Logický program - struktura, základní pojmy, datová struktura seznam, práce s databází Prologu. Hlavní odlišnosti oproti procedurálnímu programování, možnosti použití neprocedurálního programovacího jazyka.

Prolog je jazyk pro programování symbolických výpočtů. Jazyk vychází z principů matematické logiky (PROgramování v LOGice). Jeho úspěch byl podnětem pro vznik nové disciplíny matematické informatiky – logického programování, což je perspektivní styl programování na vyšší abstraktní úrovni. Prolog je také strojovým jazykem nejmodernějších počítačů. Má doposud specifické oblasti použití - umělou inteligenci, znalostní inženýrství, atp.

Historie: První verze jazyka sloužila pro efektivní odvozování logických důsledků z určité třídy formulí predikátové logiky. V dalších verzích byl Prolog zvolen jako základní jazyk logického procesoru centrální jednotky počítače 5. generace. To vyvolalo vlnu zájmu o Prolog a metody logického programování. Prolog se začal rychle prosazovat vedle jiných jazyků pro symbolické výpočty (např. Lisp).

K čemu se používá Prolog: Od počátku byl Prolog využíván při zpracování přirozeného jazyka (francouštiny) a pro symbolické výpočty v různých oblastech umělé inteligence. Používá se v databázových a expertních systémech i v klasických úlohách symbolických výpočtů, jako je návrh a konstrukce překladačů, a to nejen jako prostředek vhodný pro reprezentaci a zpracování znalostí, ale i jako nástroj pro řešení úloh.

Porovnání s jinými jazyky: Prolog je konverzační jazyk nové generace programovacích jazyků, která by měla usnadnit tvorbu programů co nejširšímu okruhu uživatelů. Snaží se odstranit část rutiny při programování a tím si zajistit místo vedle osvědčených jazyků jako Fortran, Cobol a Pascal.

Uživatel Prologu se může více soustředit na popis vlastností relací, tedy na otázku co se má vypočítat, aniž by byl na každém kroku nucen řešit detaily spojené s otázkou jak to udělat a kam uložit získané výsledky. V Prologu nejsou příkazy pro řízení běhu výpočtu ani příkazy pro řízení toku dat. Chybějí prostředky pro programování cyklů, větvení apod. a nepoužívá se přiřazovací příkaz. S tím souvisí i rozdílná úloha proměnných. Proměnná v Prologu označuje po dobu výpočtu objekt, který vyhovuje určitým podmínkám. Jeho vymezení se při výpočtu upřesňuje. Průběh výpočtu je v Prologu řízen jeho interpretem na základě znění programu. Programátor může chod výpočtu ovlivnit řídícími příkazy v mnohem menší míře než u jiných jazyků. I když byl původně Prolog navržen jako jazyk specializovaný na symbolické výpočty, moderní implementace směřují k obecnějšímu použití.

Logickým programem budeme rozumět posloupnost faktů (nepodmíněných příkazů) (H.) a pravidel (podmíněných příkazů) (`H :- A1, A2, ..., An.`)

Datové typy

Konstanty, **proměnné** a **struktury** souhrnně označujeme jako **termy**. Každá implementace jazyka Prolog obsahuje řadu tzv. **vestavěných predikátů** (procedury nabízené tou kterou implementací jazyka Prolog), které jsou ze syntaktického hlediska také považovány za termy. Konstanta nebo proměnná se nazývá **atomický term**. Atomickými termy jsou tedy jak jednoduchá jména objektů, tak i čísla a proměnné. Z atomických termů můžeme vytvářet i složitější výrazy, které se nazývají struktury.

Proměnné začínají velkým písmenem nebo speciálním symbolem "_" (podtržítko) a jsou tvořeny libovolnou posloupností písmen, číslic a znaku "_". Proměnnou je také samotný znak "_". Takovou proměnnou nazýváme **anonymní proměnná**, která je obdobou zájmen cokoli nebo kdokoli. Její zvláštností je to, že každý její výskyt představuje jinou proměnnou, tedy výraz `p(_,_)` je totéž co `p(X,Y)`, nikoli `p(X,X)`. Hodnoty anonymních proměnných se v dialogu uživateli nevypisují.

Klauzule: Programování v Prologu spočívá v deklarování faktů o objektech a relacích mezi objekty, v definování pravidel o objektech a relacích platících mezi nimi a v zodpovídání dotazů. Klauzule jsou výrazy jazyka, pomocí nichž se zapisují nepodmíněné příkazy – **fakta**, podmíněné příkazy, které vyjadřují **pravidla** a výrazy, které mají tvar dotazu – **cílové klauzule**.

Fakta – nepodmíněné příkazy: Nepodmíněné příkazy vyjadřují fakt o relaci. Fakta jsou vztahy, které mají jméno, za nímž následuje výčet objektů, kterých se vztah týká. Objekty jsou odděleny čárkami a jejich výčet je uzavřen do kulatých závorek. Každý fakt je ukončen tečkou. Pokud fakt neobsahuje proměnné, hovoříme o tzv. **základním faktu**.

Př. - fakta uložena v databázi, která vyjadřují, že osoba M je matkou X a Y

```
matka(M, X).
matka(M, Y).
```

Podmíněný příkaz je oddělen znaménkem implikace "`:-`" na dvě části, ukončený tečkou. Část vlevo od dvojznaku se nazývá **hlava pravidla**, napravo od něho je **tělo pravidla**.

Př.: `sourozenci(X,Y) :- matka(M,X), matka(M,Y).`

Program může obsahovat dotazy, kterým říkáme **cílové klauzule**. Cílové klauzule v textu programu nesmějí obvykle obsahovat proměnné, na něž není vázaná hodnota.

Otázka (cílová klauzule) začíná dvojicí znaků "`?-`" a končí tečkou. Na cílovou klauzuli položenou v průběhu dialogu uživatele se systémem nejsou kladena žádná omezení týkající se proměnných v ní obsažených. Dotaz, který neobsahuje proměnné se nazývá základní otázka.

Odpověď (řešení), kterou systém vrací, je buď "**yes**", "**no**" nebo výpis hodnot proměnných obsažených v otázce (kromě hodnot anonymních proměnných -> ty se v dialogu uživateli nevypisují). Jelikož databáze v Prologu nemůže obsahovat negativní data, chápeme odpověď "**no**" dvěma způsoby: 1) záporná odpověď nebo 2) neexistující odpověď.

Př. - dotaz na hledání společné matky

```
?- matka(M,X), matka(M,Y).
```

Predikáty: To co jsme dříve nazývali procedurou můžeme přesněji popsat jako všechny klauzule začínající **stejným funktorem** a mající **shodný počet argumentů**. Mohou mít stejné jméno struktury a různý počet argumentů. Proto se často uvádí jméno procedury spolu s počtem jejích argumentů (tj. **Aritou**). Procedury (**predikáty**) existují **standardní** (**vestavěné predikáty** - to jsou procedury, které jsou implementovány přímo v systému) a dále procedury, které si uživatel definuje sám. Program je pak tvořen z jednotlivých procedur, které se skládají z příkazů. Příkazy jsou **nepodmíněné (**tj. **fakta)** a **podmíněné (**tj. **pravidla)**. Jedná se jen o jiné chápání – výklad – posloupností klauzulí.

Seznamové struktury: Velice častou datovou strukturou používanou v symbolických výpočtech jsou **seznamy**. Prolog chápe seznamy jako běžné prologovské struktury, tj. termy s funktorem a argumenty a také tak s nimi zachází. Seznam je tvořen posloupností prvků (tj. libovolných termů) oddělených čárkami a uzavřených do hranatých závorek **[]**. Prvkem seznamu tedy může být konstanta, proměnná či struktura. Seznam je určitou strukturou, která se však vyjadřuje jiným způsobem.

Př.:

```
[tohle, je, prvni, seznam, jen_z_konstant, 1]
[druhy, seznam, [je, [slozitejsi], obsahuje(strukturu)]]
```

Pro vytváření seznamů v Prologu existuje předdefinovaný funktor "**.**" (tečka), který je definován jako dvouargumentový funktor ("." je jménem struktury, označením vestavěného predikátu). Seznam se skládá z **hlavy** a **těla** (zbytku seznamu). Hlavou seznamu může být cokoli – atom, seznam, libovolný term, ale tělem seznamu je vždy seznam.

Př.: `.(Hlava, Tělo)`

Jedinou výjimkou je **prázdný seznam**, který neobsahuje žádné prvky: je reprezentován degenerovaným stromem `[]`, který se nedá rozložit na dvě části. Proto prázdný seznam **nemá hlavu ani tělo**. Ve skutečnosti se jedná o vyčleněný atom.

Př. prázdný seznam `[]`

Zápis seznamů: Seznamy lze zapisovat různými způsoby, Prolog však každý seznam interpretuje interně jako strukturu. V Prologu se k zápisu seznamu používá **binární funktor "."**. Jako syntaktická pomůcka se zavádí **"|" svislá (někdy přerušovaná) čára**, která odděluje hlavu seznamu od těla seznamu. Mezi zvláštnosti seznamů patří to, že programátor má k dispozici speciální **seznamovou notaci**, která je přehlednější a uživatelsky pohodlnější. Všechny tyto zápisy jsou ekvivalentní.

Př. - zápis čtyřprvkového seznamu pomocí funktoru "."

```
. (a, . (b, . (c, . (d, [] )))
```

Př. - zápis čtyřprvkového seznamu pomocí funktoru ".", jestliže chápeme "." jako infixový operátor

```
(a . (b . (c . (d . [] )))
```

Př. - zápis čtyřprvkového seznamu pomocí "|"

```
[a | [b, c, d]]
```

Př. - zápis čtyřprvkového seznamu pomocí seznamové notace

```
[a,b,c]
```

Z praktických důvodů se dovoluje používat svislou čáru k oddělení prvé části seznamu (tj. za druhou, třetí a další položkou) od zbytku seznamu, který musí být vždy seznam. Hlavou seznamu je i v tomto případě pouze první položka seznamu.

Př. - následující zápisy označují tentýž seznam, hlavou seznamu je i v těchto příkladech „a“

```
[a, b, c, d]
[a | [b, c, d]]
[a , b | [c, d]]
[a , b, c | [d]]
[a , b, c, d | [] ]
```

Př. - naopak tento příklad není seznamem, protože za symbolem "|" musí následovat opět seznam

```
[a | b]
```

Operace se seznamy: Seznamy jsou nejdůležitější struktury používané v Prologu. Mezi základní procedury pro práci s nimi patří append, member, length, delete, aj.

```
append  /* používá se pro spojování seznamů */
member  /* používá se pro určování příslušnosti prvku k seznamu */
length  /* určuje délku seznamu, tj. počet prvků seznamu */
delete  /* odstraňuje prvek ze seznamu */
```

Seznam je rekurzivně definovaná datová struktura, také procedury s ním pracující jsou rekurzivní.

