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 atd.

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

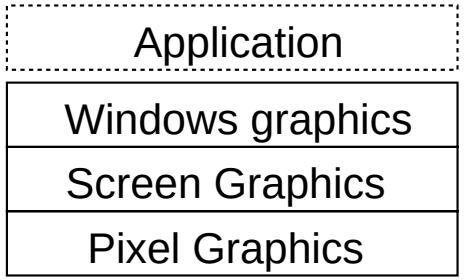
Rozdělení do vrstev

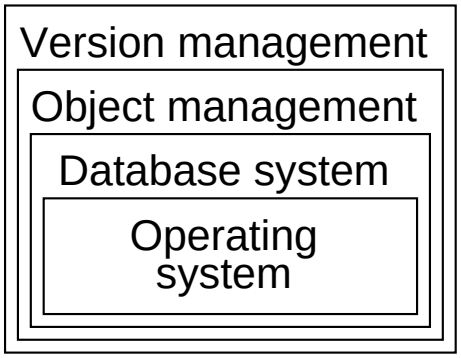
- vrstvené systémy = uspořádaná množina virtuálních světů
- každý svět je vystavěn z prvků nižšího světa a poskytuje stavební prvky vyššímu světu
- mezi vrstvami vztah klient-poskytovatel - nižší vrstvy (poskytovatelé) poskytují služby vyšším vrstvám (klientům)
- znalost je jednosměrná, tj. podsystém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

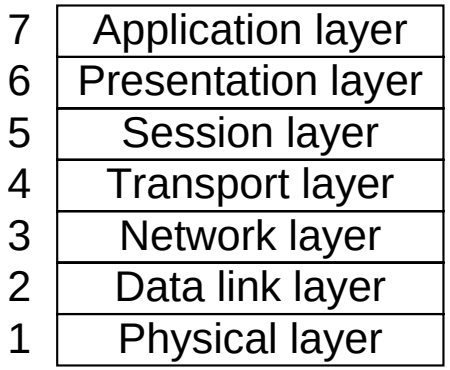
- objekty ve stejné vrstvě mohou být nezávislé, ale mezi objekty v různých vrstvách obvykle existuje korespondence (např. podsystémy poskytují obdobné služby na různých úrovních abstrakce)

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







Dva typy vrstvených architektur:

- *uzavřené (striktně vrstvené)* - vrstva je implementována pouze pomocí prostředků nejbližší nižší vrstvy
  - omezuje závislosti mezi vrstvami = princip skrývání informací
  - dovoluje snadnější změny rozhraní - změna ovlivní jen nejbližší vyšší vrstvu
  - například síťové modely (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 svrchní vrstvu (= požadovaný systém)
- spodní vrstva je dána dostupnými zdroji (HW, OS, knihovny)
- pokud je velký rozdíl, je při návrhu zapotřebí přidat mezilehlé vrstvy pro přemostění případné konceptuální mezery
- malé systémy cca 3 vrstvy, pro velké systémy obvykle postačuje 5-7 vrstev (i pro nejsložitější systémy je podezřelé více než 10 vrstev)

Poznámka (doporučení z RUP)

