmět, BankovníÚčet apod.)
    - entitní objekty samy od sebe neiniciují komunikaci
    - v UML stereotyp <<entity>>
  - řídící třídy
    - koordinují chování v systému
    - např. třídy obsahující řídící logiku, používající nebo nastavující obsah entitních tříd
    - obvykle se týkají realizace jediného případu užití
    - pokud je chování jednoduché, nemusejí být zapotřebí
    - v UML stereotyp <<control>>



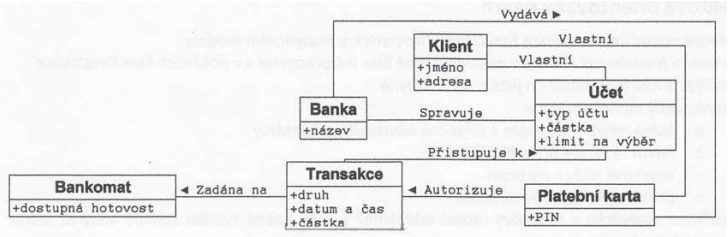
Obrázek 49 Rozdělení tříd analytického modelu

- entitní objekty – např. Klient a Účet
- výběr peněz ověřuje centrální počítač -> aktér (mimo hranice našeho systému), komunikace s tímto aktérem probíhá prostřednictvím tzv. ATM sítě, její rozhraní -> hraniční objekt
- hraniční objekty uživatelského rozhraní -> klávesnice, obrazovka, čtečka platebních karet, výdejní automat bankomatu, tiskárna stvrzenek apod. (vytvoření náčrtku, prototypu, který by si uživatel mohl vyzkoušet apod.)
- vytváříme předběžné asociace mezi třídami
  - mohou odpovídat fyzickému umístění nebo vztahu vlastnictví (má spojení s, je součástí, je obsažen v, patří), řízení a komunikaci (řídí, komunikuje s)
  - měly by být pojmenovány účelem asociace
  - nesnažíme se rozlišit asociace



Obrázek 50 Analytický diagram tříd - příklad

- zrušíme nepotřebné nebo nesprávné asociace
  - zrušíme asociace, které nejsou podstatné pro řešený problém (snažíme se o malý počet vazeb) nebo které představují popis implementace
  - zrušíme předběžné asociace, představující jednorázové akce; např. bankomat sice přijímá platební karty, ale mezi bankomatem a platební kartou neexistuje trvalý (strukturální) vztah (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
  - vyhýbáme se ternárním a vyšším asociacím, většinou je lze přestrukturovat na binární asociace nebo případně popsat asociační třídou
- hledáme primární atributy objektů a asociací
  - nejdůležitější logické atributy, relevantní pro aplikace - viditelné vlastnosti jednotlivých objektů (jméno, rychlost, barva apod.)



Obrázek 51 Analytický diagram tříd - příklad

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

Obrázek 52 Analytický model diagramu tříd - detail

- 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 jednoduchá sekvence akcí s jasným koncem, ale iterativní proces, ke kterému se musíme vracet po většinu doby vývoje systému.

„Analysis is frustrating, full of complex interpersonal relationships, indefinite, and difficult. In a word, it is fascinating.“

Objektově orientovaný návrh

- přesné určení implementace funkcí specifikovaných v analytickém modelu
- primární modelovací aktivita v poslední etapě fáze Rozpracování a v počátcích fáze Konstrukce
- analýza a návrh probíhají do jisté míry současně
- navrhovaný model obsahuje
  - jeden návrhový systém a případné návrhové podsystémy
  - návrh realizace případů užití
  - návrhové třídy a rozhraní
  - první verzi diagramu nasazení
- udržovat analytický a návrhový model odděleně? – ano, pokud systém splňuje alespoň jednu z následujících podmínek
  - je velký, komplexní nebo strategický
  - často se mění
  - bude mít dlouhou životnost
  - jeho části budou předány externím dodavatelům nebo výrobcům

- vytvoření návrhových tříd (design classes) – lze je implementovat
  - z analytických tříd - budou 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.)
  - z domény řešení – knihovny užitkových tříd, knihovny GUI, znovupoužitelné komponenty,…

Návrhové třídy

- obsahují
  - kompletní sadu atributů (název, typ, nepovinné implicitní hodnoty, typ viditelnosti)
  - metody (název, názvy a typy atributů, návratový typ, typ viditelnosti)
- jsou charakterizovány následovně
  - veřejné metody třídy definují dohodu s klientem
  - jsou úplné – poskytují vše, co se od nich očekává
  - jsou dostatečné – metody třídy jsou zaměřeny na realizaci zamýšleného účelu třídy
  - služby třídy jsou jednoduché, nedělitelné a jedinečné
  - jsou vysoce soudržné (mají definovanou minimální možnou množinu vlastností a metody jsou zaměřené ke splnění jediného účelu)
  - mají minimální počet vazeb k jiným třídám (použitelnost kódu v jiné třídě není důvod pro existenci vazby)
  - díváme se na ně očima klientů

- pro analytické třídy budeme vytvářet jednu nebo více návrhových tříd
  - návrh hraničních tříd
    - jedna hraniční třída bude odpovídat jednomu oknu nebo formuláři
    - jedna třída pro každé API nebo protokol
  - entitní (datové) třídy
    - často pasivní a perzistentní, vybereme pro ně způsob implementace: v souboru, v relační databázi (z perzistentních tříd vytvoříme tabulky atd.)
    - nejsou-li třídy perzistentní, pak budou implementovány v paměti (pro 3vrstvou architekturu rozhraní datové a aplikační vrstvy)
  - řídící třídy - obsahují aplikační logiku

Definice operací tříd

- standardní konstruktory se předpokládají => z návrhu je vynecháváme
- operace, které čtou a nastavují veřejné atributy, se (většinou) předpokládají
- hledáme operace - prvním vodítkem 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)
    - nakreslíme stavový diagram objektu nebo operace; např. stavový diagram pro bankomat

Obrázek 53 Stavový diagram pro bankomat

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

Definice atributů tříd

- vycházíme 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
    - snažíme se 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

- upřesňujte analytické asociace do návrhových asociací
    - upřesnění asociace do podoby agregace nebo kompozice (pouze u cyklů je ponechána neupřesněná asociace nebo je použita relace závislosti)
    - implementace asociací 1:1, 1:N, M:1, M:N, obousměrných asociací a tříd asociací
    - přidávání průchodností (řiditelností) - označuje se šipkou v asociačním vztahu, násobností a rolí asociací
- postup při upřesňování asociací
    - přidejte k asociaci násobnosti a názvy rolí (pokud nejsou k dispozici z analytického modelu)
    - rozhodněte, na které straně asociace je celek a kde součást
    - vyhledejte násobnost na straně celku, jeli rovna hodnotě 1, je možné použít kompozici, jinak je nutno použít agregaci
    - přidejte řiditelnost od celku k součásti – navrhované asociace musí být jednosměrné

Obrázek 54 Agregace

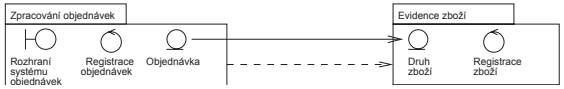
- pokud třída obsahuje více než cca 10 atributů, 10 asociací nebo 20 operací, není dobře navržená a potřebuje rozdělit

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
- dědění použijeme pouze v případě jasné relace „je“ (dědění je nejsilnější možná forma vazby, je nepružné, znamená často nepřijatelná omezení), podtřídy reprezentují speciální druh dané třídy, nikoli roli
- vícenásobné dědičnosti se raději vyhněte (realizovatelná je v C++)
- navrhujte a realizujte rozhraní (je to dohoda bez implementačních detailů) – flexibilnější

Vytváření balíčků

- pro lepší srozumitelnost (strukturování),
- pro izolaci částí, které se budou častěji měnit (případně obojí)
- do balíčku vkládáme funkčně příbuzné třídy (změna chování jedné třídy způsobí změnu chování druhé)
- třídy uvnitř balíčku veřejné (public) i soukromé (private)
    - veřejná třída může mít asociace s libovolnou jinou třídou; 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.



Obrázek 55 Balíčky a jejich závislosti

Definice rozhraní a podsystémů

- rozhraní odděluje specifikaci od implementace, lze jej připojit k třídám, podsystémům, komponentám,…- definuje služby poskytované těmito entitami
- jestliže klasifikátor (např. třída) uvnitř podsystému nebo komponenty implementuje veřejné rozhraní, pak se říká, že toto rozhraní je implementováno podsystémem nebo komponentou¨
- návrh zamžený na implementaci (propojení specifických tříd, vytvoření jednoduchého modelu) vs. návrh zaměřená na dohodu (třídy připojeny k rozhraní, které má různé realizace, flexibilní, ale složitější model)
- postup při hledání rozhraní
    - napadněte každou asociaci a každé odeslání zprávy
    - vyčleňte skupiny operací, které lze použít i jinde
    - vyčleňte operace, které se opakují ve více třídách
    - vyhledejte třídy, které mají v systému stejnou roli
    - hledejte možnosti pro budoucí rozšíření
- používejte rozhraní i k ukrývání detailů jednotlivých podsystémů
- podsystém jsou typem balíčků, používají se např. k vyjádření obsáhlých komponent, zapouzdření starších systémů, osamostatnění částí systému,…

Realizace případů užití

- rozšíření analytické realizace případů užití
- seskupení spolupracujících návrhových objektů a tříd sloužících k realizaci případů užití (návrhových diagramů interakcí, návrhových diagramů tříd)
- návrhové diagramy interakcí lze použít k modelování ústředních mechanismů, např. persistence objektů (tyto mechanismy mohou zasahovat mnoho případů užití)
- můžeme vytvářet návrhové diagramy interakce podsystémů