Řízení databáze

Nejdůležitějšími predikáty pro řízení databáze jsou predikáty **assert** a **retract**. Umožňují ukládat klauzule do databáze nebo je z ní odstraňovat. Užitečným predikátem pro práci s databází je i predikát **listing** a **clause**.

```
assert /* ukládá klauzule do databáze */
asserta /* ukládá klauzule do databáze na začátek */
```

```
assertz /* ukládá klauzule do databáze na konec */
retract /* odstraní z databáze klauzuli */
retractall /* odstraní z databáze všechny klauzule */
listing /* způsobuje výpis klauzulí v databázi */
clause /* hledá klauzule podle kterých může unifikovat klauzule */
```

Řízení průchodu programem

- **! (řez)**
  - představuje klíčové rozhodnutí
  - překročí-li jej zpracovávání, už se nejde vrátit
  - lze jej využít k zefektivnění výpočtu - nenutíme Prolog, aby se navracel o mnoho podcílů zpět, pokud víme, že tak nenajde nová řešení.
  - dva typy řezů
    - **červený** - pokud jej z programu smažu, bude program pracovat chybně
    - **zelený** - pokud jej smažu, může dojít ke snížení efektivity, ale funkce programu se nezmění
- **fail**
  - nikdy nesplněný predikát
  - nelze se přes něj dostat
- **repeat**
  - při průchodu programem blokuje návrat
  - nekonečný cyklus:

```
repeat
  write('Pracuji'), nl,
fail.
```

Příklady řezů

```
pocitej :-
  repeat,
  write('Zadej cislo mensi nez 100'),
  read(S),
  S < 100, !,
  faktorial(S).
```

Zde je řez použit k oddělení načítání čísla, které má být menší než 100, od výpočtu faktoriálu. (Samotný predikát faktoriál v ukázce neuvádíme). Pokud se z nějakého důvodu nepodaří faktoriál vypočítat, Prolog se nebude vracet k novému načítání čísla a rovnou skončí. Jak vidíte, řezem je možné ovlivnit průběh výpočtu tak, že zamezíme zbytečným
návratům. Kromě toho lze ale řezem zabránit i nežádoucímu pohybu dopředu, přesněji řečeno zbytečnému vykonávání alternativních větví pravidla.

```
zjisti1(A) :- 0 is A mod 2, write('cislo je sude'), nl, !.
zjisti1(A) :- 1 is A mod 2, write('cislo je liche'), nl.
```

Pravidlo má zjistit, zda argument `A` je sudé, nebo liché číslo. Pokud se vykoná první větev pravidla - tedy číslo bude sudé - Prolog už nebude zkoumat, jestli je číslo liché. Předchozí variantu je možné zkrátit takto:

```
zjisti2(A) :- 0 is A mod 2, write('cislo je sude'), nl, !.
zjisti2(A) :- write('cislo je liche')
```

Všimněte si, že kdybychom z pravidla `zjisti1` odstranili řez, na jeho fungování by se nic nezměnilo (tzv. zelený řez). Avšak kdybychom řez vynechali v pravidle `zjisti2`, program by

nefungoval (tzv. červený řez). Jak by se projevila chyba, kdybychom odstranili řez z predikátu `zjisti2`?
Pro sudá čísla by pravidlo vracelo dvě řešení - hlášení, že číslo je sudé i že je liché. Pro lichá čísla by predikát fungoval dobře, neboť jeho první část by se nespustila.

Odlišujeme tedy dva druhy řezů:

- zelený řez, který je možné odstranit, aniž by se tím změnilo fungování programu
- červený řez, který nelze odstranit bez poškození programu

Databáze, databázový systém. Hlavní funkce DBS. Historický vývoj DBS. Modely dat. Relační algebra: projekce, selekce, spojení. SQL.

1. Hlavní funkce DBS

Databázový systém (DBS)- souhrn prostředků, které dovolují přístupným a bezpečným způsobem manipulovat s uloženými daty (bází dat) - vytvářet, modifikovat nebo aktualizovat je a především v těchto datech vyhledávat požadované informace.

DBS = BD + SŘBD + DBA

Báze dat (BD)	– množina informací, o nichž chceme mít přehled.
Systém řízení báze dat (SŘBD)	– program pro shromažďování a udržování dat.
Databázová aplikace (DBA)	– program, který umožňuje uživatelům přístup k datům, velkou část tvoří uživatelské rozhraní.

V architektuře DBS lze rozlišit tři úrovně:

Externí úroveň	- je reprezentována daty z pohledu uživatele
Konceptuální úroveň	- je schématem celé databáze
Interní úroveň	- koresponduje s fyzickým uložením dat na paměťových médiích

DBS by měl poskytnout:
– efektivní manipulaci s databází,
– souběžný přístup,
– ochranu dat,
– zotavení se z chyb systému.

2. Historický vývoj DBS

60. léta – první síťové SŘBD sálových počítačů, 1971 – SŘBD IMS od IBM pro program Apollo s hierarchickým uspořádáním dat, 1974 – počátky relační databáze a první verze SQL, 80. léta – prudký rozvoj relačních DB vývoj SQL, 90. léta – první objektově orientované databáze.

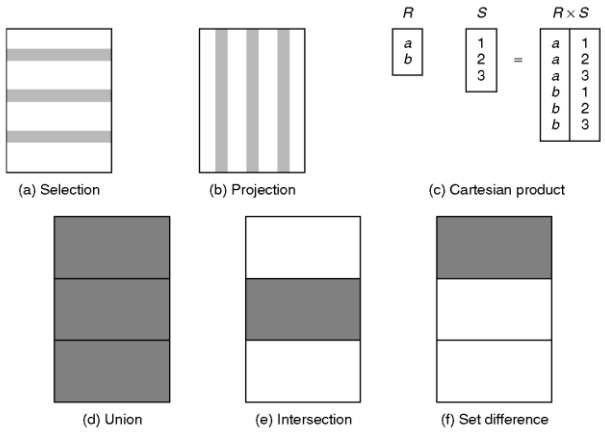
3. Databázové modely dat

Rozlišujeme čtyři modely logické struktury dat - hierarchický, síťový, relační a objektový.
Hierarchický databázový model. Všechny vztahy mezi daty mohou být znázorněny pomocí stromové struktury. Každý element má právě jednoho předchůdce, ale může mít více následovníků. Řeší snadno a rychle vztahy 1:N, problémy můžou nastat při řešení vztahů M:N a při změně struktury dat.
Síťový databázový model. Síťový model se vyznačuje tím, že každý prvek může mít více následovníků i předchůdců. Data jsou reprezentována kolekcemi záznamů a vztahů mezi nimi. Řeší snadno a rychle vztahy 1:N i M:N, problémy můžou nastat při změně struktury dat.
Relační databázový model. Nejrozšířenějším datovým modelem je model relační. Byl navržen s cílem dosažení datové nezávislosti a neporušenosti. Informace uchovávány v jednom typu objektu jsou uchovávány v tabulkách s určitou strukturou. Tabulky jsou navzájem provázány relacemi, relace pak usnadňují vyhledávání různých informací uložených v těchto tabulkách.
Objektový databázový model. Vychází z principů objektově orientovaného přístupu. Objektem je datová struktura definovaná jako třída s určitými vlastnostmi a metodami. Komunikace mezi objekty probíhá pomocí zpráv. Hlavní výhodou objektově orientovaných platforem je možnost přímého vyjádření složitostí modelované reality v databázi. Díky tomu, že součástí uložených objektů je také jejich chování, zjednodušuje se struktura aplikace. Objektově orientované databázové systémy jsou vhodnější pro práci se složitě strukturovanými daty (např. geografické systémy, průmyslový design).

4. Relační algebra: projekce, selekce, spojení

Relační model je založen na přesných výrocích teorie množin. Základem jsou množinové operace sjednocení, průnik, rozdíl a kartézský součin. Relační algebru lze chápat jako množinu operací na

relacích, pracuje s celými relacemi a výsledkem jsou opět relace. Základními operacemi nad relačními tabulkami jsou projekce, selekce a spojení.



Projekce tabulky definovaná podmnožinou atributů představuje vypuštění některých sloupců neobsažených v požadované množině atributů. Součástí operace projekce je i případné vypuštění duplicitních řádek ve výsledné tabulce. Tabulku lze naopak také rozšířit o sloupec vzniklý nějakou operací nad hodnotami uložených atributů. Projekce relace R na atributy B: **pB(R)**, B je podmnožina atributů R

Selekce tabulky definovaná podmnožinou definičního oboru relace (tj. logickou podmínkou charakterizující tuto podmnožinu) představuje průnik relace s touto novou podmnožinou.
Selekce relace R podle logické podmínky j: **sj(R)**

Spojení (join) dvou relací vytvoří třetí relaci, výsledná relace obsahuje všechny kombinace které vyhovují zadané podmínce porovnání hodnot. Podle způsobu porovnávání hodnot ve spojovaných sloupcích se rozlišují 3 druhy spojení – spojení přes rovnost (Equijoin), přes nerovnost (Theta join) a vnější spojení (Outer join).

Pokud výsledná relace obsahuje prvky na základě rovnosti hodnot stejnojmenných atributů, jde o přirozené spojení.

Další možností je spojení dvou tabulek bez omezení - kartézský součin tabulek. Výsledná tabulka obsahuje spojené řádky každý s každým. Např. má-li první tabulka 10 záznamů, druhá 20, tak výsledná po spojení bude mít 200 záznamů.

Vnější spojení dvou relací umožňuje zahrnout do výsledné relace i ty řádky, pro které neexistuje v druhé relaci stejná hodnota ve společném sloupci; chybějícím hodnotám druhé relace se přiřadí hodnota NULL. Zůstanou tak ve výsledné relaci zachovány i ty řádky, které by se při přirozeném spojení ztratily.

Left Outer Join – vytvoří A natural join B a řádky relace A, které nelze s ničím spojit, doplní hodnotami NULL. Výsledná relace bude obsahovat všechny řádky levé relace, tj. relace A.
Right Outer Join – vytvoří A natural join B a řádky relace B, které nelze s ničím spojit, doplní hodnotami NULL. Výsledná relace bude obsahovat všechny řádky pravé relace, tj. relace B.

5. SQL

Structured Query Language - strukturovaný dotazovací jazyk pro komunikaci s relačními databázemi SQL je nezávislý na datech, uživatel se nezabývá fyzickým uložením dat. Jazyk SQL se používá pro získání informací z databázových souborů, přičemž požadavky na výběr se specifikují popisně. SQL je standardem standardizačních systémů ANSI a ISO. SQL je neprocedurální jazyk, popisujme jaká data chceme najít, ne jak to má počítač provést.

Základní příkazy
SELECT (získávání dat), INSERT (vkládání dat), UPDATE (modifikace dat), a DELETE (mazání řádků)
Základní funkce příkazu SELECT:
SELECT – specifikuje které sloupce mají být zahrnuty ve výsledku.
FROM – specifikuje tabulku(y) která se má použít
WHERE – filtruje řádky
GROUP BY – vytváří skupiny řádků se stejnou hodnotou v daném sloupci
HAVING – filtruje skupiny podle dané podmínky
ORDER BY – specifikuje pořadí hodnot ve výsledku (ASC, DESC)
DISTINCT – eliminuje duplicity

Agregační funkce – mohou být použity pouze v seznamu příkazu SELECT a v klauzuli HAVING. Mezi agregační funkce patří:
COUNT – vrátí počet hodnot ve specifikovaném sloupci
SUM – vrátí součet hodnot ve specifikovaném sloupci
AVG – vrátí průměr hodnot ve specifikovaném sloupci
MIN – vrátí nejmenší hodnotu ve specifikovaném sloupci
MAX – vrátí největší hodnotu ve specifikovaném sloupci
Kromě COUNT(*), každá funkce nejdříve eliminuje hodnoty NULL a pracuje pouze se zbylými hodnotami.

Pohledy – je objekt, který uživateli poskytuje data ve stejné podobě jako tabulka. Na rozdíl od tabulky, kde jsou data přímo uložena, obsahuje pohled pouze předpis, jakým způsobem mají být data získána z tabulek a jiných pohledů. Jde o virtuální relaci, která se vytvoří v okamihu vyžádání. Výhody pohledů:
– soustřeďují potřebná data pro uživatele
– skrývají složitost podkladových dat
– zjednodušují správu uživatelských oprávnění
– definují uspořádání dat pro export do jiných aplikací
Uložená procedura (anglicky stored procedure) je databázový objekt, který neobsahuje data, ale část programu, který se nad daty v databázi má vykonávat.
Trigger (česky spoušť) v databázi definuje činnosti, které se mají provést v případě definované události nad databázovou tabulkou. Definovanou událostí může být například vložení nebo smazání dat.
Transakce – je skupina, posloupnost příkazů, která se navenek tváří jako jeden příkaz. V SQL pro transakce existují tři základní příkazy:
BEGIN – začátek transakce,
ROLLBACK – odvolání transakce, všechny změny se anulují,
COMMIT – potvrzení transakce, změny jsou trvale zaznamenány.
Privilegia (oprávnění) uživatelů – jsou akce (příkazy), které uživatel může realizovat tabulkou nebo pohledem příslušné databáze. Privilegia uživatelů můžou být omezena na vyjmenované databáze, tabulky nebo sloupce.Vlastník databáze nebo tabulky uděluje ostatním uživatelům potřebná privilegia použitím klauzule GRANT, odebírá pak použitím klauzule REVOKE.