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

Printer driver	CD ROM driver	Network driver

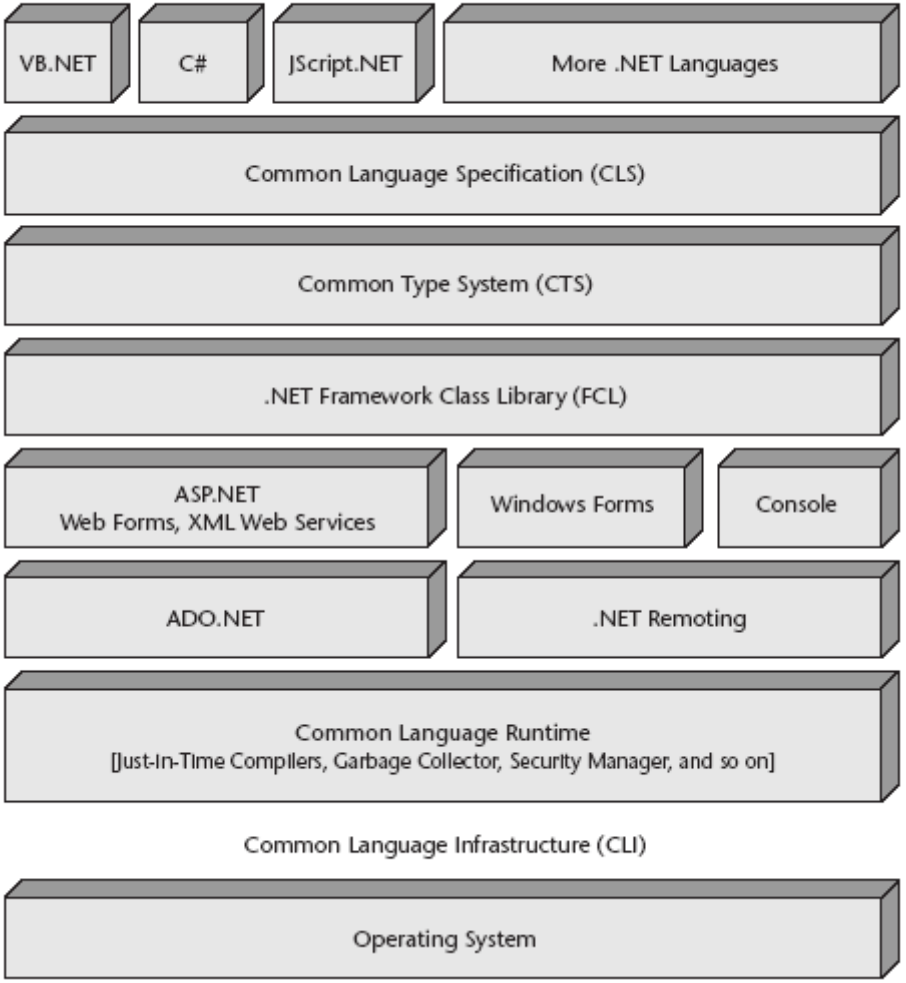
(operační systém obsahuje ovladače pro jednotlivé typy zařízení)

- podsystémy mohou o sobě navzájem něco vědět, ale protože tato znalost není velká, nevznikají podstatné závislosti mezi oddíly (např. v operačním systému podsystém pro plánování procesů a správu paměti, plánování procesů někdy potřebuje vědět, kolik je dostupné paměti apod.)

-> systém může být postupně dekomponován do podsystémů pomocí vrstev a oddílů (vrstvy můžeme dělit na oddíly a naopak oddíly do vrstev)

- ve většině velkých systémů směs vrstev a oddílů

Figure 1: An overview of the .NET architecture



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 podsystémům, 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, případně implementujeme HW

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)

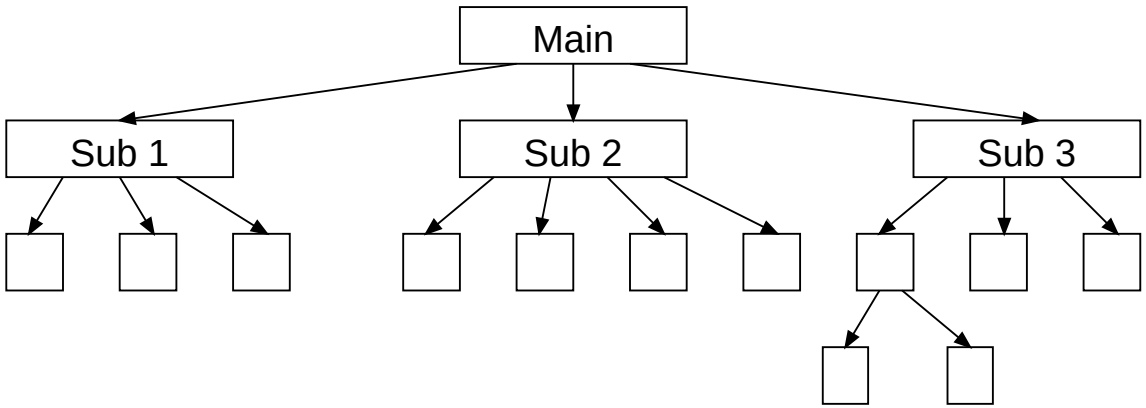
Výběr mechanismu řízení

- 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 vstup)



Systémy řízené událostmi

- běh systému řídí dispečer poskytovaný podsystémem, programovacím jazykem nebo OS
- s jednotlivými událostmi jsou svázány procedury aplikace
- systém někdy poskytuje frontu událostí, nově příchozí události se řadí do fronty, ze které dispečer vybere následující událost a zavolá odpovídající proceduru ("callback")
- procedura po skončení obsluhy události vrací řízení dispečeru
- 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). Většinou to bývá vedoucí týmu.

