

Obsah

I Počítačová grafika a analýza obrazu	4
1 Systémy barev v počítačové grafice, nelinearita grafického výstupu (gamma korekce), kompozice rastrových obrazů (alfa kanál), HDR.	5
2 Afinní a projektivní prostor. Afinní a projektivní transformace a jejich matematický zápis. Modelovací a zobrazovací transformace v počítačové grafice.	9
3 Metody získávání fotorealistických obrazů, rekurzivní sledování paprsku, radiometrie, zobrazovací rovnice, Monte Carlo přístupy ve výpočtu osvětlení, urychlovací metody.	15
4 Standardní zobrazovací řetězec a realizace jeho jednotlivých kroků, modely osvětlení a stínovací algoritmy, řešení viditelnosti, možnosti výpočtu globálního osvětlení v reálném čase, stručná charakteristika standardu OpenGL.	20
5 Komprese obrazu a videa, principy úprav obrazu v prostorové a frekvenční doméně.	27
6 Základní metody úpravy a segmentace obrazu (filtrace, prahování, hrany, oblasti, rohy).	35
7 Základní metody rozpoznávání objektů, příznakové rozpoznávání. Univerzální příznaky pro rozpoznávání (např. HOG), trénovací klasifikátory (např. SVM).	39
8 Hluboké neuronové sítě (např. konvoluční, popis jednotlivých vrstev).	49
9 Rekonstrukce 3D objektů z 2D obrazů (základní principy).	51
II Matematické základy informatiky	53
10 Konečné automaty, regulární výrazy, uzávěrové vlastnosti třídy regulárních jazyků.	54
11 Bezkontextové gramatiky a jazyky. Zásobníkové automaty, jejich vztah k bezkontextovým gramatikám.	57
12 Matematické modely algoritmů -Turingovy stroje a stroje RAM. Složitost algoritmu, asymptotické odhady. Algoritmicky nerozhodnutelné problémy.	60
13 Třídy složitosti problémů. Třída PTIME a NPTIME, NP-úplné problémy.	65
14 Jazyk predikátové logiky prvního řádu. Práce s kvantifikátory a ekvivalentní transformace formulí.	68
15 Pojem relace, operace s relacemi, vlastnosti relací. Typy binárních relací. Relace ekvivalence a relace uspořádání.	71
16 Pojem operace a obecný pojem algebra. Algebry s jednou a dvěma binárními operacemi.	73
17 FCA – formální kontext, formální koncept, konceptuální svazy.	75
18 Asociační pravidla, hledání často se opakujících množin položek.	78
19 Metrické a topologické prostory – metriky a podobnosti.	81
20 Shlukování.	84