Příklad SQL příkazu

```
SELECT [DISTINCT | agregacni_funkce]
    atributy
    FROM název_tabulky
    [WHERE podmínka_výběru]
    [GROUP BY atributy [HAVING podmínka_agregace [agregacni_funkce]]]
    [ORDER BY atributy]
 [UNION [ALL]]
```

Příklad SQL procedury (pro firmu vyhleda a zobrazi jeji pobocky)

```
CREATE PROCEDURE PobockyFirmy
        @idFirmy int
        AS SELECT f.nazev As "Název firmy", p.doplnkovyNazev AS "Pobočka",
        A.ulice AS "Ulice", O.obec AS "Obec", O.psc AS "PSČ",
        P.telefon AS "Telefon", P.fax AS "Fax", P.email AS "E-mail",
        P.www AS "WWW" FROM Pobocka P left outer join Firma AS F ON P.idFirmy = F.idFirmy,
Obec AS O, Adresa AS A
        WHERE
        F.idFirmy = @idFirmy AND P.idAdresy = A.idAdresy
        AND A.idObce = O.idObce
```

Trigger SQL (v případě smazání firmy smaže také všechny navázané záznamy jako jsou pobočky, osoby atp)

```
CREATE TRIGGER smazVseProFirmu ON Firma INSTEAD OF DELETE AS
DECLARE @Firma int, @FirmaNazev varchar(40)
SELECT @Firma = idFirmy FROM Deleted
SELECT @FirmaNazev = nazev FROM Deleted
DELETE FROM Aktivita WHERE Aktivita.idOsoby IN (SELECT Osoba.idOsoby FROM Osoba
WHERE Osoba.idFirmy = @Firma)
DELETE FROM Prilezitost WHERE Prilezitost.idOsoby IN (SELECT Osoba.idOsoby FROM Osoba
WHERE Osoba.idFirmy = @Firma)
DELETE FROM Osoba WHERE Osoba.idFirmy = @Firma
DELETE FROM Pobocka WHERE Pobocka.idFirmy = @Firma
DELETE FROM Firma WHERE Firma.idFirmy = @Firma
PRINT 'Byla smazána firma ' + @FirmaNazev=
```

Konceptuální modelování. E-R model a jeho grafické znázornění. Relační model. Typy vztahů mezi entitami a jejich reprezentace v relačním modelu. Vlastnosti relační tabulky. Normální formy relačního schématu.

1. Entity-Relationship (E-R) model a jeho grafické znázornění

E-R model je prvním krokem návrhu databáze. Jde o modelování databáze na konceptuální úrovni. Pracuje s pojmy entita = objekt, typ entity = množina objektů stejného typu, relationship = vztah, atribut = vlastnost. E-R model představuje tedy metodu modelování entit a vztahů mezi nimi. U jednotlivých vztahů je definována kardinalita vztahů, u atributů pak doména atributu určující přípustné hodnoty. U atributů je určován i primární klíč - minimální množina atributů jednoznačně identifikující entitu. Pro grafické znázornění E-R modelu je možno využít např. notace UML.

2. Relační model

Základy byly položeny v sedmdesátých letech (E.F.Codd). Hlavním cílem bylo vnést disciplínu do způsobu manipulace s daty, zabezpečit nezávislost dat (aby se při manipulaci s daty nebylo nutné zajímat o přístupové mechanizmy) a zvýšit produktivitu programování. Relační model je založen na vlastnostech relační algebry. Databáze je kolekce databázových relací reprezentovaných tabulkami. Všechny informace jsou uloženy v relacích. Databázová relace se liší od matematické definice ve 2 aspektech:

- Relace je vybavena pomocnou strukturou - schématem relace. Schéma relace tvoří:
  - jméno relace
  - jména atributů
  - definice domén
- Hodnoty jsou atomické

V roce 1985 Codd specifikoval 5 skupin pravidel pro relační databázové systémy:

1. základní pravidla
2. pravidla týkající se struktury dat
3. pravidla týkající se integrity dat
4. pravidla týkající se modifikace dat
5. pravidla o nezávislosti dat

Pravidel je celkem 12, jde o pravidla jako např. schopnost RDB manipulovat s daty pomocí operací relační algebry, podpora relací, domén, primárních klíčů, cizích klíčů, podpora hodnoty NULL na reprezentaci chybějící informace, nezávislost dat od aplikace, která data používá atd.

Transformace E-R modelu do relačního modelu

Typ entit reprezentujeme tabulkou, její sloupce tvoří atributy. Vztah M:N reprezentujeme tabulkou, která obsahuje primární klíče typů entit, participujících na daném vztahu. V případě vztahu 1:1 nebo N:1 není nutné vytvářet novou tabulku. U vztahu N:1 definujeme cizí klíč na straně N. Kompozici reprezentujeme tabulkou, přičemž do nové tabulky zahrneme klíč rodičovské tabulky.

3. Typy vztahů mezi entitami a jejich reprezentace v relačním modelu

Vztah vyjadřuje určité propojení mezi entitami. Kardinalita vztahu vyjadřuje kolik entit jednoho typu může být ve vztahu s kolika entitami z druhého typu entit.

Vztah typu 1:1 – vztah, ve kterém na obou stranách vystupuje pouze jeden objekt dané entity.
Vztah typu 1:N – na jedné straně je jediný objekt, který je ve vztahu s jedním nebo více objekty na straně druhé. Jedná se o typ, který se vyskytuje velmi často (např. oddělení a zaměstnanec).
Vztah typu M:N – vztahy, kde vystupuje více objektů na obou stranách (např. zaměstnanec a úkol, kde jeden úkol může řešit více zaměstnanců a současně jeden zaměstnanec může řešit více úkolů).

ISA hierarchie
U ISA vztahu hovoříme o nadtypech a podtypech. Hlavní účel zavedení ISA vztahu mezi typy entit je, že jedna entita může zdědit atributy druhé entity, ale také může mít další atributy. Podentity jsou identifikovány výhradně předkem, tj. všechny entity v ISA hierarchy sdílí jediný identifikátor. Podentita může pak mít i další, speciální způsob identifikace.

4. Vlastnosti relační tabulky

- Každá tabulka má jednoznačné jméno.
- Každý sloupec v tabulce má jednoznačné jméno.
- Všechny hodnoty daného sloupce jsou stejného typu.
- Nezáleží na pořadí sloupců.
- Nezáleží na pořadí řádků.
- Tabulka nemůže mít duplicitní řádky.
- Všechny hodnoty jsou atomické.
- Každá tabulka musí mít primární klíč.

5. Normální formy relačního schématu

Při splnění určitých podmínek lze tabulku relační databáze prohlásit za normalizovanou. Normalizace přispívá k zamezení redundancí, k posílení integrity dat a omezuje vznik anomálií při aktualizaci a údržbě dat. Většinou jsou tabulky normalizovány do třetí normální formy. Platí tato pravidla:

- relace je v 1. normální formě, jsou-li všechny její atributy atomické, tj. dále nedělitelné;
- relace je v 2. normální formě pokud je v 1. normální formě a současně je každý neklíčový atribut plně závislý na primárním klíči;
- relace je v 3. normální formě pokud je v 2. normální formě a současně jsou všechny její neklíčové atributy vzájemně nezávislé.

Funkční závislost
Je dána relace R(A,B), kde A,B mohou být složené atributy.
B je funkčně závislý na A, když pro každou hodnotu A je jednoznačně daná hodnota B.
B je plně funkčně závislý na A, je-li funkčně závislý na A a není funkčně závislý na žádné podmnožině A.

Postup normalizace
Není – li relace v 1.NF (obsahuje vícehodnotový atribut), vytvoříme novou tabulku.
Není-li relace v 2.NF, tak vytvoříme projekci, abychom eliminovali neúplné funkční závislosti na primárním klíči.
Není-li relace v 3.NF, tak vytvoříme projekci, abychom odstranili tranzitivní závislosti.

Příklad:
R(**<u>A, B</u>**,C,D); A $\rightarrow$ D …. není v 2.NF – vytvoříme projekci: R1(A,B,C) a R2(A,D)
S(**<u>E</u>**, F,G); F $\rightarrow$ G … není v 3.NF – vytvoříme projekci: R1(E, F) a R2(F,G)

Umělá inteligence: základní pojmy a definice, oblasti výzkumu umělé inteligence, předmět výzkumu, metody UI. Úloha symbolů v umělé inteligenci, systémy symbolů, hypotéza o systémech symbolů.

Základní pojmy

- Expertní systém = počítačový program, který má za úkol poskytovat expertní rady, rozhodnutí nebo doporučit řešení v konkrétní situaci.
- Neuronové sítě = výpočetní model používaný v UI (Umělá neuronová síť (ANN, artificial neural network) je prostředkem pro zpracování komplexních dat, využívajícím ke své práci množství propojených procesorů a výpočetních cest.)
- Umělý život = evoluční algoritmy, které se neustále zdokonalují tak, že se kombinují dohromady a dobré výsledky se zachovají. Tyto kombinace jsou ještě doplněny náhodnými změnami – mutacemi. atp.
- Turingův test - pojmenovaný podle svého tvůrce Alana Turinga, který jej prezentoval roku 1950) je pokus, který má za cíl prověřit, jestli nějaký systém umělé inteligence se opravdu chová inteligentně. Jelikož inteligence je pojem, který lze jen těžko definovat, tím hůře testovat, používá Turingův test porovnání s člověkem.

Definice umělé inteligence

- Vědní disciplína, zabývající se teorií systému zpracovávajících znalosti a schopných se více méně samostatně rozhodovat.
- (M. Minsky 1967) Věda o vytváření strojů nebo systémů, které budou při řešení určitého úkolu užívat takového postupu, který –kdyby ho dělal člověk – bychom považovali za projev jeho inteligence
- (E. Rich 1991) Zabývá se tím, jak počítačově řešit úlohy, které dnes zatím zvládají lépe lidé

Směry výzkumu

- Inženýrský (snaha konstruovat systémy, které se chovají „inteligentně“)
- Psychologický (poznat zákonitosti lidského myšlení se snahou pochopit a modelovat je)

Oblasti výzkumu UI

- Jazyky pro UI
- Matematická logika
- Reprezentace znalostí
- Metody řešení úloh

Aplikační oblasti

- **Expertní systémy** - znalostní inženýr zpovídá experta v dané oblasti a snaží se vložit jeho znalosti do systému.

- **Neuronové sítě** - šance simulovat některé funkce lidského myšlení.

- **Počítačové vidění** - svět je složen z třídimenzionálních objektů, ale do lidského oka i do kamery robota vstupuje 2D obraz. Některé systémy pracují v 2D, ale plnohodnotné počítačové vidění vyžaduje 3D informaci, která není jen množinou 2D obrazů. V současnosti jsou jen omezené možnosti, jak reprezentovat 3D scénu, a ty nejsou ani zdaleka tak dobré, jako lidské oko.

- **Rozpoznávání řeči** - v 90. letech dosáhlo praktických výsledků pro limitované účely. Dnes je možné instruovat počítač řečí, ale většina uživatelů se vrátila ke klávesnici a myši jako k vhodnějším.

- Analýza scén
- Robotika

- Řešení úloh a plánování
- Strojové učení
- Zpracování řečové informace
- Porozumění přirozenému jazyku
- Umělý život

Metody umělé inteligence

- využívají znalosti, přičemž tyto znalosti by měly být reprezentovány tak, že pokrývají zobecnění
    - nereprezentuje se každá individuální situace
    - slučují se případy mající podstatné společné rysy
- jsou blízké pojmům, ve kterých je chápou a používají lidé
- dají se lehce modifikovat
- dají se použít ve velkém počtu případů

Základní metody

- prohledávání
- využití znalostí
- abstrakce

Další metody

- **metoda cílů a prostředků**, která spočívá v redukci výchozí úlohy na řadu dílčích úloh (podcílů), jejichž kombinace řešení vede k vyřešení výchozí úlohy. Tento proces má rekurzivní charakter ve smyslu redukce složitějších úloh na jednodušší. Celý proces redukce končí úspěšně v případě nalezení elementárních úloh, pro něž máme prostředky k vyřešení, nebo neúspěšně, nelze-li takovou posloupnost nalézt.
- **metoda zkoušek a omylů**, jejíž princip lze pochopit např. jako metodu, která se používá pro cestu bludištěm. Účinek této metody je zesilován použitím heuristik při určování dalšího postupu. S tím souvisí metody prohledávání stavových prostorů
- **metoda analogie**, spočívající v aplikaci postupů, které se už ukázaly úspěšné při řešení jistého okruhu problémů, na úlohy podobné nebo blízké
- **použití opačného postupu**, kdy vycházíme od řešení úlohy a opačnými kroky se snažíme přijít k výchozí situaci. Tento postup není neobvyklý ani u lidského myšlení a používá se také ve formálních systémech, v nichž je znám počáteční stav úlohy, výsledný stav úlohy a elementární transformace jednoho stavu ve druhý.
- **práce s formálními systémy**, používaná zejména u systémů matematicko-logických, směrujících k tvorbě systémů pro dokazování vět, deduktivních systémů atd.