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

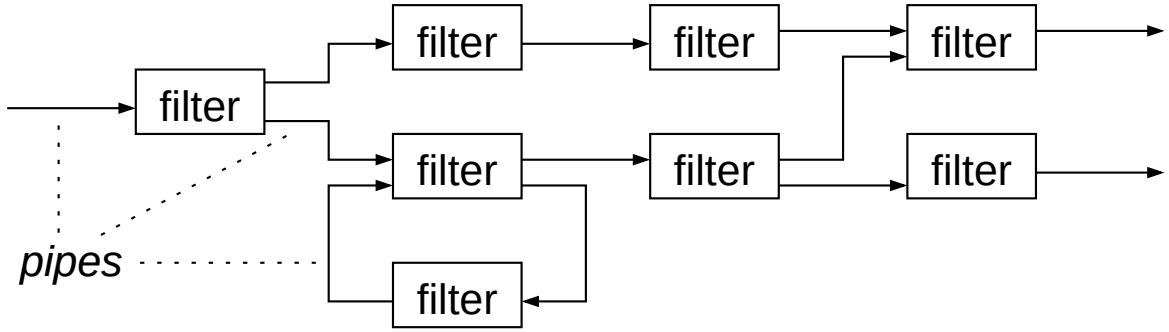
- další důvody použití :
  - části systému mají být nahraditelné
  - je důležitá přenositelnost do různých OS nebo HW
  - můžeme využít již existující vrstvu (OS, komunikaci po síti...)

- používá se v mnoha SW produktech, například některé operační systémy

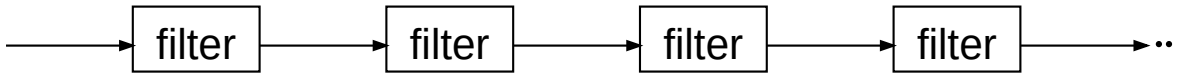
Styl dataflow

- 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



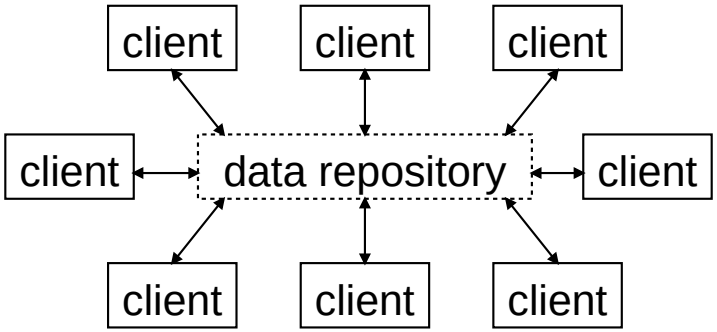
- *dávkově sekvenční architektura*
  - pokud výše uvedená architektura degeneruje do jedné lineární posloupnosti transformací
  - vstupem dávka dat, aplikuje na ní posloupnost sekvenčních komponent (filtrů)



  - 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



- 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

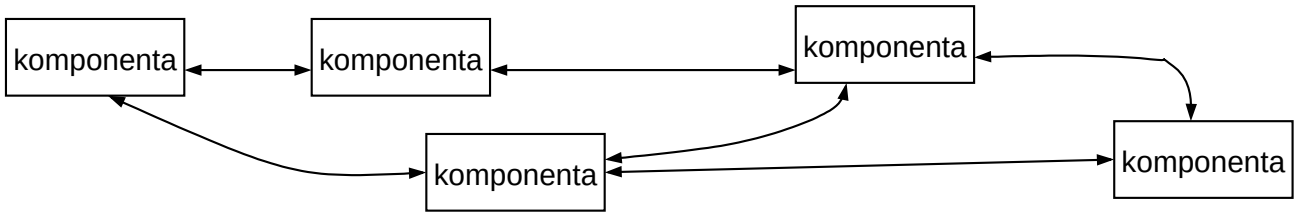
- například systémy pro umělou inteligenci - neznáme vhodné uzavřené algoritmické) řešení problému, ale umíme řešit pomocí např. znalostních agentů, kteří k řešení přispívají