Diagramy aktivit a stavové diagramy

- diagramy aktivit jsou speciálním případem stavových diagramů
- reaktivní objekty (reagují na vnější události, mají určitý životní cyklus, jejich aktuální chování vyplývá předešlého chování) poskytují kontext stavového diagramu
- lze modelovat i složené stavy vnořenými stavovými diagramy

Kontrola modelu

- ověřit realizace případů užití, v návrhu nám nesmí chybět chování potřebné pro některý případ užití

Strukturovaná analýza systému

- strukturované metodiky pro analýzu a návrh systému historicky předcházely objektovým metodikám
- na funkce a na data se zaměřují víceméně odděleně
    - odpovídá strukturovanému programování
    - funkce jsou aktivní a mají chování
    - data jsou pasivní, ovlivněna funkcemi
- funkce systému postupně rozdělujeme shora dolů na části, nejčastěji pomocí diagramů datových toků (data-flow diagrams)
- dobře slouží 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 (pokud se naopak budou měnit data, je OO přístup výhodnější, protože změny jsou zapouzdřeny do jednotlivých objektů)

- výhoda strukturovaných metodik
    - mohou být jednodušší, vytvářené modely mohou být srozumitelné zákazníkovi => zákazník se snáze účastní strukturované analýzy (??)
    - návrh systému může být rychlejší (nevytváříme přídavnou strukturu tříd)
- nevýhody strukturovaných metodik
    - jsou považovány za nemoderní
    - výsledný systém se většinou hůře udržuje
    - nižší stupeň abstrakce -> složitější analýza, návrh,… velkých systémů

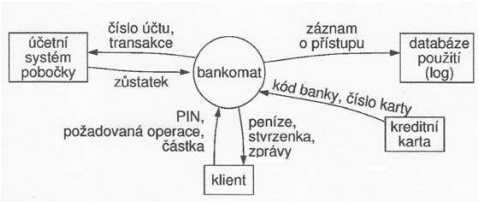
- tvorba modelu = abstrakce klíčových vlastností studovaného systému (vstup do dalších fází SW procesu):
    - model kontextu systému - určuje hranice vytvářeného systému
    - modely strukturované analýzy: diagramy datových toků, datový slovník, ERA diagramy, specifikace činností procesů
    - modely strukturovaného návrhu: strukturogramy (structure charts) – na jejich základě kódujeme
- vytváření modelu - podpora CASE nástrojů (editor modelů, částečná kontrola modelu, automatická tvorba dokumentace), srovnání některých CASE nástrojů podporujících strukturované metodiky v článku V. Řepy: Programování ve velkém, Softwarové noviny, 5/2003.

Model kontextu systému

- hranice systému, tj. co bude tvořit systém a co bude okolí systému (již ve specifikaci požadavků)
- v některých případech nemusí být úplně zřejmé, 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 - kontext diagram)
- hranice vytvářeného systému - kolečko s vepsaným názvem systému
- lidé, organizace nebo jiné systémy, se kterými náš systém komunikuje (terminátory) - pojmenovaným obdélník

- vstupující a vystupující data - šipky mezi systémem a terminátorem

- př. model kontextu systému pro bankomat:



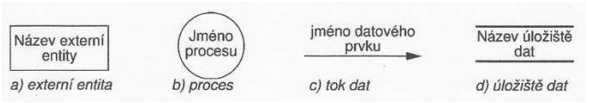
Obrázek 56 Bankomat - kontext systému

Diagramy datových toků (data-flow diagram, DFD)

- také data-flow graph, work flow diagram, function model, bubble chart, process model…
- součást mnoha strukturovaných metod cca 1955-1990 (ale i některých OO metodik, např. OMT), různé notace
- jednoduché a intuitivní (je možné je vysvětlit zákazníkovi)
- použitelné pro modelování toku dat SW systémem, dokonce i modelování toku dat a fyzických předmětů (materiálu) v organizaci
- 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. (tj. odpovídá pojmu "aktér" z případu užití)
    - znázornění obdélníkem s názvem uvnitř
- proces
    - provádí transformaci dat
    - znázornění kolečkem (bublinou)
        - je vhodné bubliny číslovat
        - je nutné bubliny 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

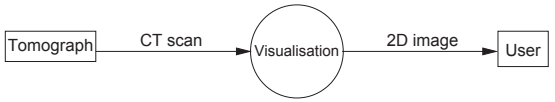


Obrázek 57 Základní notace DFD

Použití DFD

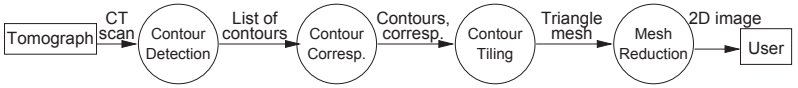
- DFD se používá pro reprezentaci systému libovolné úrovně abstrakce
- DFD úrovně 0 = fundamentální model systému, model kontextu systému
    - celý SW systém je zakreslen jako jedna bublina, má jeden nebo více vstupů a výstupů
- DFD dalších úrovní
    - systém rozdělíme do menších částí a znázorníme na větší úrovni podrobnosti (model je postupně zjemňován)
    - vytvořené DFD by neměly být příliš velké - měly by se bez problémů vejít na papír velikosti A4

- Př. Vizualizace snímků z tomografu (vstup - 2D řezy tělěm pacienta, výstup -2D pohled na snímek)



Obrázek 58 DFD úrovně 0

- Systém lze znázornit na větší úrovni podrobnosti (model může být postupně zjemňován)



Obrázek 59 DFD úrovně 1

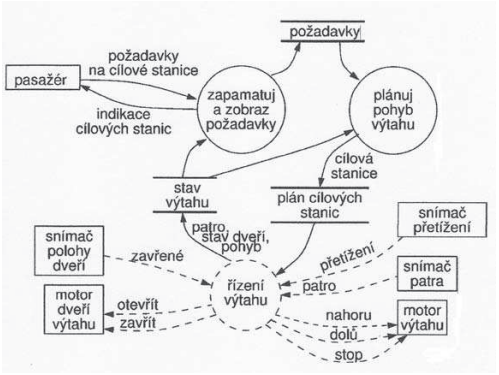
Co by se v DFD nemělo vyskytovat

- černé díry - bubliny, které mají vstup, ale nemají výstup
- spontánní generátory - mají výstup, ale žádný vstup; jediný rozumný případ je generátor náhodných čísel
- neoznačené procesy a datové toky – vyskytují se zejména proto, že analytik není schopen najít pro ně rozumný název, např. sdružují nesouvisející procesy/toky dat (proces nebo datový tok je nutno rozdělit)

Rozšíření DFD

- výše uvedená základní varianta DFD je vhodná pro modelování systémů řízených datovými vstupy prakticky bez žádné vnější události (typicky obchodní aplikace).
- Pro RT systémy celá řada rozšíření základního DFD (tok řízení znázorněný přerušovanou šipkou a řídící proces jako bublina zakreslená přerušovanou čarou

- př. Řízení výtahu



Obrázek 60 DFD Řízení výtahu

- dále potřebujeme
  - vytvořit definici dat
  - specifikovat logiku procesů (na konečné úrovni zjemnění doplňujeme popis specifikací procesu buď v přirozeném jazyce, nebo v pseudokódu)

Datový slovník

- metoda reprezentace obsahu dat v datových tocích a pamětech
- seznam datových prvků s definicemi

Důležitost popisu dat viz Yourdonův "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.

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 "Karel IV."?

-> vytváříme formálnější definici datového slovníku, data

  - jednoduchá - známé typy dat, je třeba zadat obor hodnot, příp. použité jednotky, příp. přesnost
  - složená - popíšeme odkazem na jednoduché položky

- Pak např. Jméno osoby je 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. | Ph.D.]
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 | ®]
malé_písmeno = *malá písmena české abecedy*
[a | á | b | c | č | ... | z | ž]

- 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 datového slovníku 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
- dvě (stejné) definice, je nebezpečí, že při změně jednu z nich zapomeneme opravit

ERA diagramy

- popisují strukturu uchovávaných dat na vysoké úrovni abstrakce
- základy (Chen 1976), později vývoj různými směry - existuje mnoho notací
- slouží jako vstup pro návrh databáze, např. "pamětí" v DFD
- podrobně viz Databázové systémy nebo např. Jaroslav Pokorný: Konstrukce databázových systémů. Vydavatelství ČVUT, 1999.

Specifikace činnosti procesů

- zjemňování DFD - rozklad problému na elementární procesy komunikující pomocí datových toků
- nutné specifikovat, co se děje v elementárním procesu
- použití nástrojů
  - pseudokódy nebo jejich grafický ekvivalent
  - rozhodovací tabulky
  - rozhodovací stromy
  - konečný automat (konvence stavového diagramu z UML)

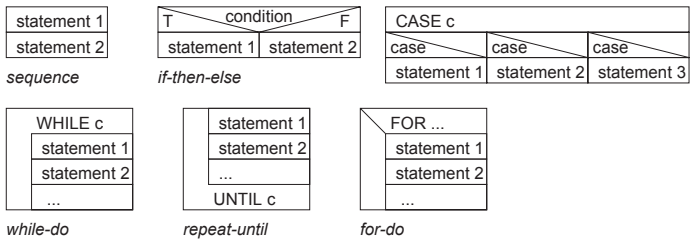
Pseudokódy

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

- 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, proces pravděpodobně není elementární

Nassi-Shneidermanovy diagramy

- někdy se používají místo pseudokódu
- po vysvětlení jsou pro zákazníky často srozumitelnější než pseudokód (podle publikovaných výzkumů pro 75-80% lidí)
- vyžadují nemalé množství grafiky - vyplatí se pouze tehdy, máme-li SW podporu pro jejich vytváření
- neměli bychom překročit 3 úrovně vnoření