Úloha symbolů v UI

Symboly tvoří základ nejúčinnějších způsobů zpracování informací technickými systémy. Všechny metody UI jsou postaveny na manipulaci se symboly. Na symbolové úrovni jsou reprezentace objektů definované pomocí symbolů, kterými mohou různé programy manipulovat.
Jakékoliv další vlastnosti, které nabývají symboly svou fyzikální existencí, jsou pro informatiku nepodstatné a v rámci informatiky se neuvažují. Cokoliv, co splňuje tyto předpoklady, může plnit v informatice, a tedy i v umělé inteligenci úlohu symbolů:

- symboly jsou entity, které jsou fyzikálně realizovatelné a které mají v prostoru a čase jisté trvání. Můžeme je identifikovat a jejich kvalitativní určenost se v čase nemění tak, aby to ovlivňovalo naše úvahy o nich
- jsme schopni z nich konstruovat rozsáhlejší struktury. Tyto struktury pak dědí všechny vlastnosti symbolů. Symboly a jejich struktury jsou reprodukovatelné
- symboly mají schopnost označovat kromě sebe i jiné entity, i takové, které existují pouze v našich představách. Ve spojení s vhodným zařízením na zpracování symbolů mají schopnost označovat také postupy zpracování entit svého typu. Vztah označované entity a označujícího symbolu trvá, dokud to předpokládáme. Tento vztah máme možnost kdykoliv přerušit.

Systémy symbolů
Koncem třicátých let navrhl A. M. Turing abstraktní zařízení, které precizně matematicky definoval a které dnes nazýváme Turingův stroj. V podobě tohoto pojmu má informatika k dispozici nejenom abstrakci symbolů, ale také zařízení, která jsou schopna symboly a z nich vytvořené struktury zpracovávat. Nazýváme je systémy symbolů. Příkladem je abstraktní model počítače, např. právě už zmíněný Turingův stroj.

Fyzikální systémy symbolů jsou technická zařízení na zpracování symbolů ve tvaru elektronicky vytvořených stop na vhodných paměťových mediích. Dnes to jsou převážně počítače. Počítače se osvědčily i při rozhodování na takové kvalitativní úrovni, která předpokládá u lidí dlouhodobou odbornou přípravu a praxí nabyté zkušenosti. To je případ expertních systémů.

Hypotéza o systémech symbolů
Hypotézu vyslovili v roce 1976 dva průkopníci umělé inteligence A. Newell a H. A. Simon. Jde o domněnku, vyslovenou na základě jisté empirické zkušenosti. Úkolem UI je vyvrátit nebo potvrdit tuto hypotézu, případně nalézt hranice jejího opodstatnění Zmíněná hypotéza zní: *„Systémy zpracování symbolů umožňují vytvořit všechny nutné a postačující podmínky inteligence."*

Metody a principy neinformovaného hledání řešení problémů. Metody a principy heuristického hledání řešení problémů. Produkční (pravidlové) systémy.

1. Řešení úloh v UI

Řešení úloh je důležitou částí oboru umělé inteligence. Jedním ze způsobů řešení úloh je **prohledávání stavového prostoru**. Je-li dán počáteční model prostředí a požadovaný koncový model prostředí, je úkolem systému na řešení úloh hledat vhodnou posloupnost akcí, jejichž aplikací lze přejít od počátečního modelu ke koncovému. Každému modelu odpovídá určitý **stav** prostředí, množina těchto stavů se nazývá **stavový prostor řešení**. Stavový prostor lze reprezentovat orientovaným grafem (tj. hrany grafu jsou orientované →). Každý uzel reprezentuje stav, každá orientovaná hrana přechod mezi stavy. Řešení úlohy lze pak formulovat jako hledání cesty v orientovaném grafu.
Hledání musí být **systematické**:

- nevynechá ani jeden objekt
- ani jeden objekt nevybere dvakrát.

Obecný **algoritmus prohledávání** grafu:

- z množiny otevřených uzlů vybere jeden, přeřadí ho mezi uzavřené uzly, expanduje jej - vytvoří jeho následovníky a ty zařadí mezi otevřené uzly - kromě těch, které se již nacházejí v množině otevřených nebo množině uzavřených uzlů.
- jestliže v množině otevřených uzlů se nachází cílový, prohledávání je úspěšně ukončeno
- jestliže množina otevřených uzlů je prázdná, prohledávání je neúspěšně ukončeno
- jinak se opět vrátí k výběru uzlu z množiny otevřených uzlů. Expandováním a přidáváním nových uzlů rozšiřujeme podgraf prohledávacího grafu.

Expanze uzlu

- u zvolené cesty vytvoříme všechny přímé následovníky jejího posledního uzlu a všechny je přibereme do reprezentačního podgrafu. Když dosáhneme cílový stav, budování podgrafu skončí.

Tato strategie pracuje se 2 množinami uzlů :

- množiny otevřených uzlů (OPEN) = všechny uzly, které ještě nebyly prozkoumány. Uzel se stává otevřeným v okamžiku, kdy je vytvořen při expanzi svého předka. Pro otevřený uzel platí, že jeho následovníci ještě nebyli vytvořeni a nejsou ještě součástí reprezentačního podgrafu.
- množiny uzavřených uzlů (CLOSE) = všechny uzly, které již byly prozkoumány. Uzel se stává uzavřeným v okamžiku, kdy je expandován a jsou vytvořeni jeho následovníci.

Podle toho, jak jsou znalosti o dané úloze využity v mechanismu prohledávání stavového prostoru, rozlišujeme algoritmy **informované** a **neinformované**.

2. Metody a principy neinformovaného hledání řešení problémů

- **slepé prohledávání do šířky**
  - při slepém prohledávání do šířky (breadth-first search) se nejdříve expanduje uzel s minimální hloubkou
  - hloubkou uzlu ve stromu řešení rozumíme počet hran na cestě od počátečního uzlu k danému uzlu
- **slepé prohledávání do hloubky**
  - při tomto prohledávání se přednostně expanduje uzel s největší hloubkou
  - prohledávání do hloubky je často spojeno s omezením maximální prohledávané hloubky, při jejímž dosažení se používá mechanizmus navracení (backtracking)

3. Metody a principy heuristického hledání řešení problémů

Informované metody prohledávání stavového prostoru řešení problému používají pro výběr nejvhodnějšího vrcholu k expandování různé typy heuristik. **Heuristika** je postup, jak postupovat při výpočtu konkrétní situace, který není hromadný, platí jen pro konkrétní hodnoty a konkrétní problémy a zpravidla byl odpozorována z praxe. K rozhodnutí, který uzel grafu expandovat se používají různé **hodnotící funkce**. Platí, že čím kvalitnější heuristické znalosti o problému jsou použity k formulaci hodnotící funkce, tím efektivnější bude prohledávání. Podle obecné dohody tato funkce vybírá ten uzel, ve kterém nabývá minimální hodnotu - je-li takových uzlů víc, pak náhodně jeden z nich. Pokud hodnotící funkce dobře postihuje vlastnosti a charakter úlohy, budou vždy expandovány „nejperspektivnější“ uzly a zabrání se prohledávání cest, které nevedou k cíli. Nejjednodušší verzí informovaného algoritmu je **gradientní algoritmus**, resp. cesta nejvyššího gradientu. Gradientní algoritmus expanduje ten uzel, který byl vyhodnocen pomocí hodnotící funkce f jako nejlepší, a vyhodnocuje jeho následovníky. Následovník, který má "nejlepší" ohodnocení funkcí f je vybrán k další expanzi. "Rodič" i "sourozenci" tohoto uzlu jsou ihned zapomenuti, v paměti je jen rozvíjený uzel. Prohledávání je zastaveno, pokud je dosaženo stavu, který má lepší ohodnocení funkcí f, než jeho následovníci. Podstatnou nevýhodou tohoto algoritmu je možnost uváznutí v lokálním extrému. Vzhledem k zapomínání ostatních uzlů než právě rozvíjeného není vyloučeno zacyklení.
Další možností použití heuristiky může být cesta **nejmenšího odporu** (nejmenší náklady, energie apod.).

4. Produkční (pravidlové) systémy

- poskytují vhodnou strukturu na popis a provádění procesu prohledávání
- cílem činnosti pravidlového systému je vyprodukování dat (údaje), která (který) používatel považuje za řešení svého problému
- znalosti jsou v systému reprezentovány pomocí pravidel
- produkční pravidla reprezentují poznatek ve formě pravidel:
  - *jesliže platí* předpoklady => *pak platí* důsledek
- typickým případem reprezentace znalostí použitím pravidel je logické programování
- tři základní složky produkčních systémů:
  - **báze dat** (reprezentace faktů)
    - část může obsahovat trvale platná fakta
    - část může obsahovat aktuálně platná fakta
  - **báze (produkčních) pravidel**
    - předpokladová část (předpoklady) → důsledková část (důsledek, akce)
  - **inferenční mechanizmus (interpreter)**
    - řídí proces aplikací produkčních pravidel na obsah báze dat, tj. řídí výběr pravidla, vykonání pravidla a uložení výsledku
    - cyklicky provádí tři kroky:
      1. Porovnání se vzorem
      2. Řešení konfliktových situací - je-li výsledkem kroku 1. více než jedno pravidlo, je třeba zvolit mezi nimi pro aplikaci jediné
      3. Aplikace zvoleného pravidla
    - určuje, jak a v jakém pořadí aplikovat pravidla na bázi dat, principiálně lze rozlišit:
      - přímé (dopředí) řetězení, kdy při aplikaci produkčních pravidel postupujeme ve směru od počátečního stavu k některému ze stavů cílových (strategie řízená daty)
      - zpětné řetězení, kdy se vychází od cíle ve směru počátečních stavů (strategie řízená cílem)

Expertní systémy – základní pojmy, architektura. Znalostní inženýrství. Znalostní projekty, jejich řízení a příčiny selhání. Aplikace expertních systémů, typy úloh pro aplikaci expertních systémů.

Základní pojmy
Expert

- osoba, která má hluboké znalosti v nějaké oblasti lidské činnosti a je schopna rychle a kvalitně řešit problémy, které se v této oblasti běžně ale i zřídka kdy vyskytují
- má v řešení těchto problémů zkušenosti, dovede tyto zkušenosti vhodně a v pravý čas využít
- nezřídka do postupů řešení různých problémů zapojuje i ty své schopnosti, které obvykle nazýváme intuicí, citem pro okolnosti řešení problému, atd.

Báze znalostí

- obsahuje znalosti experta potřebné k řešení zvoleného problému
- zachycuje znalosti od nejobecnějších, učebnicových, až po úzce speciální, od všeobecně známých až ke znalostem „soukromým“, od exaktních až k nejistým

Expertní systém

- program, který využívá vhodně v počítači uložených poznatků lidských expertů k řešení problémů, které obvykle v praxi vyžadují znalost expertů
- napodobují činnost lidských expertů při řešení problémů

Fakta

- je třeba je identifikovat
- jsou relevantní k řešenému problému
- musíme zajistit způsob, jakým je bude systém zpracovávat při změně uvažovaného světa

Heuristiky

- považujeme je za nejvyšší úroveň znalostí
- na této úrovni je expert schopen vytvářet nová pravidla určena ke zpracování různých, někdy dokonce unikátních okolností a problémů
- Heuristické řešení je často jen přibližné, založené na poučeném odhadu, intuici, zkušenosti nebo prostě na zdravém rozumu. První odhad se může postupně zlepšovat, i když heuristika nikdy nezaručuje nejlepší řešení. Zato je univerzálně použitelná, jednoduchá a rychlá.
- Nejjednodušší heuristická metoda je pokus a omyl, kterou lze použít kdekoli, od přihledávání šroubů na kolo až po řešení algebraických problémů.