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



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

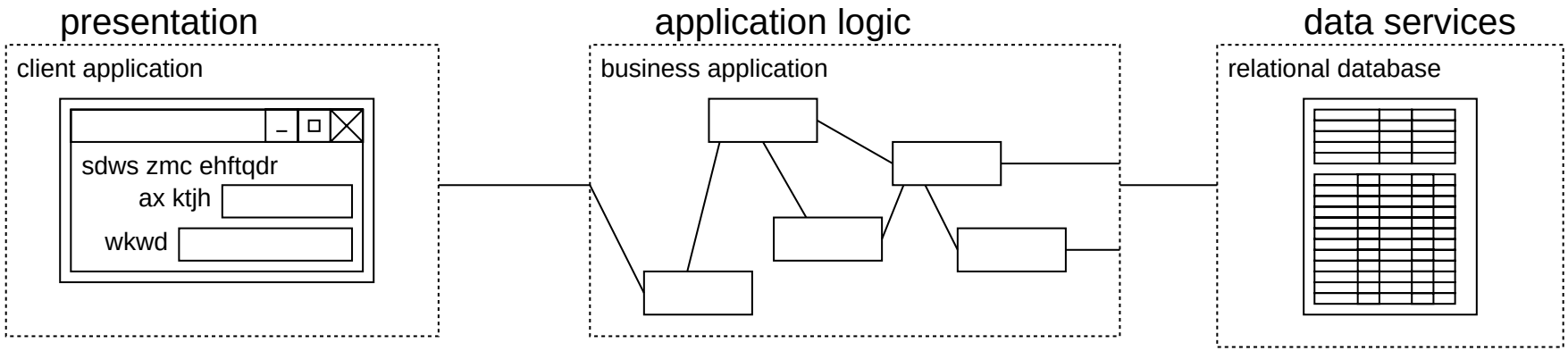
- nejčastější podstyly stylu klient-server:
  - třívrstvá architektura (3-tier architecture)
  - tlustý klient (fat client)

Třívrstvá architektura (3-tier architecture)

- funkčnost se rozděluje do 3 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

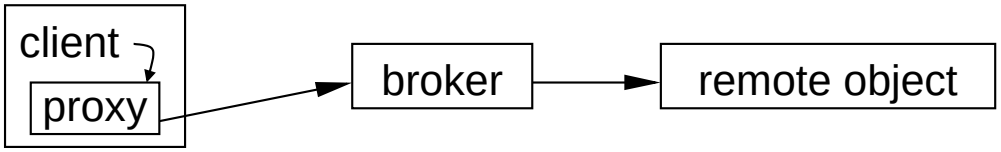


Tlustý klient (fat client)

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

Broker

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

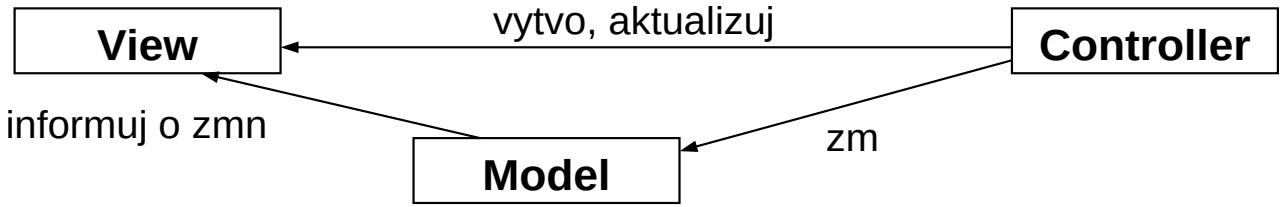


- proxy volá brokera, který zjistí, kde se nachází vzdálený objekt
- příklad: CORBA (Common Object Request Broker Architecture) - architektura+infrastruktura pro spolupráci aplikací prostřednictvím sítí

Interaktivní systémy

Architektura Model-View-Controller (MVC)

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



- odděluje funkční vrstvu aplikace ("Model") od dvou aspektů uživatelského rozhraní (nazývaných "View" a "Controller")

- *Model* - objekty, které budeme v aplikaci prohlížet a měnit (např. obchodní data)

- *View (náhled)*

  - zobrazení modelu (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, který se má zobrazit
  - reakci na uživatelský vstup je možné měnit, aniž byt 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 jedné vrstvy 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