Obrázek 61 Nassi-Shneidermanovy diagramy

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
- najdeme všechny relevantní vstupy, určíme počet možných kombinací
- vytvoříme tabulku
  - sloupce = pravidla: počet pravidel je dán počtem kombinací vstupů
  - řádky tabulky: nejprve všechny vstupy, pak všechny výstupy
- horní polovina tabulky definuje podmínky
  - ve spodní polovině je pro každý sloupec (tj. podmínku) uveden výstup

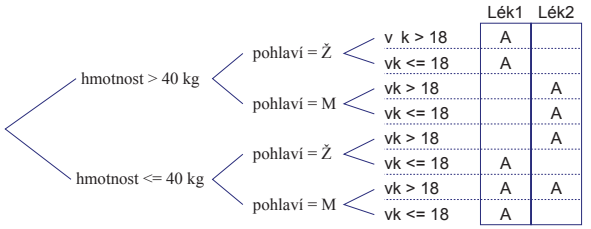
	1	2	3	4	5	6	7	8
věk > 18	A	N	A	N	A	N	A	N
pohlaví = Ž	A	A	N	N	A	A	N	N
hmotnost > 40 kg	A	A	A	A	N	N	N	N
podávat lék 1	A	A				A	A	A
podávat lék 2			A	A	A		A	

Obrázek 62 Rozhodovací tabulka

- někdy bývá srozumitelnější rozhodovací strom:

- výhoda těchto způsobů
  - specifikací není určen způsob implementace
  - s uživatelem můžeme probrat jedno pravidlo po druhém
  - máme jasnou odpověď pro všechny kombinace podmínek

- př. rozhodovací strom



Obrázek 63 Rozhodovací strom

Vyvažování modelů

- používáme
  - 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 - modelují funkce systému
    - datový slovník, ERA diagramy - modely pro 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
  - 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

Strukturovaná analýza

- využití nástrojů uvedených výše
- dvě známé metodiky
  - klasická DeMarcova metodologie
  - Yourdonova "moderní strukturovaná analýza"

Klasická DeMarcova metodologie

- vstupem uživatelské požadavky, výstupem 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

- co má analytik vyrobit – model původního systému nebo nového?
- předpokládejme, že
  - existující systém s ne zcela zřetelnou strukturou plní určité funkce
  - systém z nějakého důvodu nahrazujeme novým systémem
  - jak najít specifikaci nového systému?

- vytváření 4 modelů systému
  1. fyzický model stávajícího systému (jaký systém používá zákazník?)
     - analytik zmapuje stávající systém, jeho funkční strukturu a data
     - model může obsahovat zpracování a přesun fyzických formulářů apod.
  2. logický model stávajícího systému (jeho logická struktura)
     - z fyzického modelu vytvoříme logický (cca 75% redukce)
     - zrušíme všechny implementační detaily, tj. modelujeme, 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 funkce
     - 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 (což je koneckonců pravda)
- 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 "oddechu" prodlužuje, což snižuje ochotu spolupracovat s analytikem (tj. 4 modely je prostě moc)
- zmíněné problémy řeší "moderní strukturovaná analýza" (Yourdon 1989)

Moderní strukturovaná analýza

- místo čtyř modelů systému se zaměřuje přímo na nalezení esenciálního modelu systému, tj. logického modelu nového systému
- kroky
  - vytvoříme model prostředí (definuje hranici mezi systémem a zbytkem světa)
    - definujeme účel systému (ne 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

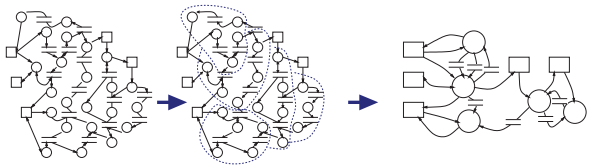
- události se klasifikují na
  - událost datového toku (F) – událost, která se projeví příchodem dat; např. událost "přišla objednávka"
  - časová událost (T) - např. zpracování transakcí mezi bankovními účty nastane ve 3:00
  - řídící událost (C) - asynchronní událost např. v RT systémech
- identifikace 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 užití v OO analýze)
- 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)
- pro každou událost se 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

- ne postup shora-dolů, ale identifikace odpovědí na události
    - pro každou událost ze seznamu nakreslíme bublinu, očíslujeme jí podle čísla události
    - bublinu pojmenujeme podle odpovědi na událost (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 funkci provádí člověk nebo 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, budou tvořit bubliny v DFD vyšší úrovně
        - každý agregát se týká úzce příbuzných odpovědí na události, tj. většinou obsahuje procesy zpracovávající příbuzná data
        - sdružovat bychom měli skupiny po cca 7+-2 procesech+pamětech, při tom máme příležitost "schovat" paměti, které jiné procesy nepotřebují



Obrázek 64 Proces zjednodušování DFD

- někdy procesy naopak nejsou primitivní a vyžadují další rozdělení (většinou jsme bublinu nedokázali dobře pojmenovat, proto jí rozdělíme na primitivní procesy)
- vytvoříme specifikaci procesů
- dokončíme datový slovník, ERA model
- výše uvedené informace tvoří tzv. esenciální model systému

Vytvoření uživatelského implementačního modelu

- alokace esenciálního modelu na lidi a stroje (které bubliny a paměti budou realizovány manuálně) - rozhoduje 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. parametrické (mimofunkční) požadavky na aplikaci
- oficiálně poslední krok moderní strukturované analýzy požadavků
- následuje návrh architektury, podrobný návrh, implementace, testování

- při analýze nemusejí být použity všechny nástroje (modely) - používají se jen ty, které mají v daném kontextu smysl
- další metodiky pro strukturovanou analýzu
    - např. SADT, SREM/RDD, SA/SD
    - v ČR asi nejpoužívanější SSADM (Structured Systems Analysis and Design Method); podobný záběr a nástroje jako DeMarcova a Yourdonova strukturovaná analýza (výstupy - DFD, model entit,…), je to standard - velmi podrobně definované kroky včetně výstupů a předepsaných kontrol před přechodem k dalšímu kroku.
    - použití SSADM v některých zemích podmínkou pro získání státních zakázek.

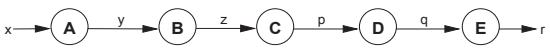
Strukturovaný návrh systému

- také funkčně orientovaný návrh (function-oriented design)
- vstupem 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

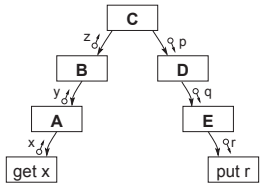
- v rámci podsystémů transformujeme DFD na hierarchii podprogramů (každá bublina se stane podprogramem)

- např. z DFD



Obrázek 65 DFD diagram

vzniknou podprogramy s následující hierarchií

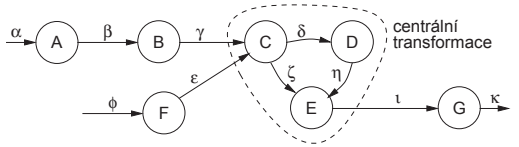


Obrázek 66 Hierarchie podprogramů na základě DFD

Transformace DFD na hierarchii podprogramů

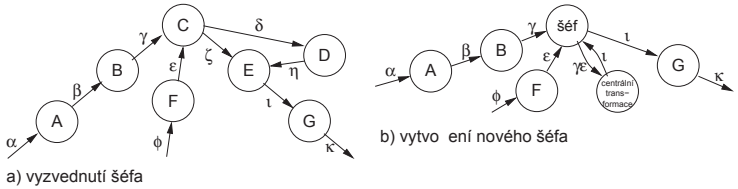
v obecném případě není převod DFD tak přímočarý, používá se následující postup

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



Obrázek 67 Centrální transformace v DFD diagramu

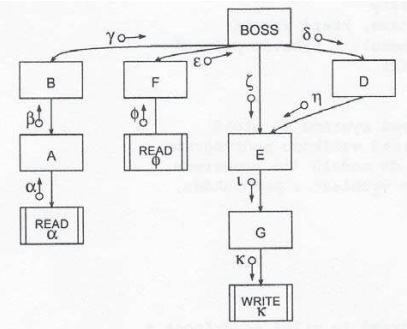
- *vytvoříme hierarchii procesů*
  - vyjadřuje vztah nadřízenosti a podřízenosti
  - nejprve musíme najít kandidáta na šéfa
    - pokud je dobrý kandidát (v centrální transformaci), zvedneme ho a necháme ostatní bubliny viset dolů, viz obrázek (a)
    - pokud je potenciálních kandidátů více, vyzkoušíme,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



Obrázek 68 Vytvoření hierarchie procesů

- 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))

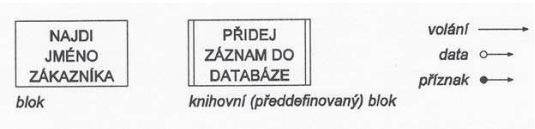
- *transformace DFD na hierarchii bloků (každá bublina se stane blokem)*
  - "šéf" se stane řídícím blokem, ostatní budou jeho podřízení
  - vytvoříme počáteční strukturogram



Obrázek 69 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
    - např. názvy bloků - postupně: "získej položku" (blok pro čtení dat), "získej transakci" (pomocí nižšího bloku vytvoří transakci), "získej platnou transakci" (ověří platnost transakce)

- *strukturovaný návrh se znázorňuje pomocí tzv. strukturogramů*
    - ukazují rozdělení systému do hierarchie bloků a jejich vzájemnou komunikaci
    - blok představuje podprogram programovacího jazyka; blok se znázorňuje jako obdélník obsahující název bloku
    - knihovní procedury a funkce (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



Obrázek 70 Význam jednotlivých elementů strukturogramu

- *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

- *výše uvedené kroky provedeme pro jednotlivé DFD, máme k dispozici množinu nezávislých strukturogramů*
    - vytvoříme nadšéfa, který bude volat jednotlivé šéfy
    - optimální je např., pokud nadšéf obsahuje konstrukci case, která vyvolá některého podřízeného (např. výběrem položky v menu)