Architektura

Znalostní inženýrství

- Hlavní náplní je vytvoření expertního systému (tzv. znalostní aplikace)
- Expertní systém vniká spoluprací experta a znalostního inženýra (iterativní proces)
- Účelem získávání znalostí je opatření všech potřebných znalostí pro tuto činnost

Fáze vývoje expertního systému

- Analýza problému (identifikace probledá, posouzení vhodnosti)
- Specifikace
- Vývoj (konceptuální návrh, návrh implementace, implementace, vyhodnocení)
- Využití (nasazení systému, údržba systému)

Projektový tým znalostních projektů:

- řádní členové (vedoucí projektu-celková odpovědnost, znalostní inženýři-získávání znalostí, návrháři-výběr prostředí, vývojoví pracovníci-zprovoznění prostředí)
- podpůrní členové týmu (manažer projektu-příprava cíl.prostředí, manažeři dílčích úkolů, doménoví experti-zdroj znalostí, uživatelé)
- konzultanti (specialisté, externí poradci, analytici)

Typy selhání

- *základní* selhání (projekt nedokončen, nedosaženy žádné konkrétní výsledky)
- *primární* selhání (projekt dokončen, neplní uspokojivě všechny stanovené cíle nebo specifikace-systém je bezcenným)
- *sekundární* selhání (selhává v plnění několika důležitých požadavků-lze modifikovat)
- *kvalitativní* selhání (omezení použití systému)
- *časové a nákladové* selhání (špatně řízený projekt)

Typy selhání¨

- chybí manažerská podpora
- stanovené cíle příliš vysoko
- malé zapojení uživatele do projektu
- podcenění operačních a organizačních činností
- chybí doménoví experti / příliš mnoho doménových expertů
- slabý projektový tým
- použití neadekvátních (neefektivních) nástrojů

Aplikace expertních systému

- programovací jazyky (LISP a jeho varianty, PROLOG, C, C++, Pascal, …)
- vývojová prostředí (EXSYS Professional, M.4, G2, RTworks)

Systém G2

– prostředí pro vývoj rozsáhlých expertních systémů reálného času s vysokými nároky na kontinuální a inteligentní sledování procesů, diagnostické účely a řízení

- možné aplikační oblasti jsou řízení výrobních procesů, telekomunikace, medicína, řízení kosmických letů, finančnictví nebo robotika

Typy expertních úloh
diagnostické

- diagnóza (proces nalezení chyb či chybných funkcí systému)
- interpretace (analýza dat s cílem určení jejich významu)
- monitorování (interpretace signálů a dat a určení okamžiku, kdy je nutná intervence)

generativní

- návrh (vytváření konfigurací objektů vyhovujících daným podmínkám)
- plánování (nalezení posloupnosti akcí k dosažení cíle)
- predikce (předpověď běhu budoucích událostí na základě modelu minulosti a současnosti)

Objektové modelování a programování - základní pojmy, podstata, využití. Softwarový proces. UML. Událostmi řízené programování. Architektura MVC.

základní pojmy

- **Program** – posloupnost příkazů popisujících určitou činnost
- **Proces** – prováděný (běžící) program
  - **Sekvenční** – instrukce zpracovávány postupně
  - **Paralelní** - instrukce zpracovávány souběžně
- **Procesor** – zařízení, které dokáže vykonávat příkazy programu
- **Data** – objekty (údaje), s nimiž pracují procesy
- **Zdrojový kód** – kód programu zapsaný v nějakém programovacím jazyce
- **Cílový kód** – binární kód, (vytvořený po překladu zdr. kódu) je spustitelný
- **Bytekód (java)** – spustitelný mezikód, vzniká jako cílový kód, ale je spouštěn a prováděn běhovým prostředím (Runtime Environment, Virtuální stroj)

Objektové programování – používá programovací jazyk k napodobení objektů skutečného světa definováním tříd.

Zásady objektového programování (zdroj: Rudolf Pecinovský)

- Programovat proti rozhraní a ne proti implementaci
- Dbát na důsledné zapouzdření a skrývání implementace
- Zapouzdřit a odpoutat části kódu, které by se mohly měnit
- Maximalizovat soudržnost (cohesion) entit (balíčků, tříd a metod). Každá entita by měla řešit jen jeden konkrétní úkol
- Koncentrovat zodpovědnost za řešení úkolu na jednu entitu – návrh řízený zodpovědnostmi (responsibility driven design)
- Minimalizovat vzájemnou provázanost (coupling) entit
- Vyhýbat se duplicitám kódu

Podstata, využití

základní objektové koncepty

- **Abstrakce** – separování důležitých rysů od nedůležitých v závislosti na kontextu
- **Zapouzdření (encapsulation)** – data a operace objektu tvoří nedělitelný celek
- **Dědičnost (inheritance)** – schopnost objektů dědit vlastnosti a chování předka
- **Polymorfizmus** – jev, kdy operace stejného jména je používána pro více objektů odlišných tříd. Vlastnost, která umožňuje:
  - jednomu objektu volat jednu metodu s různými parametry (parametrický polymorfismus)
  - objektům odvozených z různých tříd volat tutéž metodu se stejným významem v kontextu jejich třídy, často pomocí rozhraní
  - přetěžování operátorů znamená provedení operace v závislosti na typu operandů (operátorový polymorfismus)
- **Komunikace** (zasílání zpráv = volání metod) – objekty mezi sebou komunikují zasíláním zpráv. Výsledkem přijetí zprávy příjemcem je vykonání nějaké operace

Objektové modelování – snaha reálně popsat existující (nebo vznikající) systém v zjednodušené (abstraktnější) podobě

Význam modelování

- snadné změny s nízkými náklady oproti reálnému systému
- usnadnění komunikace v týmu a se zákazníkem

- přehled o aktuálním stavu projektu
- vytváření dokumentace

Princip tří architektur

- konceptuální úroveň – model reality, nejvyšší abstrakce, popisuje obsah IS, ne formu (CO)
- technologická úroveň – popis technologie s ohledem na prostředí implementace (JAK)
- fyzická úroveň – popis detailů implementace v konkr. prostředí (ČÍM)

Softwarový proces

Životní cyklus informačního systému (IS)

- Analýza firemního prostředí – jak funguje firma
- Analýza IS – co má IS nabízet
- Návrh IS – jak realizovat požadovaný systém
- Implementace IS – realizace – programování
- Nasazení IS – nasazení u zákazníka

Unified Modeling Langue (UML) - grafický jazyk pro specifikaci, vizuální popis, tvorbu a dokumentaci jednotlivých součástí softwarového systému. Jazyk pro OO modelování.
Modelování typových úloh – nástrojem je **Diagram typových úloh** (Use Case Diagram), vyjadřuje vztahy Aktér-úloha a úloha-úloha. Důležité jsou scénáře.

Modelování tříd – nástrojem je **Diagram tříd (**Class Diagram**).** Jeto základní strukturální diagram UML.

- **Třída** – abstraktní definice množiny objektů (atributy, metody)
- **Rozhraní tříd** – definuje kontrakt, ke kterému se třídy přihlašují
- **Asociace** – definují vztahy mezi objekty

Modelování dynamiky systému – dva základní (vzájemně izomorfní) diagramy **OSD** a **OCD**

- **sekvenční diagram** (Object Sequence Diagram - OSD) – zobrazuje interakci objektů s důrazem na časovou posloupnost, mapován k jedné typové úloze
- **diagram objektové spolupráce** (Object Collaboration Diagram - OCD) – pohled na strukturu spolupráce – vztahy mezi objekty
- **diagram aktivit** – lze použít pro modelování chování, ale není izomorfní s OSD a OCD, modeluje typovou úlohu jako posloupnost aktivit – ne jako interakci uživatele a systému

Událostmi řízené programování (Event-Driven Programming) – událost vzniká buď jako výsledek interakce mezi uživatelem a GUI nebo jako důsledek změny vnitřního stavu aplikace či OS.
Obsluha události – úsek kódu, který je při vzniku události automaticky vyvolán a provádí činnost k události připojenou (Event Handler)

Typy událostí

- klik/dvojklik
- změna stavu komponenty
- stisk/uvolnění klávesy/tl. myši
- překreslení
- událost systému nebo zpráva časovače

Architektura MVC - Model-view-controller (MVC) - je softwarová architektura, která rozděluje **datový model** aplikace, **uživatelské rozhraní** a **řídicí logiku** do tří nezávislých komponent tak, že modifikace některé z nich má minimální vliv na ostatní.

- **Model (model) -** doménově specifická reprezentace informací, s nimiž aplikace pracuje

- **View (pohled)** - převádí data reprezentovaná modelem do podoby vhodné k interaktivní prezentaci uživateli
- **Controller (řadič) -** reaguje na události (typicky pocházející od uživatele) a zajišťuje změny v modelu nebo v pohledu

Čistě OOP versus hybridní prog. Jazyky

- **čisté OOP** jazyky jsou jazyky, které nepřipouštějí jiné programovací modely. Funkci nemůžeme napsat samostatně, pokud není součástí třídy. Nemůžeme deklarovat globální proměnnou. Příklady čistých jazyků jsou Smalltalk a Eiffel.
- **S hybridními** jazyky můžeme dělat cokoli chceme včetně úplného vypuštění OOP principů. Příklady hybridních jazyků jsou všechny ty, které jsou kompatibilní s již existujícími, jako například C++ nebo Object Pascal.

Zdroje:

Přednášky PRO1, OMO1, PRO2
http://cs.wikipedia.org/wiki/Model-view-controller
http://cs.wikipedia.org/wiki/Polymorfismus_(programování)
http://www.fi.muni.cz/usr/jkucera/pv109/2002/xkriz1.htm

Práce s kolekcemi – algoritmy a implementace ve zvoleném programovacím jazyce.

„Kolekce jsou objekty, které seskupují skupinu elementů do jednoho celku (=kontejnery)“

Další zdroje vhodné pro prostudování:
Kontejnery I. - Úvod do problematiky, Základy práce s kolekcemi v jazyce Java
Kontejnery II. - Kategorie kontejnerových rozhraní, Implementace, Wrapppery
Kontejnery III. – Algoritmy

Kontejnery (v Javě obvykle nazývané "kolekce") jsou datové objekty, které nám umožňují snadno a efektivně spravovat variabilní hromadná data (i další objekty). Podle způsobů uložení dat a práce s nimi rozlišujeme různé druhy kontejnerů a při jejich použití si vždy vybereme ten, který se pro daný případ nejlépe hodí v konkrétním jazyce. Základními operacemi dle typu kontejneru je iterace, porovnávání, přidávání a mazání.

Zvolený jazyk: Java

- **Pole** – nejzákladnější struktura vhodná zejména pro primitivní hodnoty, které se nemusí nahrazovat objektem (známe předem počet objektů ve skupině). Nevýhodou je jeho dynamické plnění (*nutné definovat jeho rozsah – časová náročnost*). Pole mohou být vícerozměrné.
- **Kolekce** – struktura vhodná zejména pro práci s objekty, kdy se „nemusíme“ starat o velikost kolekce. Nové objekty lze vkládat bez nutnosti realokace. Vhodné při vytváření předem neznámého počtu objektů.

The Collection Framework (CF) – systém rozhraní a tříd, které bezprostředně souvisí s kolekcemi. Naprostou většinu z nich najdeme v balíku java.util (v Javě 5.0 - tj. od JDK 1.5.0 došlo v CF k významným změnám).