- *výsledný produkt strukturovaného návrhu*
    - rozdělení systému do bloků
    - jeden nebo více strukturogramů, z bloků vzniknou podprogramy
    - neřeší sdružení podprogramů do modulů nebo návrh vnitřku podprogramů (zde můžeme vycházet z pseudokódu)

Vytváření modulů

- modul = množina dat a podprogramů, například zdrojový soubor v C - bývá obvykle definován jako nejmenší jednotka zdrojového textu programu, kterou můžeme samostatně přeložit

Vylepšení modularity podsystému

- viz provázanost, soudržnost …později

Návrh architektury systému

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

Rozdělení systému do podsystémů

- provádíme pro všechny větší aplikace
- podsystém bude obsahovat aspekty systému s nějakými podobnými vlastnostmi (podobná funkčnost, stejné fyzické umístění apod. – např. kosmická loď bude obsahovat podsystémy pro podporu životních funkcí, pro navigaci, pro řízení motorů, pro řízení experimentů apod.)
- podsystémů by nemělo být moc (i pro velké systémy cca do 20)

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

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

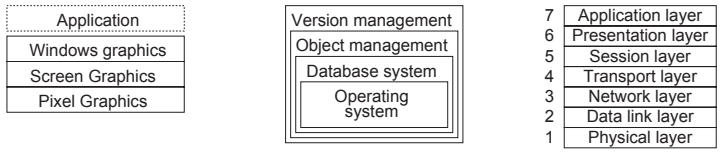
- základní rozdělení do horizontálních vrstev nebo vertikálních oddílů

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. podsystémy poskytují obdobné služby na různých úrovních abstrakce)

Příklad (interaktivní grafický systém) - vrstvy

- 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



Obrázek 71 Příklady vrstvené architektury

Dva typy vrstvených architektur

- *uzavřené (striktně vrstvené)* - vrstva je implementována pouze prostředky nejbližší nižší vrstvy
    - omezuje závislosti mezi vrstvami = princip skrývání informací
    - dovoluje snadnější změny rozhraní - změna ovlivní jen nejbližší vyšší vrstvu
    - například síťové modely (sedmivrstvý ISO/OSI model)

- *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 (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 vyšší vrstvy (= 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 (populární třívrstvá architektura), pro velké systémy i více vrstev (i pro nejsložitější systémy je podezřelé více než 10 vrstev)

Poznámka (doporučení z RUP)

- 0 - 10 tříd 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 služby prostředí, přepsáním této vrstvy můžeme systém přenést na jinou platformu

Rozdělení do oddílů

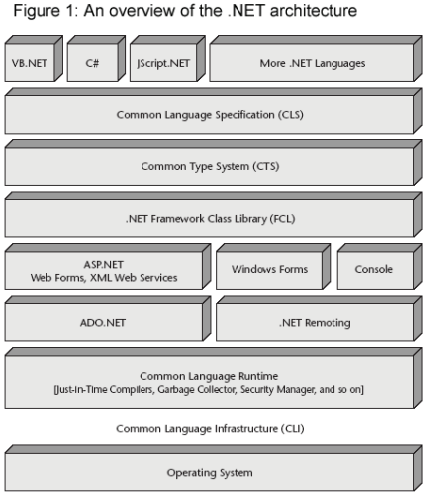
- oddíly (partitions) rozdělují systém vertikálně na nezávislé nebo slabě svázané podsystémy, každý z nich poskytuje jiný typ služeb (operační systém obsahuje ovladače pro jednotlivé typy zařízení)

Obrázek 72 Rozdělení do oddílů

- 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



Obrázek 73 Vrstvy a oddíly -.NET architektura

Topologie systému

- po identifikaci základních podsystémů určujeme toky dat
- data mohou „téci" mezi všemi podsystémy, v praxi však jen zřídka
- většinou jednodušší uspořádání (jednoduchá roura - např. překladač, hvězda - např. hlavní systém řídí podřízené) apod.
- konkrétní příklady později

Identifikace paralelismu

- identifikace podsystémů, které budou 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

- 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

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 (pokud je aplikace paralelizovatelná)

Podsystém jako HW nebo SW

- HW můžeme považovat za pevně zadrátovanou 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 spolu nejvíce komunikují, by měly být přiřazeny stejné jednotce
- určíme druh a relativní počet fyzických jednotek

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 (např. účetní systém může používat relační databázi pro komunikaci mezi podsystémy nebo aplikace pro zpracování obrazu může používat soubor - matici pixelů)

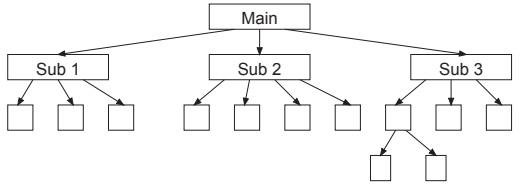
- *soubory*
    - levné, jednoduché a permanentní
    - nízká úroveň abstrakce - v aplikaci je nutný další kód pro práci s nimi
    - vhodné pro data, která jsou objemná, ale zároveň obtížně strukturovatelná v terminologii DB systémů
    - vhodné pro data, která mají malou informační hustotu, nebo data, která jsou uchovávána krátkou dobu
- *databáze*
    - 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 většinou v imperativním programovacím jazyce)

Výběr mechanismu řízení

- tři způsoby řízení v souvislosti s externími událostmi
    - 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
- používají 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)



Obrázek 74 Systém řízený procedurálně

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
- řízení událostmi ve skutečnosti simuluje spolupracující vlákna uvnitř úlohy, ale na rozdíl od skutečného paralelismu zablokuje dlouho trvající procedura celou aplikaci
- téměř všechny GUI (Windows), simulace apod.

- jednoduchá obsluha nových typů událostí
- vede přirozeně k objektově orientovaným systémům - události se posílají jako zprávy jednotlivým objektům

Paralelní systémy

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

Poznámka:

- je dobré (někdy i nutné), aby pro každý projekt existovala osoba, která je zodpovědná za architekturu systému (šéf-architekt)

Architektonický styl (architectural style, macro-architectural pattern)

- znovupoužitelná abstrakce systému
- vyskytuje se opakovaně v různých aplikacích (typické řešení)
- 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 manažerské rozhodování (např. pro rozdělení do týmů, organizaci dokumentace projektu apod.)
- 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)

Obecné struktury

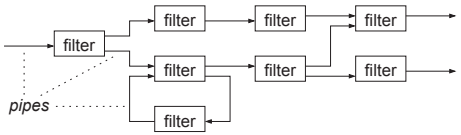
Vrstvené systémy

- Viz výše +
- používají se, pokud funkčnost může být rozdělena na část specifickou pro aplikaci a generickou část použitelnou pro mnoho aplikací
- další důvody použití
    - části systému mají být nahraditelné
    - je důležitá přenositelnost do různých OS nebo HW
    - můžeme 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



Obrázek 75 Roury a filtry

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

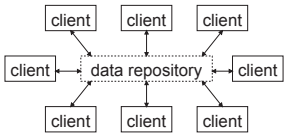
Obrázek 76 Dávkově sekvenční architektura

..

    - příklad - 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



Obrázek 77 Pasivní datové úložiště

- používá se, pokud je požadováno uchování, výběr a správa velkého množství dat a pokud jsou data vhodně strukturovaná
- hlavní cíle
    - integrovatelnost - klientské podsystémy pracují nezávisle
    - škálovatelnost - možnost snadno přidávat nové klienty a data

Aktivní datové úložiště: styl tabule (blackboard)

- množina agentů spolupracuje pomocí datového úložiště
    - úložiště posílá oznámení agentům o změně dat, která je zajímají
    - agent vyhodnotí obsah úložiště, případně vloží výsledek nebo částečné řešení
- například systémy pro umělou inteligenci - neznáme vhodné uzavřené algoritmické řešení problému, ale umíme řešit pomocí např. znalostních agentů, kteří k řešení přispívají

Distribuované systémy

- 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
- + viz výše

Obrázek 78 Peer-to-peer architektura

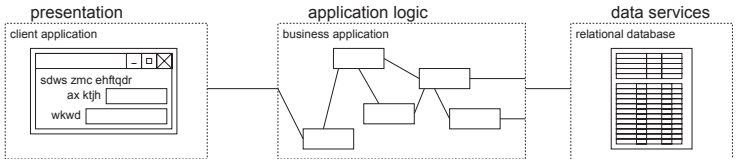
Architektura klient/server

- 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 (3layer architecture)

- funkčnost se rozděluje do tří částí (každá vrstva svá specifika, vyžaduje jiný typ programování, jiné testování apod.)
    - prezentační vrstva
        - klienti - obvykle jeden klient slouží jednomu uživateli
        - vytváří ji ve velké míře designéři
        - výhodné co největší oddělení od aplikační vrstvy, zvláště pokud aplikace poběží na víc zařízeních
    - datová vrstva
        - „technický“ kód, databázoví specialisté
        - načtení-uložení dat z/do databáze
        - zajišťuje správné fungování transakcí, odolnost proti chybám, rychlost při vysoké zátěži
        - databáze + uložené procedury + obslužný kód
    - aplikační logika
        - logika aplikace, reprezentace tříd (např. účet), obchodní pravidla (např. jak se počítá úrok)
        - obchodní pravidla se můžou často měnit – modifikace komponent

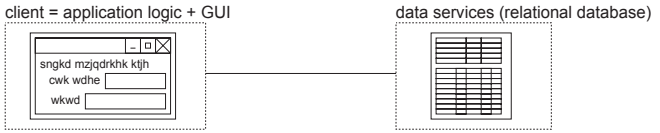
- v současné době nejoblíbenější
- častá varianta: tenký klient - klientem je např. WWW prohlížeč



Obrázek 79 Třívrstvá architektura

Tlustý klient (fat client)

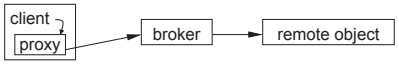
- klient obsahuje prezentaci i aplikační logiku, využívá data z databáze
- oproti třívrstvé architektuře relativně snadný návrh, ale obtížná údržba



Obrázek 80 Tlustý klient

Broker

- cílem transparentní distribuovanost - objekt volá metodu jiného objektu, aniž by věděl, že objekt není lokální



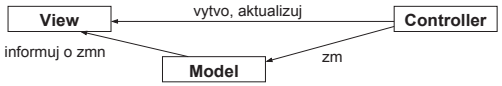
Obrázek 81 Broker

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

Interaktivní systémy

Architektura Model-View-Controller (MVC)

- architektura používaná pro klasické i webové interaktivní aplikace
- blízká pojetí třívrstvé architektury



Obrázek 82 MVC

- 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 (prezentace modelu na obrazovce)
        - při změně obsahu (stavu) modelu oznámí toto model všem závislým view, je vyvolána změna zobrazení
        - více možností prezentace modelu – více view
    - *Controller* - obsluhuje interakci uživatele s modelem
        - na základě změn modelu a vstupů uživatele (stisku kláves, výběru z menu) vybírá View, které se má zobrazit
        - reakci na uživatelský vstup je možné měnit, aniž by to ovlivnilo model nebo view (klávesová zkratka, výběr z menu)
        - typicky jeden Controller pro množinu příbuzných fcí

Ostatní styly

Virtuální stroj

- výpočet je 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

Návrh systému shora dolů

- alternativa metodičtějších přístupů
- popisováno v mnoha publikacích
- postup návrhu architektury v podstatě stejný pro objektové i strukturované metody (využití architektonických stylů)
- při postupu vycházejícím ze strukturované analýzy důraz na
    - fyzický (umístění komponent) pohled
    - strukturální (rozdělení do implementačních částí) pohled
        - rozhodneme, které části esenciálního modelu alokovány na které počítače, přiřadíme „procesy“ a „paměti“ alokované na stejný počítač jednotlivým procesům

Další pravidla pro návrh architektury

Komponenty mají být navržené tak, aby každá odpovídala jedné vývojářské specializaci

- uživatelské rozhraní – „designér“
- datová vrstva – databázový specialista
- specializované komponenty – vzdálená komunikace, importy, exporty dat, apod.

Oddělení standardního („technického“) kódu od kódu reprezentujícího aplikační doménu

- „technický“ kód - komunikace s databází, porovnávání řetězců, vyhledávací strom, knihovny standardních funkcí,…

Oddělení stabilního a často se měnícího kódu

- Často se měnící kód – samostatné komponenty
- Životnost kódu jednotlivých vrstev může být velmi rozdílná

Další poznámky:

- Na návrhu architektury se dá hodně zkazit
    - Špatná škálovatelnost
    - Zbytečně složitá struktura
    - Předvídání dalšího vývoje
    - Špatně navržená komunikační rozhraní
    - Je lepší jej svěřit zkušenému specialistovi a ještě jej např. zoponovat
- Architektura a nesplnitelné požadavky
- Experimenty v architektuře
    - ve složitých projektech rozhodně NE!, osvědčená řešení
    - v případě nové technologie zkušební prototyp (zjistí se reálnost celkové koncepce) – vyplatí se

Kvalita objektově orientovaného návrhu

- Kdy je objektově orientovaný návrh dobrý a kdy špatný, jak se to pozná?
    - Zapouzdření a spojitost
    - Různé abstrakce návrhu – jiná použitelnost, domény tříd, zatížení tříd
    - Soudržnost tříd
    - Stavový prostor, neměnné podmínky třídy, předběžné a následné podmínky operací
    - Přizpůsobení typů a uzavřené chování – robustní hierarchie tříd
    - Nebezpečí mechanismů dědičnosti a mnohotvarosti
    - Kvalita rozhraní třídy

Zapouzdření a spojitost (vzájemná závislost)

- Existují už na strukturované úrovni, v OOP další význam
- Důležité vlastnosti pro pochopitelnost a zpracovatelnost oo softwaru

Struktura a úrovně zapouzdření

- První princip zapouzdření (o úroveň výše než samotná struktura kódu) = podprogram = úroveň 1 (kód má úroveň zapouzdření rovnou nule)
- Třída (objekt) – seskupení podprogramů a vlastností = úroveň zapouzdření 2
- Úrovně zapouzdření 3 a 4 - balíčky a komponenty (viditelné jen některé třídy nebo části jejich rozhraní)
- Úroveň 3
    - horizontální seskupení tříd – podle subjektu (třídy spolu nemusejí spolupracovat) – odlišné balíčky pro odlišné subjekty
    - vertikální seskupení tříd – vzájemně spolupracující třídy – obvykle z několika domén - komponenty

Tabulka 6 Struktura a úrovně zapouzdření

Od:/K:	Úroveň 0 (řádek kódu)	Úroveň 1 (procedura)	Úroveň 2 (třída)
Úroveň 0 (řádek kódu)	Strukturované programování	Vzdušnost	-
Úroveň 1 (procedura)	soudržnost	Vázání	-
Úroveň 2 (třída)	-	Soudržnost tříd	Vázání tříd

- vzdušnost (fan out) – míra počtu odkazů na další procedury (podle počtu řádků v dané proceduře)
- soudržnost (cohesion) – míra naplnění jednoho smysluplného chování dané procedury
- vázání (coupling) – míra počtu a síly spojení mezi procedurami
- soudržnost tříd - míra jednoho smysluplného chování dané třídy (soustředěnost operací a atributů na naplňování smyslu třídy)
- vázání tříd – míra počtu a síly spojení mezi třídami

- a možnosti dalších vztahů mezi jednotlivými úrovni – hledání universa – kandidát: spojitost

Spojitost

- spojité sw prvky jsou zrozeny ze související potřeby (během analýzy, návrhu, programování)
- definice spojitosti pro sw: Spojitost mezi dvěma sw elementy A a B znamená:
    - lze postulovat nějakou změnu A, která bude vyžadovat i změnu B (nebo alespoň důkladné prověření), aby byla zachována celková správnost
    - lze postulovat změnu, která bude vyžadovat společnou změnu A i B, aby byla zachována celková správnost

Varianty spojitosti

A: int i;
B: i:=5;

- spojitost typu (změna int např. na char)
- spojitost názvu (změna i na j)

- explicitní spojitost – přímo vidíme
- implicitní spojitost

I: JUMP J+15
…
J: LR …
…
K: (sem se skáče z I)

- čím implicitnější spojitost, tím hůře se hledá.
- sw elementy spolu nemusejí komunikovat, aby byly spojité

- Spojitosti jsou
    - směrové
        - jednosměrně spojité (spojitosti názvu – např. dědění)
        - obousměrně spojité
    - nesměrové – explicitně na sebe neukazují (používají např. stejný algoritmus)

Statická spojitost – platí na úrovni tříd

- názvu - proměnné mají stejný název, aby se odkazovaly na stejnou věc
- typu (třídy) – např. proměnnou typu int naplníme celočíselnou hodnotou
- konvence – např. osobní čísla začínající A patří studentům aplikovaných věd
- algoritmu – např. používáte kódovací tabulku, do které vkládáte a z které vybíráte prvky, musíte používat stejný kódovací algoritmus
- pozice – sekvence řádků kódu, pořadí argumentů

Dynamická spojitost

- vykonávání – ekvivalent spojitosti pozice (inicializace proměnné před použitím, nastavení a testování semaforu, apod.)
- časování – systémy reálného času
- hodnoty – většinou omezení hodnot (rohy obdélníka musí i při posunu zachovat obdélník – nelze posunout jeden roh)
- identity – dva objekty odkazují na tentýž objekt

- „nespojitost“ (spojitost rozdílu) - dvě různé proměnné se nemohou jmenovat stejně, problém u vícenásobné dědičnosti

Spojitost a hranice zapouzdření

- bez existence zapouzdření – problémy
    - množství spojitosti (nespojitosti)
    - co je spojitost a co jej jen náhodná podobnost

- Spojitost – důvod funkčnosti objektové orientace

Spojitost a správa kódu

- pro lepší správu kódu
    - minimalizace spojitosti (a také nespojitosti) – tj. rozdělení celku do zapouzdřených částí
    - minimalizace spojitosti překračující hranice zapouzdření
    - maximalizace spojitosti v hranicích zapouzdření

- jinými slovy: Podobné věci patří k sobě, odlišné věci je třeba oddělit… a podobné věci jsou ty, které vykazují vzájemnou spojitost - platí nejen pro úroveň 2, ale i pro vyšší úrovně

Jak se spojitost hledá?