Související pojmy:

- **Iterátor** – prostředek zajišťující sekvenční přístup k datům, pracující krokově, v každém dalším kroku poskytne přístup k dalšímu prvku. Rychlé právě v případě sekvenčního procházení. *Zajišťuje možnost procházení prvků bez znalosti jejich implementace*.
    - nezáleží na pořadí, nebo je pořadí dané vlastnostmi kolekce
    - metoda *iterator()* daného kontejneru
    - metoda *next()* - vrátí další prvek v kontejneru
    - metoda *hasNext()* - je-li v kontejneru další prvek
    - metoda *remove()* - vyjmutí posledního objektu vráceného iterátorem
- **Porovnatelnost** – důležitá vlastnost, kterou potřebujeme pro uložení do některých kontejnerů. Datové objekty mohou mít porovnatelnost implementovánu prakticky libovolným způsobem, třída implementuje rozhraní Comparable (obsahuje jedinou metodu `compareTo()`).
    - **Vector** – nejzákladnější implementace = „pole“ s proměnnou velikostí indexace od 0, v případě uložení primitivního datového typu nutné zapouzdření. Třída ještě před vznikem CF. Chování třídy `Vector` je prakticky shodné s třídou `ArrayList` s drobnými rozdíly. `Vector` má některé metody navíc (např. hledání od určitého indexu), lze mu stanovit přírůstek kapacity a je synchronizovaný (viz dále). V běžných případech dnes nemáme důvod používat `Vector` namísto třídy `ArrayList`.
    - **HashTable** – implementuje hašovací tabulku mapující klíče na hodnoty (obojí obecně typu Object) -> polymorfní, každý objekt - klíč musí implementovat metody hashCode a equals

Kolekce – základní „operace“

add(object) – boolean *addAll(Collection)* *clear()* *contains(Object*) – boolean *containsAll(Collection)* – boolean *isEmpty()* - boolean *iterator()* - vrací iterátor *remove(Object)* *removeAll(Collection)* – boolean *retainAll(Collection)* – ponechá ty, které se schodují s argumentem *size()* *toArray()* - vrátí pole	addAll(kolekce,pozice) get(pozice) indexOf(objekt) lastIndexOf(objekt) remove(pozice) listIterator(), listIterator(pozice) set(pozice, objekt) subList(od,do)

Máme dvě základní množiny kontejnerů, každá z nich vychází z jednoho rozhraní:

- **Kolekce** – „normální kolekce“ - skupina samostatných prvků (rozhraní *Collection*)
  Vždy zahrnují:
  - Rozhraní: umožňují pracovat s kolekcí bez ohledu na detaily její implementace
  - Implementaci: konkrétní znovupoužitelné implementace rozhraní.
  - Algoritmy: metody, které provádějí užitečné výpočty typu vyhledávání, třídění s objekty kolekcí

  - **Množiny** (rozhraní `Set`) - každý prvek může být v kontejneru **pouze jednou** (jinak kontejner zabrání vložení). Není zde obdoba kontejneru *multiset* (možnost vícenásobné přítomnosti téhož prvku). Prvek obecně nemá určenu žádnou polohu v množině, na základě které by k němu bylo možné přistupovat. Zvláštním případem je podrozhraní `SortedSet`, což je seřazená množina.

    - objekty, vložené do kontejneru, musí mít definovat metodu *equals()* - umožní určit jedinečnost

    Implementace

    - *HashSet*
      - množina s hešovací tabulkou -V situaci, kdy je třeba uložit obrovské pole nějakých řetězců a jednotlivé řetězce velmi rychle vyhledávat. Sekvenční prohledávání díky své velké časové náročnosti není vhodné. Jednotlivé řetězce se rozdělilí do skupin, které si jsou podobné. K tomu jsou hešovací funkce, která dokážou jednotlivé řetězce rozptýlit do oněch políček.
      - Objekty musí definovat metodu *hashCode()* - vrací celé číslo (int) „co nejlépe“ charakterizující obsah objektu
    - *TreeSet*
      - setříděný kontejner typu množina
      - rozhraní *SortedSet*

        - metody *first()* - vrátí nejmenší prvek
        - *last()* - největší prvek
        - *subSet(fromElement, toElement)* – podmnožina od určeného do druhého určeného prvku
        - *headSet(toElement)* – podmnožina menších prvků, než *toElement*
        - *tailSet(fromElement)* – podmnožina větších prvků, než *fromElement*

  - **Seznamy** (rozhraní `List`) - prvek může být v kontejneru vícekrát a má určenu jednoznačnou polohu (index). Indexován od 0.
    - seřazení seznamu – *sort(list)*

- promíchání seznamu – *shuffle(list)*
- obrácení pořadí – *reverse(list)*
- hledání binárním dělením, kopírování seznamu, konverze kolekcí na pole atd.

Implementace

- *ArrayList*

```
ArrayList<Person> persons = new ArrayList();
```

  - seznam implementovaný pomocí pole
  - rychlý přímý přístup k prvkům
  - vkládání a odebírání je pomalé!
- *LinkedList*
  - vkládání a odstraňování prvků je optimalizované, stejně tak sekvenční procházení
  - přímý přístup k prvkům je relativně pomalý
  - navíc obsahuje metody jako *addFirst(), addLast(), getFirst(), getLast(), removeFirst(), removeLast()*
  - nejsou definovány v rozhraní
    - umožňují zřetězený seznam použít jako zásobník, frontu či oboustrannou frontu

- **Mapa – „**asociativní kolekce (pole)“ - (mapované = obsahující dvojice klíč-hodnota), rozhraní *Map*. Lze rovněž rozšiřovat do více rozměrů => vytvoří se mapa, jejímiž hodnotami jsou také mapy. Objekty tedy nevyhledáváme podle čísla (indexu), ale pomocí jiného objektu - klíče.
Každý člen této dvojice může být libovolného referenčního typu (rozhraní, objektové třídy a pole, přistupujeme v zásadě pouze pomocí referencí). Neobsahuje duplicitní prvky.

  - metoda *put(klíč, hodnota)* vloží nejen hodnotu, ale zároveň její klíč
  - metoda *get(klíč)* – vrátí objekt odpovídající danému klíč

  Implementace

  - *HashMap*
    - založeno na hešovací tabulce
    - rychlé vkládání a vyhledávání položek
  - *TreeMap*
    - oproti *HashMap* přináší seřazení klíčů
    - výsledky se zobrazují setříděné

Nevýhody kontejnerů:

- Neznámý typ
  - kontejner obsahuje odkazy na objekty obecného typu *Object* (kořenová třída všech objektů)
  - neexistuje žádné omezení pro to, co se dá do kontejneru uložit (ani v případě, kdyby si to člověk výslovně přál.
    Od verze Javy 5.0 (JDK 1.5.x): `Vector<String> vector = new Vector<String>();`
  - je-li třeba objekt uložený v kontejneru použít, je nutné přetypovat na správný typ. Primitivní prvky je třeba vkládat jako zapouzdřující referenční typ.

Problematika perzistentního (trvalého) ukládání dat ve vybraném programovacím jazyce.

*"Data s dobou života překračující běh aplikačního programu i vypnutí počítače. Data mohou být uložena v jednoduchém plochém **souboru**, nebo mohou být uložena v nějakém typu **databáze**."*

Další zdroje vhodné pro prostudování:
I/O operace I. – Standardní IO API
I/O operace II. – Pokročilejší využití IO API
JDBC – Java Database Connectivity (eng.)
Relační databáze – Relační databáze
ORM – Objektové-relační mapování (eng.)

Zpracované otázky z minulých ročníků (viz. Podklady.zip)

Podle přístupu k datům můžeme persistentní data rozdělit do kategorií:

- **částečně perzistentní:** datové úložiště obsahuje **jen** aktuální verzi dat.
- **perzistentní:** datové úložiště obsahuje aktuální verzi dat a verzi dat před poslední změnou (v případě, že během transakce dojde k chybě, je použita funkce *rollback*, která vrátí DB do původního stavu, před začátkem transakce).
- **plně perzistentní:** datové úložiště obsahuje aktuální verzi dat + všechny verze dat.

Zvolený jazyk: Java (možnost práce s daty je ovlivněna vybraným jazykem a úložištěm dat)

Možnosti ukládání dat

- **Standardní IO API** (standardní vstupní/výstupní operace) uložené v balíku *java.io* (i další třídy jako **File** pro práci se soubory/adresáři či **StreamTokenizer**/**StringTokenizer** = rozdělení obsahu na úseky určené zadanými oddělovači).
  Každá z abstraktních tříd má specifického potomka (abstraktní třídu), která je předkem filtrů/dekorátorů = převádějí primitivní I/O stream na „pokročilejší" stream, např.:
  - *DataInputStream*/*DataOutputStream* (pro primitivní datové typy)
  - *BufferedInputStream* /*BufferedOutputStream* (primitivní stream na bufferovaný stream)

  - **Soubory** (soubor s vlastní strukturou) = posloupnost dat uložená na externí paměťová média (disk, disketa, …). Používají se také pro komunikaci s periferními zařízeními.
    > Výjimky **IOException** a **FileNotFoundEcxeption**
    - **Textové** – čitelná podoba dat, posloupnost znaků uspořádaných do řádků.
      - čtení po znacích > abstraktní třída *FileReader*
        `FileReader soubor = new FileReader(„soubor.txt");` (čtení po znacích soubor.read();)
        `BufferedReader br = new BufferedReader( soubor );` čtení po řádcích br.readLine();
      - výstup po znacích > abstraktní třída *FileWriter*
        `FileWriter soubor = new FileWriter(„soubor.txt");` (zápis po znacích soubor.write();)
        `PrintWriter pw = new PrintWriter( soubor );` (po řádcích pw.println();)
    - **Binární** – číslicová podoba dat, posloupnost *bajtů*.
      - čtení po bytech > abstraktní třída *FileInputStream*
        `FileInputStream soubor = new FileInputStream(„soubor.dat")`
      - výstup po bytech > abstraktní třída *FileOutputStream*
        `FileOutputStream soubor = new FileOutputStream(„soubor.dat")`
  - **Streamy** (**proudy dat**) – implementují (jako objekty) mimo jiné přístup k datům uloženým v souborech, obecné použití (komunikace po síti), sekvenční přístup
    - **Vstupní a výstupní** (binární a znakové[1])
      - čtení po bytech > abstraktní třída *InputStream*
      - výstup po bytech > abstraktní třída *OutputStream*

[1] využití lokálních znakových sad a kódování

čtení a zápis 1 bytu > **read**(), **write**(B) (readBoolean(); readChar();)
čtení a zápis pole bytů > **read**(poleB), **write**(poleB)
čtení a zápis úseku z pole bytů > **read**(poleb,od,kolik), **write**(poleb,od,kolik)
zavření > **close**()

- čtení po znacích > abstraktní třída *Reader*
>Třída *Reader* by se měla používat vždy, když je potřeba číst text. Garantuje správnou obsluhu znakových sad a převod textu do vnitřního kódování Javy (což je *Unicode*).
- výstup po znacích > abstraktní třída *Writer*
- **ObjectInputStream** - umožní číst celé objekty, kontejnery atd. (nutná *serializace*)
- **ObjectOutputStream** oos = new ObjectOutputStream(soubor); (zápis objektů oos.writeObject(obj); (nutná *serializace*)

- **Databáze**
  Pro přístup se používá JDBC rozhraní (použití ovladače pro překlad požadavků do nativního volání daného úložiště – databáze, Excel soubory, Access atp.).
  JDBC specifikace rozpoznává čtyři typy JDBC ovladačů:

1. **JDBC-ODBC bridge –** ovladač přistupuje k databázi přes nativní ovladač ODBC, který **musí** být na daném stroji nainstalován.
2. **Native-API driver –** ovladač přistupuje k databázi přes služby nativního ovladače dané databáze (využívá jeho metody, které jsou vytvořeny v C, C++), **musí** být na daném stroji nainstalován. Jistá podoba s typem 1.
3. **Network-Protocol driver –** založen čistě na Javě a JDBC, který konvertuje svoji komunikaci do síťového protokolu a spojuje se s centrálním serverem (Network Server), který poskytuje připojení k databázi (obvykle s „poolem“ připojení). Tento server konvertuje síťový protokol, kterým komunikuje s klienty, do databázově specifického protokolu, jemuž již databáze rozumí. Takový model je vysoce efektivní, a to jak s ohledem na možnost „poolingu“ připojení a tím i zrychlení dotazování a práce s databází, tak i možnost připojení k sadě heterogenních databázových systémů.
4. **Native-Protocol driver –** Tento ovladač komunikuje s databázovým serverem přímo přes síťový protokol.