- explicitní lehce – stačí editor
- implicitní špatně – zkušenost, nadhled

Zneužití spojitosti

- funkce přítele (friend) v C++ - má přímo účel narušovat hranice zapouzdření
- neomezená dědičnost – třída využívá „neomezeně“ interně viditelné prvky nadřízené třídy – spojitost názvu, třídy,… -> dědičnost abstraktního chování se nesmí rovnat dědění interní implementace
- spoléhání se na obecně stejné pochopení nedokumentované implementace

Domény tříd

- Doména aplikace – třídy použitelné pro jednu aplikaci
    - třídy správy událostí – vykonávání činnosti pro vzniklou událost
    - třídy rozpoznávání událostí – detekce událostí

- Doména činnosti – třídy použitelné pro jeden obor či organizaci
    - třídy vztahů – asociace mezi objekty daného oboru – např. VlastnictvíPsa
    - třídy rolí – role objektů v daném oboru např. Pes, Klient, Pacient
    - třídy atributů – vlastností objektů v dané oblasti – např. ZůstatekNaÚčtu

- Doména architektury – třídy použitelné v rámci jedné počítačové architektury (nikoli architektury návrhu viz dříve)
    - třídy uživatelského rozhraní - např. Okno
    - třída pro práci s databází – např. Transakce
    - třídy pro komunikaci s přístroji – např. Port

- Doména základu – třídy obecně použitelné
    - sémantické třídy – např. Datum
    - strukturální třídy – např. Fronta
    - základní třídy – např. Int

- třídy vyšších domén jsou utvářeny třídami z nižších domén

Domény a znovupoužitelnost

- knihovny tříd základu – se kupují (součást standardních knihoven), největší znovupoužitelnost
- knihovny tříd architektury – se většinou kupují, ale nemusejí být kompatibilní se základem, vyžadují úpravy a rozšíření, „střední“ znovupoužitelnost
- knihovny tříd činnosti – se dají koupit, ale často vyhovují jen obecným aspektům oboru, „střední“ znovupoužitelnost
- knihovny tříd aplikace – píšete si sami, prakticky nemožné dosáhnout znovupoužitelnost

Zatížení

- jak vysoko se nachází třída nad základem
- zahrnuje další třídy, na které se třída musí spoléhat
- celkový počet přímých a nepřímých odkazů ze třídy T – měřítko složitosti třídy
- vyšší doména – vysoké nepřímé zatížení, nižší doména naopak
- neočekávané vysoké nepřímé zatížení v nízké doméně naznačuje chybný návrh třídy, nízké nepřímé zatížení ve vyšší doméně – pravděpodobně třída je tvořena přímo třídami základu

Zákon Demeter – princip omezení přímého zatížení třídy omezením velikosti jejích přímých odkazů (dvě verze – silný a slabý)

Pro objekt **obj** třídy T a pro libovolnou operaci **op** definovanou pro **obj** musí být každým cílem zprávy v implementaci **op** jeden z následujících objektů
- samotný objekt **obj**
- objekt odkazovaný argumentem v podpisu **op**
- objekt odkazovaný nějakou proměnnou **obj** (včetně objektů v kolekcích, na které se **obj** odkazuje)
- objekt vytvořený **op**
- objekt, na který se odkazuje globální proměnná

- Zákon omezuje umělé odkazy na další třídy v rámci dané třídy, omezuje spojitost mezi hranicemi zapouzdření (silný zákon) – usnadnění správy a vývoje systémů

Soudržnost třídy

- míra vzájemné souvislosti prvků (atributů a operací) umístěných v externím rozhraní třídy
- problémy pozorovatelné v rozhraní třídy

- *Smíšená soudržnost instancí* – třída má prvky, které jsou pro určité objekty třídy nedefinované – indikuje nedokonalou hierarchii tříd a nutnost zavést třídy na nižší abstraktní úrovni

- *Smíšená soudržnost domén* - třída obsahuje prvek, který přímo zatěžuje danou třídu nevlastní třídou jiné domény (třída B je nevlastní třídě A, pokud lze A plně definovat bez odkazu na B) – indikuje, že jsme třídu zatížili třídou jiné než nižší domény (často se tam plete např. doména architektury) – třídy bychom měli zatěžovat pouze třídami z nižších domén

Smíšená soudržnost rolí - třída obsahuje prvek, který přímo zatěžuje danou třídu nevlastní třídou ležící v téže doméně

- největší nebezpečí tvoří smíšená soudržnost instancí, nejmenší smíšená soudržnost rolí

Stavový prostor a chování

- = vlastnosti třídy

- stavový prostor – celek všech povolených stavů libovolného objektu třídy T
- dimenze stavového prostoru přibližně odpovídá atributům definovaným v dané třídě (nelze počítat atributy, které lze odvodit z jiných)

Stavový prostor podřízené třídy

- je-li třída B podřízenou třídou třídy A, pak stavový prostor třídy B musí být zcela obsažen ve stavovém prostoru třídy A
- je-li třída B podřízenou třídou třídy A, pak stavový prostor podřízené třídy B musí zahrnovat přinejmenším dimenze třídy A, může jich však obsahovat i více

Chování podřízené třídy

- povolené chování třídy T je množina přechodů, které může třída T vykonat mezi stavy ve stavovém prostoru T

- je-li třída B podřízenou třídou třídy A, pak chování třídy B je omezeno chováním třídy A
- je-li třída B podřízenou třídou třídy A, pak chování třídy B rozšiřuje chování třídy A

Neměnná třídy – omezení stavového prostoru

- neměnná třídy je podmínka, kterou musí v každém okamžiku naplnit každý objekt dané třídy (když je objekt v rovnováze)

Předběžné a následné podmínky

- vztahují se k operacím
- předběžná podmínka musí být pravdivá na začátku operace, pokud není, může operace odmítnout vykonání a vyvolat stav výjimky
- následná podmínka musí být pravdivá na konci vykonávání operace, v opačném případě je implementace operace nesprávná je třeba ji opravit

- podmínky společně formují kontrakt s klientem (viz scénáře užití)

Přizpůsobení typu a uzavřené chování

- vytváření robustních hierarchií tříd
- typ a třída, typ a rozhraní
- hierarchie tříd, hierarchie typů

Přizpůsobení typu

- je-li P skutečným podtypem T, pak P musí být přizpůsobeno T. (Objekt typu P může být poskytnut v libovolném kontextu, kde je očekáván objekt typu T)

- hierarchie třída/podtřída musí odpovídat principu přizpůsobení typu
- jak zajistit přizpůsobení typu, tedy polymorfismus?

Principy kontravariance a kovariance

1. Neměnná podřízené třídy je přinejmenším tak silná jako neměnná nadřízené třídy
2. Každá operace nadřízené třídy má odpovídající operaci v podřízené třídě se stejným názvem a podpisem
3. Předběžná podmínka každé operace není silnější než odpovídající předběžná podmínka operace v nadřízené třídě – princip kontravariance
4. Následná podmínka každé operace je přinejmenším tak silná jako odpovídající podmínka operace v nadřízené třídě – princip kovariance

Souhrn požadavků na přizpůsobení typu

- má-li být podřízená třída P podřízeným typem třídy T, pak musí být splněna následující omezení
  1. Stavový prostor P musí vůči stavovému prostoru T splňovat podmínky uvedené výše

  Pro každou operaci třídy T (T.op), kterou P přepisuje operací P.op, platí
  2. P.op musí mít stejný název jako T.op
  3. Seznam argumentů formálního podpisu operace P.op musí odpovídat seznamu argumentů formálního podpisu operace T.op
  4. Předběžná podmínka P.op musí být rovna nebo slabší než předběžná podmínka T.op
  5. Následná podmínka P.op musí být rovna nebo silnější než následná podnínka T.op

Princip uzavřeného chování

- přizpůsobení typu vede k dobrému návrhu v operacích přístupu, samotné však nestačí
- operace změny – princip uzavřeného chování – chování zděděné podřízenou třídou z nadřízené třídy musí respektovat neměnnou dané podřízené třídy
- k uzavření podřízené třídy do chování třídy nadřízené nedochází automaticky, musíme si navrhnout sami
- návrhář má povinnost zajistit uzavřenost chování své třídy

- 90% situací v objektovém návrhu je přímočarých, 10% problematických je popsáno výše

Návrhové vzory (design patterns)

- při řešení problémů se často dá vycházet z ověřeného způsobu řešení
- řeší, **jak** se problém řeší v objektově orientovaném prostředí -> UML modely návrhu

- **GoF patterns** - 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.)

- **J2EE patterns -**
  http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html

- **.NET patterns (většinou GoF patterns pro C#)**
  http://www.dofactory.com/

Další literatura

- *B. Eckel: Thinking in Patterns.* Výklad o návrhových vzorech založený na jazyce Java. Předběžnou verzi lze získat ve formátu PDF na adrese http://www.mindview.net/Books
- *J. W. Cooper: Java Design Patterns.* Addison-Wesley 2000. ISBN 0-201-48539-7. 330 stran. Návrhové vzory založené na jazyce Java.
- *J. W. Cooper: The design patterns JAVA companion* (el.verze zdarma)
- *J. W. Cooper: Design patterns and Object oriented programming in Visual Basic 6 and VB.NET* (el. verze zdarma)
- *I. Kraval: Design Patterns v OOP se zaměřením na Javu, C# a Delphi*, http://www.objects.cz (el. verze po žádosti zdarma)

Návrhový vzor – obecně

- Popisuje zobecnění konkrétních situací (přirovnání k matematickému vzorci) -> problém s úrovní abstrakce…a pochopením…vzor se vysvětluje na konkrétních použitích
- proměnné vzorce - účastníci vzoru
- motiv – známé použití vzoru -> srovnání motivu a problému naší aplikace -> schopnost vzor použít
- smysl – identifikace vzoru – velmi stručné vystižení problému a řešení
- kombinace vzorů

Katalog vzorů

- Vyhledávání podle smyslu vzoru
- Vyhledávání podle kritérií – klasifikace vzoru:
  - Podle účelu (purpose)
  - Rozsahu platnosti (scope)
  - Důležitější rozdělení dle účelu

Klasifikace vzorů podle účelu

- Pro tvorbu objektů – tvořivé vzory - creational patterns – řeší tvorbu objektů s tím související problematiku (např. odstínění tříd)
- Pro struktury – strukturální vzory - structural patterns – řeší opakující se struktury objektů
- Řešící chování – vzory chování - behavioral patterns

Klasifikace vzorů podle rozsahu platnosti

- S rozsahem platnosti tříd – class patterns – popisují interakce mezi třídami (dědění, agregace,…)
- S rozsahem platnosti objektů – object patterns – popisují vztahy mezi objekty, většinou problémy typu změna „něčeho v runtime“, flexibilnější než třídní vzory

Skladba vzoru

- Název a klasifikace
- Alias – jiné názvy pro vzor
- Smysl – většinou jedna věta
- Motiv – demonstrace problému
- Aplikovatelnost – podmínky vedoucí k použití vzoru
- Struktura vzoru – popisuje vzor jako takový
- Účastníci vzoru – role vzoru s účastníky vzoru, popis jejich postavení a účasti ve vzoru
- Spolupráce – interakce mezi účastníky vzoru, tj. zasílání zpráv, scénáře,…
- Důsledky – výhody a nevýhody použití vzoru
- Implementace – „historicky“ Smalltalk, C++, dnes Java, .NET, Delphi
- Příklad vzoru – často spojeno s částí implementace
- Známé použití – v ČR není známo
- Související vzory – řešení často kombinace vzorů, zástupné vzory

Myšlenkový základ vzorů

- Dva pohledy: klient-rozhraní a rozhraní-implementace, pro návrhový vzor důležitý první pohled -> změnu implementace klient nepozná -> nebudeme se zaměřovat na implementaci
- Polymorfismus
- Posílání zpráv a vyvolávání metod
- Terminologie – zpráva, metoda, operace
- Abstraktní třída
- Abstraktní operace
- Čistá abstraktní třída
- Rozhraní

Přehled vzorů

Iterátor

- jeden z nejjednodušších a nejčastěji používaných návrhových vzorů
- smysl: jednoduchý a sekvenční přístup ke složité struktuře objektů

- motiv: agregovaný objekt (jako je např. seznam) by měl umožňovat procházení, aniž by k tomu uživatel musel znát vnitřní strukturu agregátu (která se může změnit) – uživatel = zjednodušený pohled na složitou strukturu a její snadné ovládání

- iterátor udržuje informaci o aktuálním objektu v agregátu, umí se posunout na následující prvek agregátu a prvek poskytnout uživateli (anebo také vrátit první prvek seznamu, poslední prvek seznamu apod.)

- moderní OO programovací jazyky obsahují podporu iterátorů
- př. Java - 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.

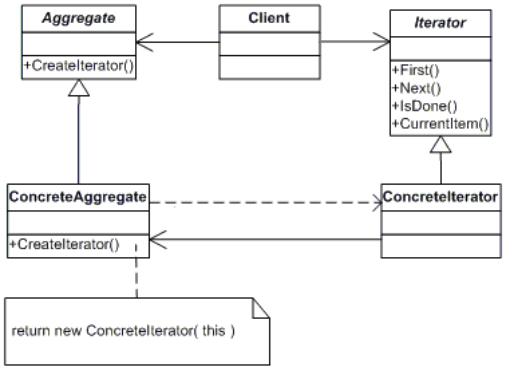
```
for (Iterator ii = c.iterator(); ii.hasNext();) {

    System.out.println(ii.next());

}
```

- jednoduché nabízení vlastních složitých struktur = předřazení objektu s rozhraním vzoru Iterator (možné použít rozhraní Iterator daného prostředí)
- tvorba iterátoru – nejlépe od dané struktury… např. operace CreateIterator()

Struktura vzoru



Obrázek 83 Návrhový vzor Iterátor

Použití vzoru

- skrytí složité struktury složeného objektu
- polymorfní průchod různými typy struktur
- násobný přístup ke složené struktuře a další

Související vzory

- vzor Composite se často doplňuje o vzor Iterator
- tvorba Iteratoru = vzor Factory Metod (Prototype)
- robustní Iterator – vzor Observer

Reification (Zkonkrétnění)

- obecný termín s významem "zkonkrétnit něco",
- v objektovém návrhu znamená "udělat objekt z něčeho, co na první pohled jako objekt nevypadá"
- není návrhový vzor, ale spíše meta-vzor, ze kterého vznikají konkrétní návrhové vzory

- když o nějaké aktivitě nebo chování začínáme uvažovat jako o metodě, ale časem zjistíme, že má spoustu komplikovaných variant - pak je aktivita vhodným kandidátem na třídu, varianty chování jsou kandidáty na podtřídy

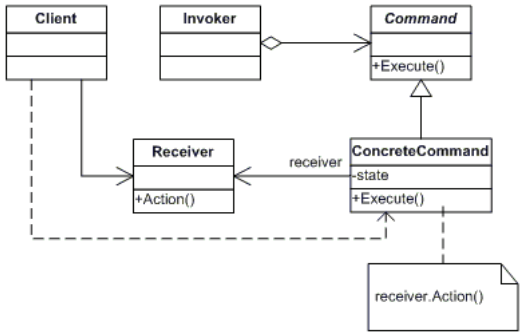
- např. vyhledávání: na první pohled vypadá jako úloha pro metodu, časem zjistíme, že vyhledávání lze specifikovat mnoha způsoby, vytvoříme třídu Search, jejíž instanční proměnné udržují informace o kritériích vyhledávání
- objekt třídy Search se nám hodí, pokud budeme chtít vyhledávat podle trochu odlišných kritérií
- pokud bychom chtěli prohledávat různá data, můžeme vytvořit několik podtříd třídy Search
- nejdůležitější metoda se bude jmenovat např. "execute" nebo "doIt" - bude reimplementována polymorficky v podtřídách

- třída může vzniknout i z něčeho většího, než je metoda, např. Jacobson říká, že vhodným kandidátem na objekt je celý případ užití
- pokud provedeme reifikaci aktivity, říkáme vzniklému objektu "řídící objekt" (control object)

Příkaz (Command)

- smysl: zapouzdřuje požadavek do podoby objektu -> požadavek je proměnná (seznamy požadavků, parametrizace požadavků apod.)
- motiv
  - dynamické skládání prvků GUI
  - flexibilní zpracování návratové hodnoty
  - a další
- klient volá operaci, ta vrací hodnotu, máme se rozhodnout, jak na ni budeme reagovat (řešení – switch - funkční, ne flexibilní), vytvoříme abstraktní třídu "Command" s operací "execute"
- pro každé zpracování návratové hodnoty vytvoříme konkrétní podtřídu, která operaci implementuje, výběr převedeme na výběr instancí z těchto tříd
- klient je odstíněn od konkrétního volání operace objektů reagujících na návratovou hodnotu

Struktura vzoru



Obrázek 84 Návrhový vzor Command

Použití vzoru

- flexibilní přidávání podtříd Command, s objekty se pracuje jako s proměnnými
- oddělení klienta od realizace požadavku, klientovi můžeme vyměnit dynamicky „jeho command“
- Cilent – vytváří instance jednotlivých Commandů a poskytuje informace o objektu Receiver
- Invoker – definuje, kdy má být metoda execute() zavolána
- Composite + Command = rekurzivní MacroCommand
- v procedurálním programování – volání funkce přes pointer
- samotný Command nemusí jenom delegovat, ale může i sám něco vykonávat
- kopie vykonaných objektů Command v seznamu + operace Undo() – historie

Související vzory

- Composite – MacroCommand
- Memento – Undo()
- Prototyp – seznam historie

Memento

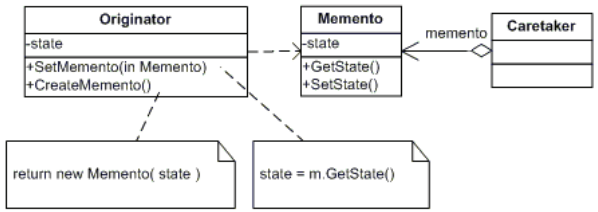
- smysl: klient získá stav objektu, aby bylo možné vrátit objekt do původního stavu, není porušeno zapouzdření

- motiv: zavoláme operaci objektu, změní se jeho stav a my chceme vrátit stav do stavu původního, nelze se spoléhat na inverzní operace, budeme si pamatovat výchozí stav

- klient používá objekt třídy Originator, potřebuje uchovat současný stav, zavolá operaci createMemento(), vytvoří se objekt Memento, překopíruje se do něho stav objektu třídy Originator, klient si uchová objekt Memento, při návratu klient zavolá operaci setMemento(Memento memento) a předá mu objekt memento, který obdržel při předchozím volání operace createMemento()

- klient ví, kdy má memento použít (vytvořit, vrátit zpět, odstranit), ale neví, co v něm je
- objekt třídy Originator ví, jak se memento tvoří a jak se podle něj rekonstruuje stav, ale neví, kdo a kdy bude o memento žádat, nabízí pouze vytvoření a vydání objektu mementa

Struktura vzoru



Obrázek 85 Návrhový vzor Memento

Použití vzoru

- rozdělování rolí, zjednodušení problému
- kombinace vzorů Command a Memento – Command podporuje operaci Undo(), pro kterou se používá vzor Memento, existuje seznam vykonaných Commandů, mělké kopie (každá kopie stejné instance Commandu vlastní memento, ale referenci na objekt)

Související vzory

- Command viz výše
- Iterator (průchod historií Commandů a Mement)

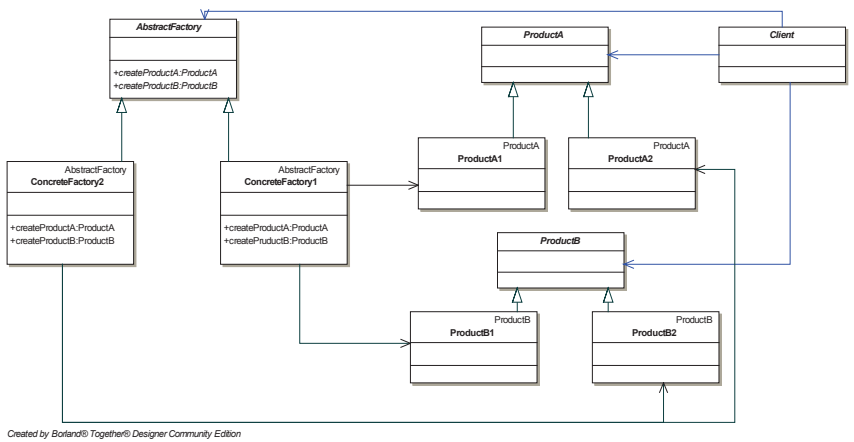
Abstraktní továrna (Abstract factory)