JDBC API

- používá pro práci s DB několik základních tříd a rozhraní, jsou to:

- **DriverManager:** Třída, která má na starost nahrávání a registraci dostupných ovladačů. Je to mezivrstva mezi JDBC ovladačem a programovým kódem.
- **Connection:** Toto rozhraní reprezentuje připojení k DB. Je to centrální rozhraní pro správu databázového spojení. Umožňuje přípravu a vykonání db dotazů, transakční zpracování.
- **Statement:** je rozhraní, které je zodpovědné za posílání SQL příkazů databázi. Rozhraní implementuje tyto metody:
  - **executeQuery(sql):** provadí SQL dotaz, který zpravidla vrací *ResultSet* což je seznam databázových vět.
  - **ExecuteUpdate(sql):** používá se v případě, že chceme modifikovat data v DB (INSERT, UPDATE, DELETE).
- **ResultSet:** Rozhraní představuje výsledky databázového dotazu(tabulku). Zpravidla je to několik datových vět. Metody tohoto rozhraní můžeme rozdělit do těchto kategorií:
  - **Navigační metody:** Tyto metody umožňují pohyb v množině výsledků pomocí **funkcí** *first(), next(), last(), previous()*.
  - **Gettry:** *ResultSet* rozhraní poskytuje metody (getBoolean, getString, atd.), které vrací hodnoty jednotlivých sloupců v aktuální řádce.
  - **Modifikační metody:** Tyto metody umožňují modifikovat (měnit, vkládat, rušit ..) data v ResultSet a následně provést modifikaci v DB. Jsou to např. insertRow(), updateString, atd.

Ukázka:

```
Class.forName("org.hsqldb.jdbcDriver"); 1: Zavedení ovladače
String url = "jdbc:hsqldb:data/tutorial"; 2: Vytvoření spojení s DB
conn = DriverManager.getConnection(url, "sa", ""); 3 : Přihlášení k DB
```

```
st = conn.createStatement(); 4: Vytvoření instance Statement pro dotazování
String sqlQuery = "SELECT uid, name, duration from EVENTS"; 5 : Vytváření dotazu.
ResultSet rs = stmt.executeQuery(sqlQuery); 6: Zpracování ResultSet.
rs.first();
while (rs.next()) {
String name = rs.getString(2);
Timestamp hireDate = rs.getTimestamp(5);
System.out.println("Name: " + name + " Hire Date: " + hireDate);
}
```

Nevýhody JDBC

- **Problém relační model + objektový přístup:** Rozhraní *ResultSet* vrací data jako tabulku, ne jako objekty (OOP v Javě). Vhodnější řešení by bylo vrácení řádků dat jako objekty, které reprezentují jednotlivé entity v tabulkách (člověk, faktura). Protože *ResultSet* toto neumožňuje, programátor je velmi často nucen si takové třídy vytvořit a zde dochází k obrovské ztrátě času.
- **Vyjímky:** Při každé operaci je potřeba odchytat mnoho vyjímek což značně znepřehledňuje kód.

Objektově relační mapování

Objektově relační mapování slouží k tomu, aby bylo možné snadno používat relační databáze v prostředí objektově orientovaných programovacích jazyků - Javě. Vzhledem k tomu, že objektově orientovaný návrh dat není jednoznačně převoditelný na relační databáze a opačně, používají se různé formy mapování. Mapování má za účel načítat data z relační databáze a naplnit jimi příslušné datové položky objektů, případně naopak datové položky objektů ukládat do databáze. Snahou ORM je co nejlepším využití obou zmíněných technologií – tj. objekty by měly reprezentovat objekty reálného světa, jak to požadují principy OOP, na straně databáze bychom zase měli využít všech možností relačních databází – indexy, pohledy, primární klíče, atd. V současnosti je snad nejpoužívanějším nástrojem pro ORM produkt od firmy JBOSS Hibernate.

Objektové databáze

Objektové databáze umožňují skladování dat s libovolnou strukturou. Programátor či kdokoliv jiný se při použití objektové databáze nemusí vůbec starat o mapování objektových struktur do dvourozměrné tabulky. Každý persistentní objekt v objektové databázi má svoji vlastní identitu, je tedy jednoznačně odlišitelný od libovolného jiného objektu nezávisle na hodnotách svých vlastností – odpadají tedy problémy s tvorbou primárních klíčů. Kvalitní objektová databáze podporuje všechny vlastnosti nutné pro práci s objekty – *zapouzdření*, *jednoznačnou identifikaci*, *reference mezi objekty* a další (*dědičnost*, *polymorfismus* atp.).
Např.: db4o, Ozone, ObjectDB.
Existují i hybridní objektové databáze, které implementují pouze některé z výše zmíněných vlastností (**N**otes **S**torage **F**ile u Lotus Notes ☺).

Webové aplikace – principy, nástroje. Vícevrstvé aplikace. Zabezpečení aplikace.

1. Principy a nástroje

Webová aplikace:

- aplikace poskytovaná uživatelům z webového serveru přes počítačovou síť Internet nebo její vnitropodnikovou obdobu intranet
- aplikace založena na systému komunikace klienta (prohlížeč klientského počítače) a aplikace běžící na webovém aplikačním serveru
- komunikace zpravidla probíhá přes protokol HTTP/HTTPS
- klient vysílá požadavek (HTTP REQUEST), server zpracuje adekvátním způsobem a posílá nazpátek odpověď (HTTP RESPONSE)
- zpracování požadavku na serveru má na starosti mechanismus, který je příslušným způsobem vyhodnotí a předá dále do aplikace. Zasílá nazpátek odpovědi klientovi tak, že generuje výsledné webové stránky
- protokol je bezstavový, nemá stálé spojení s kleinty a nemůže je proto identifikovat - velké komplikace pro webové aplikace, které vyžadují stavovou informaci, např. nákupní košík; řešení:
    i. přenášení údajů v URL a skrytých polí formuláře
    ii. cookies
    iii. session proměnné
- populární především pro všudypřítomnost webového prohlížeče jako klienta
- schopnost aktualizovat a spravovat webové aplikace bez nutnosti šířit a instalovat software na potenciálně tisíce uživatelských počítačů je hlavním důvodem jejich oblíbenosti
- statické stránky (do ½ 90. let)
    i. text, obrázky
    ii. akademická sféra, odborná veřejnost
    iii. uživatelé mohou obsah webových stránek ovlivnit jen minimálně
- dynamické stránky (od ½ 90. let)
    i. multimediální obsah
    ii. přístupné pro nejširší veřejnost
    iii. interaktivita s uživatelem

- aplikační (business) logika a datová část je umístěna na straně serveru
- webový aplikační server je kontejner, umožňující spouštět skripty příslušného jazyka
- používané technologie na straně klienta – XHTML, CSS, JavaScript, VB.Script atd.
- používané technologie na straně serveru
    i. interpretované – ASP, PHP, Perl, Java… (skriptovací jazyk se nepřekládá do strojového kódu, ale je přímo vykonáván interpretorem)
    ii. kompilované – ASP.NET (aplikace jsou předkompilovány do jednoho či několika málo DLL souborů)
- Common Gateway Interface (CGI)
    i. je protokol pro propojení externích aplikací s webovým serverem
    ii. definuje jak má webový server komunikovat s ostatními aplikacemi (CGI-skripty)
    iii. CGI skript je externí program, který je na požadavek od uživatele spuštěný WWW serverem jako samostatný proces. CGI skripty přebírají data zadaná uživatelem, zpracovávají je a jako výsledek vytvářejí většinou HTML stránky. Tyto dynamicky vytvořené stránky pak WWW server posílá zpět klientovi.

	Linux	Windows	Apache	IIS	Lighttpd	PHP	ASP.NET	Perl	Python	Java	MySQL	SQL Server	Memcached
Flickr	x		x			x		x		x	x		x
YouTube	x		x		x				x		x		
PlentyOfFish		x		x			x					?	
Digg	x		x			x					x		x
TypePad	x		x					x			x		x
LiveJournal	x		x					x			x		x
Friendster	x		x			x		x			x		
MySpace		x		x			x					x	
Wikipedia	x		x		x	x					x		x

Vícevrstvé aplikace

První softwarové aplikace využívaly jednoduchou monolitickou architekturu, která se snadno implementovala a neměla náročné požadavky na technické vybavení. V tomto případě byla databázová, aplikační i prezentační logika soustředěna v jednom monolitickém bloku. Koncept vícevrstvých architektur datuje své počátky do období rozvoje lokálních sítí. Programy využívající síť pro přístup k informacím v databázích jsou podle tohoto konceptu rozděleny do vrstev podle funkce. Uspořádání architektury může mít vrstvy dvě nebo tři.

Model třívrstvé architektury je přímou evolucí, z dnešního pohledu již koncepčně zastaralé, dvouvrstvé architektury. Dvouvrstvá architektura vychází z vrstvy klientské a vrstvy datové. Klient obsahuje většinu aplikační logiky, se kterou pracuje přímo nad datovým zdrojem. Nevýhody tohoto modelu se ukázaly při vzrůstající komplexnosti klientských aplikací. Se složitostí aplikací vzrůstaly výkonové nároky na klientské počítače, s masovým rozšířením aplikací vrůstaly nároky na sdílení zdrojů, omezení datových přenosů apod.

Řešení se našlo v podobě přidání třetí - střední vrstvy. Díky nově definované aplikační vrstvě bylo možné aplikační logiku, která do té doby ležela na klientu, přesunout na aplikační server. Díky tomuto tahu se klientům značně ulevilo, protože veškerý výpočetní výkon byl přesunut na výkonné servery. Díky přesunutí aplikační logiky na jedno místo bylo možné dosáhnout jejího sdílení a lepší správy a dostupnosti. Významnou měrou se podařilo redukovat datový přenos, jehož těžiště se přesunulo na trasu mezi aplikačním serverem a datovým zdrojem.

2. Zabezpečení aplikace

Zabezpečení přístupu

- pro identifikaci uživatele přistupujícího k aplikaci se nejčastěji používá zadání jména a hesla. Tyto údaje by měli být navíc před odesláním nebo uložením šifrovány (např. MD5, SHA1), aby je nebylo možné odchytit při posílání např. po síti nebo získat z úložiště na disku. Po zašifrování je manipulováno pouze s tzv. otiskem hesla. Šifrovací funkce je jednosměrná, určení původního hesla z jeho otisku je nemožné (téměř). Pro zvýšení odolnosti proti slovníkovým útokům se přidává k šifrovanému heslu náhodný řetězec (hash = MD5 (heslo . retezec))
- mezi další možnosti patří ověření uživatele nebo ověření totožnosti serveru pomocí certifikátů. Server posílá klientovi certifikát. Certifikát spojuje dohromady počítač s reálně existující osobou (fyzickou či právnickou). Certifikát vydává certifikační autorita (CA) – ta by měla ověřit skutečnou identitu žadatele o certifikát. Prohlížeč automaticky věří certifikátům od CA, které zná (umí ověřit podpis na certifikátu).
- na úrovni databázové vrstvy lze pro uživatele aplikace nastavit příslušnou úroveň oprávnění – přístup ke konkrétní databázi nebo tabulce. S vytvářením oprávnění se váže princip nejnižšího uživatelského práva: Uživatel (nebo proces) by měl mít nejnižší úroveň práv, která mu umožňuje provést patřičný úkol.