- klasifikace: Tvořivý, Objekt
- smysl: Zavedení rozhraní pro tvorbu rodin (řad) příbuzných nebo závislých objektů, aniž by se musely specifikovat konkrétní třídy, klient je izolován od volání těchto tříd
- alias: Kit (Souprava)

- motiv:
  - dva (více) standardů vzhledu grafického uživatelského rozhraní, označme St1 a St2,… lze zavést buď St1, nebo St2, nebo…
  - jiný vzhled definuje odlišná zobrazení a chování prvků GUI - oken, posuvníků, atd.
  - aplikace může přepínat mezi standardy
  - definována abstraktní třída FactorySt – rozhraní pro tvorbu prvků GUI, konkrétní třídy FactorySt1 a FactorySt2
  - definování abstraktní třídy pro prvky GUI - pro okna, posuvníky,…tj. např. pro okna WindowSt1 a WindowSt2 abstraktní třída WindowSt, pro posuvníky ScrollBarSt1, ScrollBarSt2, abstraktní třída ScrollBarSt
  - FactorySt – abstraktní operace pro vytvoření prvku GUI např. CreateWindow() a CreateScrollBar()

  - FactorySt1 a FactorySt2 - konkrétní implementace rozhraní FactorySt, vracejí odpovídající typy prvku GUI, tj. např. ScrollbarSt1 nebo ScrollBarSt2
  - Volba prvků GUI je převedena na volbu objektu továrny FactorySt1, nebo FactorySt2,…
  - klient vytváří prvky GUI pomocí rozhraní FactorySt, zavazuje se pouze k rozhraní definovanému abstraktní třídou, nevolá konstruktory objektů, pouze metody rozhraní
  - objekty FactorySt, tj. např. objekt FactorySt1 a objekt FactorySt2 umístíme do kolekce, nazveme např. ManagerFactory, tyto objekty se přidávají spolu s klíčem

Struktura vzoru



Obrázek 86 Návrhový vzor Abstraktní továrna

- využití rolí – nahrazení prvků GUI pojmem Product a továrny do stromu s předkem AbstractFactory a rolemi potomků ConcreteFactory
- AbstractFactory vytváří rozhraní, klient volá metody rozhraní, žádné konstruktory

Použití vzoru

- odstínění volání konstruktoru, tj. je zabezpečena flexibilita výměny produktů
- je možné vyměnit „řadu produktů“ změnou výběru továrny
- produkty typu framework, knihovny – klient nevytváří přímo objekty, ale volá metody rozhraní různých továren
- izoluje klienta od implementace konkrétních tříd, zapouzdřuje odpovědnost a proces tvorby produktových objektů
- usnadňuje výměnu produktových řad a zabezpečuje používání produktů jedné řady

Nevýhody vzoru

- přidání produktu znamená rozšíření rozhraní AbstractFactory a implementaci tohoto rozhraní v konkrétních továrnách
    - i při malém rozdílu skupiny produktů je nutné přidat novou konkrétní továrnu – řeší vzor Prototype

tzv. extenzivní flexibilní továrna

- rozhraní = 1 metoda se vstupním parametrem identifikujícím produkt – klíč produktu
- návratová hodnota metody rozhraní abstraktní továrny musí být objekt implementující stejné rozhraní – všechny produkty musí patřit do jedné rodiny s jedním předkem

Továrna (klon abstraktní továrny)

- nevyžaduje se přepínat rodiny (existuje jen jedna) + vzor se vstupním parametrem = návrhový vzor Továrna (Factory)
- slouží k odstínění konstruktoru, volá se metoda rozhraní s parametrem ID produktu, návratová hodnota typu vrcholu stromu dědičnosti
- pomocný objekt Factory ke stromu produktů (v tomto případě strom generalizace-specializace)
- převede volání konstruktoru na volání metody rozhraní s parametrem – flexibilní
- typické použití dynamické menu – strom položek (s kořenem CPolozka), číselník položek (id, název položky), rozhraní továrna (MojeTovarna) s jedinou metodou s parametrem(vytvor(int id)), pak vždy stejný kód např.

CPolozka MojePolozka = MojeTovarna.vytvor(id);

Související vzory

- abstraktní továrna realizována vzorem tovární metoda (Factory method)
- konkurenční vzor prototype
- většinou se zavádí jako Jedináček (Singleton)

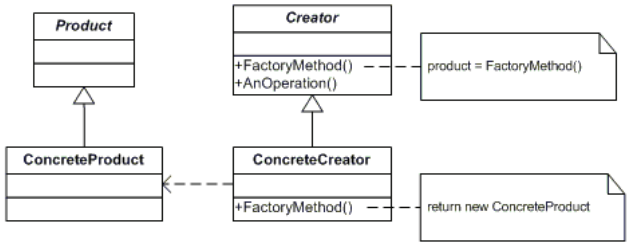
Tovární metoda (factory method)

- klasifikace: Tvořivý, Třída
- smysl: Definuje metodu rozhraní pro zrod objektu na úrovni předka, ale ponechává potomkovi k rozhodnutí, jakého konkrétního typu tento objekt bude
- alias: Virtuální konstruktor

- motiv: použití ve vzoru Abstraktní továrna, rozhraní obsahuje metody CreateProduct(), návratové typy metod jsou vrcholy stromu daných produktů, každá konkrétní třída abstraktní továrny dosazuje do návratové hodnoty konkrétní typ dědice produktu

- jiný motiv:
    - vývoj knihovny pro tvorbu editorů, několik možných editorů podle typů dokumentů
    - daný typ aplikace pracuje s daným typem dokumentu, abstraktní třídy Application a Document, podtřída třídy Document je specifická dané oblasti
    - třída Application nemůže předvídat konkrétní třídu dokumentu, pouze ví, kdy by měl být dokument vytvořen, ale neví, jaký druh dokumentu vytvořit
    - řešení: abstraktní metoda CreateDocument() předefinovaná konkrétními aplikačními podtřídami, které vytvářejí konkrétní dokumenty

Struktura vzoru



Obrázek 87 Návrhový vzor Tovární metoda

Struktura

- Product – definice rozhraní objektů vytvářených tovární metodou
- ConcreteProduct – implementuje rozhraní produktu
- Creator – deklaruje tovární metodu, může definovat výchozí implementaci
- ConcreateCreator – překrývá tovární metodu, aby vrátila instanci ConcreteProduct

Použití vzoru

- scénář typu předek ví, kdy je třeba nechat vzniknout objekt, ale neví, jaký objekt, potomek ví a rozhodne, který objekt se má vytvořit
- propojení paralelních třídních hierarchií – dva paralelní stromy propojené vazbou viz motiv Aplikace a Dokument

Pozn.

- třída Creator abstraktní? – může a nemusí, jedna z aplikací se svým typem dokumentu může být implicitní

Související vzory

– abstraktní továrna, šablona (template method), prototyp

Další příklad

- metoda openWindow() ve třídách Calculator a PhoneBook má téměř stejnou podobu:

```
x = new nějakéOkno(); // typ okna závisí na třídě
x.openWidget();
```

```
abstract class DesktopObject {

        abstract Window getWindowInstance();

        void openWindow() {
                Window w = this.getWindowInstance();
                w.openWidget();
        }
}
```

```
class Calculator extends DesktopObject {
    Window getWindowInstance() {
            .... // vrací instanci správné třídy
    }
}
```

- getWindowInstance() je tovární metoda, protože (většinou) vyrábí instanci související třídy - nemusí ale vytvářet instanci vždy - někdy může jenom vrátit odkaz na existující instanci

Singleton (Jedináček)

- klasifikace: tvořivý, objekt
- smysl: třída má jen jednu jedinou instanci globálně viditelnou
- motiv:
    - často chceme, aby nějaký objekt byl globálně viditelný a jedinečný (objekt faktory u vzoru Abstract factory)
    - instance se umístí jako privátní statický člen, zpřístupňuje se přes statickou veřejnou metodu

Použití

- unikátnost instance
- nutnost skrytého konstruktoru

- řešení v jazyku Java:

```
public class Singleton {
  private static final Singleton instance = new Singleton();

  private Singleton() {} /* soukromý konstruktor */

  public static Singleton getInstance() /* pro získání instance */
  {
    return instance;
  }
}
```

- klienti získají instanci pomocí statické tovární metody x = Singleton.getInstance();

Prototyp (Prototype)

- klasifikace: Tvořivý, objekt
- smysl: Zavádí se specifické druhy objektů vznikající klonováním prototypových instancí pomocí jednotného rozhraní

- motiv: otázka vhodné parametrizace práce se třídami – výběr třídy produktu parametrem
    - částečné řešení vzor Továrna se vstupním parametrem (metoda CreateProduct() s parametrem) – nutnost přepisovat metodu i při malých změnách, seznam produktů nelze měnit dynamicky
    - výběr obrazce v aplikaci obsluhou
    - flexibilní řešení prototypem: každý obrazec ze stromu dědičnosti implementuje metodu rozhraní Clone() – po zavolání vytvoření kopie objektu
    - instance (prototyp) je spolu s klíčem umístěna do kolekce, kolekci prototypů nazveme manažer prototypů
    - potřebujeme nový objekt obrazce, nevoláme konstruktor, ale žádáme kolekci přes klíč o novou instanci obrazce voláním metody Clone(), kopie instance je návratovou hodnotou
    - v rámci inicializace metoda pro registraci daného prototypu v kolekci
    - klient může dynamicky upravovat seznam prototypů