Zabezpečení dat

- validace dat – kontrola správnosti, úplnosti a konzistence vkládaných dat. Lze ji provádět na úrovni klienta (JavaScript), na úrovni serveru (PHP, Java Servlet), na úrovni databáze (SQL omezení), nebo kombinací těchto technik.
- šifrování přenášených dat – zvýšení bezpečnosti pomocí šifrování přenášených dat je jednou ze základních vlastností aplikací jako jsou online bankovnictví, business komunikace apod. Pro zabezpečení přenosu v rámci internetu mezi serverem s webovou prezentací a prohlížečem (uživatelem) se používají protokoly SSL (Secure Sockets Layer) nebo TLS (Trasport Layer Security), přičemž SSL je předchůdce TSL. SSL je protokol, resp. vrstva vložená mezi vrstvu transportní (např. TCP/IP) a aplikační (např. HTTP), která poskytuje zabezpečení komunikace šifrováním (a autentizací) komunikujících stran. Každá z komunikujících stran má dvojici šifrovacích klíčů - veřejný a soukromý. Veřejný klíč je možné zveřejnit a pokud tímto klíčem kdokoliv zašifruje nějakou zprávu, je zajištěno, že ji bude moci rozšifrovat jen majitel použitého veřejného klíče svým soukromým klíčem. Ustavení SSL spojení (SSL handshake) pak probíhá následovně:
    i. klient pošle serveru požadavek na SSL spojení, spolu s různými doplňujícími informacemi (verze SSL, nastavení šifrování atd.),
    ii. server pošle klientovi odpověď na jeho požadavek, která obsahuje stejný typ informací a hlavně certifikát serveru,
    iii. podle přijatého certifikátu si klient ověří autentičnost serveru. Certifikát také obsahuje veřejný klíč serveru,
    iv. na základě dosud obdržených informací vygeneruje klient základ šifrovacího klíče, kterým se bude kódovat následná komunikace. Ten zakóduje veřejným klíčem serveru a pošle mu ho.,
    v. server použije svůj soukromý klíč k rozšifrování základu šifrovacího klíče. Z tohoto základu vygenerují jak server, tak klient hlavní šifrovací klíč,
    vi. klient a server si navzájem potvrdí, že od teď bude jejich komunikace šifrovaná tímto klíčem. Fáze „handshake“ tímto končí,
    vii. je ustaveno zabezpečené spojení šifrované vygenerovaným šifrovacím klíčem.
    viii. aplikace od teď dál komunikují přes šifrované spojení. Například POST požadavek na server se do této doby neodešle.

Algoritmy pracující s grafy. Prohledávání grafů do hloubky a do šířky, využití prohledávání grafů v dalších úlohách.

Slovo **graf** se používá v různých významech. V matematice také existuje pojem grafu, který nemá takměř nic společné se známými grafy funkcí, až na název, který je odvozený z řeckého slova "grafein", co znamená "psát".
Pojem "graf" vykrystalizoval jako matematický objekt (univerzální zobrazovací technika), který se skládá z prvků dvojakého druhu – z vrcholů a hran.

Obyčejný graf G rozumíme uspořádanou dvojicí (V,E), tzn. **G = (V,E)**, V je libovolná množina vrcholů a E je podmnožina hran.

Sled v grafu G je konečná posloupnost S=(v0,e1,v1,e2,...vn-1,en,vn), v které se střídají vrcholy a hrany, a která se začíná a končí ve vrchole, přičemž pro všechny i=1,2,3,....n je ei (vi-1, vi). V sledu se hrany a vrcholy můžou opakovat. Číslo n nazýváme délkou sledu S. Sled nazýváme uzavřený, když jeho začátek a konec jsou totožné. V opačném případě ho nazýváme otevřený.

Tah je takový sled, v kterém se žádna hrana nevyskytuje dvakrát.
Otevřený tah, v kterém jsou všechny vrcholy navzájem různé, se nazývá cesta. Počet hran v cestě se nazývá délka cesty.

Strom je souvislý graf, neobsahující kružnici (acyklický souvislý graf, tj. uzavřený sled).

Podgraf G´ = (V´,E´) je indukovaný podgraf grafu G , když G´ obsahuje všechny hrany spojující vrcholy z V´ v grafu G. Podgraf G´ indukovaný množinou vrcholů V´ v grafu G

Stupeň vrcholu v je počet hran incidentních s vrcholem v, označení: degG v (deg v)
δ(G) - minimální, Δ(G) – maximální, izolovaný vrchol v ... deg v = 0

Úplný (kompletní) graf - graf, ve kterém jsou každé dva vrcholy spojené hranou.
Označení Kn, n - počet vrcholů

Regulární graf - takový graf, jehož všechny vrcholy mají stejný stupeň. Regulární graf s vrcholy, které mají stupeň k se nazývá k-regulární.

Věta (princip sudosti):
Když má graf G má m hrán, tak součet stupňů všech jeho vrcholů se rovná 2m:

$$\sum_{v \in V} \deg_G v = 2m$$

Důsledek:
Počet vrcholů lichého stupně v grafu je sudý.

Algoritmy k nalezení minimální kostry

Jarníkův algoritmus

Algoritmus hledající minimální kostru ohodnoceného grafu. Najde takovou podmnožinu hran grafu, která tvoří strom obsahující všechny vrcholy původního grafu a součet ohodnocení hran z této množiny je minimální. Algoritmus začíná s jedním vrcholem a postupně přidává další nejbližší vrcholy(nejníže ohodnocená hrana) a tím zvětšuje velikost stromu do té doby, než obsahuje všechny vrcholy.

Kruskalův algoritmus

Algoritmus hledající minimální kostru ohodnoceného grafu jehož hrany mají nezáporné ohodnocení. Najde takovou podmnožinu hran grafu, která tvoří strom obsahující všechny vrcholy původního grafu a součet ohodnocení hran z této množiny je minimální. Vybírají se hrany s nejnižším ohodnocení, tak aby nevznikla kružnice.

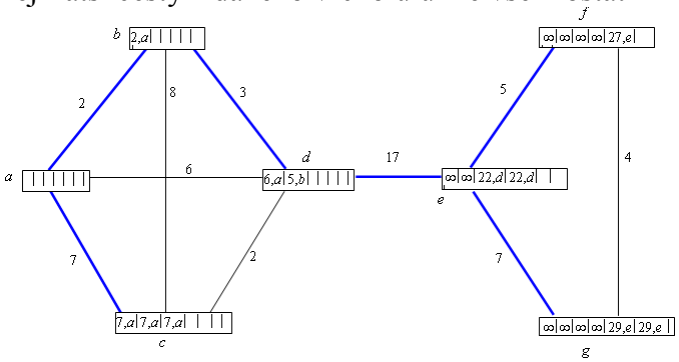
Borůvkův algoritmus

Poprvé byl publikován roku 1926 Otakarem Borůvkou jako metoda pro konstrukci efektivní elektrické sítě na Moravě. Algoritmus pracuje tak, že postupně spojuje komponenty souvislosti (na počátku je každý vrchol komponentou souvislosti) do větších a větších celků, až zůstane jen jediný, a to je hledaná minimální kostra. V každé fázi vybere pro každou komponentu souvislosti hranu s co nejnižší cenou, která směřuje do jiné komponenty souvislosti a tu přidá do kostry. V každé fázi se počet komponent souvislosti sníží nejméně dvakrát, počet fází bude tedy maximálně log2N, kde N je počet vrcholů grafu.

Dijkstrův algoritmus

Sloužící k nalezení nejkratší cesty v grafu. Je konečný (pro jakýkoliv konečný vstup algoritmus skončí), protože v každém průchodu cyklu se do množiny navštívených uzlů přidá právě jeden uzel, průchodů cyklem je tedy nejvýše tolik, kolik má graf vrcholů. Funguje nad hranově kladně ohodnoceným grafem (neohodnocený graf lze však na ohodnocený snadno převést).

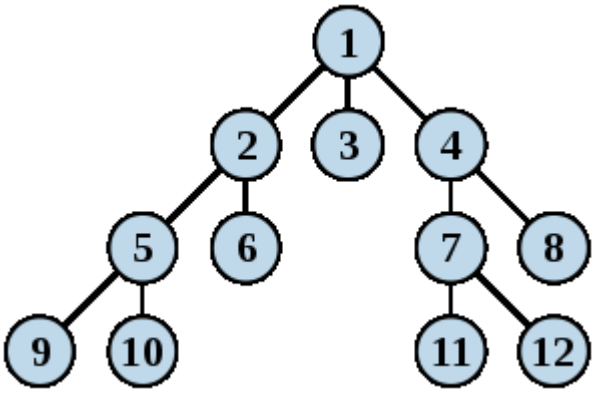
Na počátku nechť je každá hrana grafu G neobarvená. Za modrý strom považujme počáteční vrchol u, od kterého hledáme nejkratší cesty k ostatním vrcholům.
Pro každý sousední vrchol v vrcholu u provedeme dv := délka hrany (u, v) a Vv := u.
V každém z (n - 1) kroků vybereme z hran, které se modrého stromu dotýkají (tj. mají jeden koncový vrchol x v modrém stromu a druhý koncový vrchol v nikoli), hranu (x, v) pro kterou platí, že hodnota dv je minimální (existuje-li jich více, zvolíme libovolnou z nich), obarvíme ji modře a pro všechny sousední vrcholy w vrcholu v provedeme v případě, že platí dv + délka hrany (v, w) < dw, příkazy Vw := v a dw := dv + délka hrany (v, w).
Algoritmus končí získáním modrého stromu obsahujícího všechny vrcholy grafu G, tj. určením nejkratší cesty z daného vrcholu u ke všem ostatním vrcholům.



Algoritmy prohledávání do hloubky a do šířky

Prohledávání do šířky

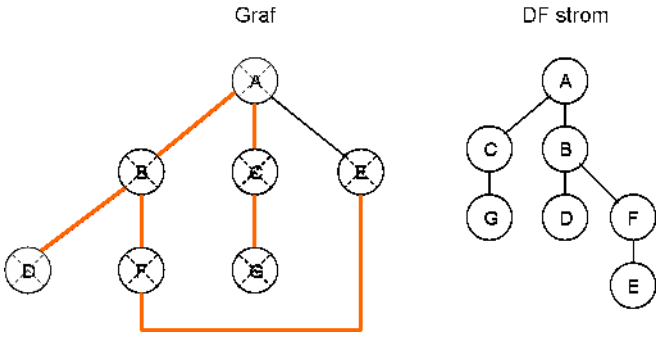
Postupně prochází všechny vrcholy v dané komponentě souvislosti. Algoritmus nejprve projde všechny sousedy startovního vrcholu, poté sousedy sousedů atd. až projde celou komponentu souvislosti. Prohledávání grafu do šířky se realizuje pomocí fronty (FIFO). Začneme od libovolného vrcholu r. Výstup algoritmu – kostra grafu.



Pořadí v jakém je přistupováno k vrcholům

Prohledávání do hloubky

Pracuje tak, že vždy expanduje prvního následníka každého vrcholu, pokud jej ještě nenavštívil. Pokud narazí na vrchol, z nějž už nelze dále pokračovat (nemá žádné následníky nebo byli všichni navštíveni), vrací se zpět backtrackingem. Prohledávání grafu do hloubky se realizuje pomocí zásobníku (LIFO). Začneme od libovolného vrcholu r. Výstup algoritmu - kostra grafu



Binární strom

Každý vrchol má nejvýše dva následníky, levý a pravý syn

procedure PREORDER (*T, v*)

- probrání vrcholu v
- jestliže existuje levý syn v_L vrcholu v, pak PREORDER (T, v_L)
- jestliže existuje pravý syn v_P vrcholu v, pak PREORDER (T, v_P)
- návrat k přímému předchůdci (otci) vrcholu v

procedure INORDER (*T, v*)

- jestliže existuje levý syn v_L vrcholu v, pak INORDER (T, v_L)
- probrání vrcholu v
- jestliže existuje pravý syn v_P vrcholu v, pak INORDER (T, v_P)
- návrat k přímému předchůdci (otci) vrcholu v

procedure POSTORDER (*T, v*)

- jestliže existuje levý syn v_L vrcholu v, pak POSTORDER (T, v_L)
- jestliže existuje pravý syn v_P vrcholu v, pak POSTORDER (T, v_P)
- probrání vrcholu v
- návrat k přímému předchůdci (otci) vrcholu v

Využití prohledávání grafů v dalších úlohách

- Určení souvislosti daného grafu – musí být zpracovány všechny vrcholy ze zásobníku

- Určení počtu komponent daného grafu – do šířky, pokud nejsou zpracovány všechny vrcholy, navýšíme počitadlo o 1 a pokračujeme dalším vrcholem dle abecedy
- Určení, zda daná hrana je či není most daného grafu – prohledávání do hloubky, pokud jsou oba vrcholy odebrané hrany zpracovány, tak hrana není most
- Určení, zda dva dané vrcholy leží v téže komponentě – po prohledání musí být ve stejné množině
- Určení nejkratší cesty (co do počtu hran) a její délky – do šířky, délka cesty je úroveň, ve které se nachází cílový vrchol
- Určení, zda je graf bipartitní – do šířky, pokud není propojení větví na stejné úrovni
- Zda existuje kružnice obsahující daný vrchol – do šířky, pokud existuje propojení mezi dvěma podstromy, pak existuje kružnice obsahující vrchol