21 Náhodná veličina. Základní typy náhodných veličin. Funkce určující rozdělení náhodných veličin.	87
22 Vybraná rozdělení diskrétní a spojité náhodné veličiny - binomické, hypergeometrické, negativně binomické, Poissonovo, exponenciální, Weibullovo, normální rozdělení.	93
23 Popisná statistika. Číselné charakteristiky a vizualizace kategoriálních a kvantitativních proměnných.	99
24 Metody statistické indukce. Intervalové odhadování. Princip testování hypotéz.	106
III Softwarové inženýrství	111
25 Softwarový proces. Jeho definice, modely a úrovně vyspělosti.	112
26 Vymezení fáze „sběr a analýza požadavků“. Diagramy UML využité v dané fázi.	118
27 Vymezení fáze „Návrh“. Diagramy UML využité v dané fázi. Návrhové vzory – členění, popis a příklady.	122
28 Objektově orientované paradigma. Pojmy třída, objekt, rozhraní. Základní vlastnosti objektu a vztah ke třídě. Základní vztahy mezi třídami a rozhraními. Třídní vs. instanční vlastnosti.	125
29 Mapování UML diagramů na zdrojový kód.	127
30 Správa paměti(v jazycích C/C++, Java , C#, Python), virtuální stroj, podpora paralelního zpracování a vlákna.	129
31 Zpracování chyb v moderních programovacích jazycích, princip datových proudů – pro vstup a výstup. Rozdíl mezi znakově a bytově orientovanými datovými proudy.	136
32 Jazyk UML – typy diagramů a jejich využití v rámci vývoje.	139
IV Databázové a informační systémy	141
33 Modelování databázových systémů, konceptuální modelování, datová analýza, funkční analýza; nástroje a modely.	142
34 Relační datový model, SQL; funkční závislosti, dekompozice a normální formy.	146
35 Transakce, zotavení, log, ACID, operace COMMIT a ROLLBACK; problémy souběhu, řízení souběhu: zamykání, úroveň izolacev SQL.	152
36 Procedurální rozšíření SQL, PL/SQL, T-SQL, triggery, funkce, procedury, kurzory, hromadné operace.	158
37 Základní fyzická implementace databázových systémů: tabulky a indexy; plán vykonávání dotazů.	162
38 Objektově-relační datový model a XML datový model: principy, dotazovací jazyky.	167
39 Datová vrstva informačního systému; existující API, rámce a implementace, bezpečnost; objektově-relační mapování.	170
40 Distribuované SŘBD, fragmentace a replikace.	173
V Počítače a sítě	175
41 Architektura univerzálních procesorů. Principy urychlování činnosti procesorů.	176
42 Základní vlastnosti monolitických počítačů a jejich typické integrované periférie. Možnosti použití.	183

43 Protokolová rodina TCP/IP.	186
44 Metody sdíleného přístupu ke společnému kanálu.	189
45 Problémy směrování v počítačových sítích. Adresování v IP, překlad adres (NAT).	192
46 Bezpečnost počítačových sítí s TCP/IP: útoky, paketové filtry, stavový firewall. Šifrování a autentizace, virtuální privátní sítě.	198

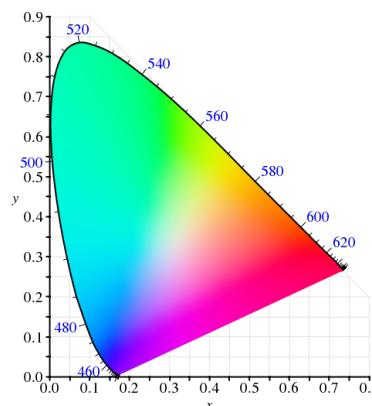
Část I

Počítačová grafika a analýza obrazu

1 Systémy barev v počítačové grafice, nelinearita grafického výstupu (gamma korekce), kompozice rastrových obrazů (alfa kanál), HDR.

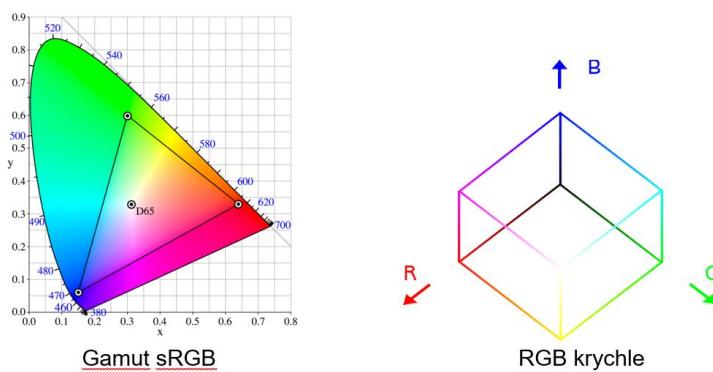
1.1 Systémy barev v PG

- Základem barevného prostoru je **barevný model**, který nám dává abstraktní matematický popis, jak lze barvy vyjádřit pomocí n-tic čísel, nejčastěji trojic.
- Mezi nejznámější barevné modely v dnešní době patří **RGB model**.
- Model RGB pracuje se třemi základními barvami: **červenou, zelenou a modrou**, z nichž se odvíjí i jeho název.
- Tyto barvy byly zvoleny na základně toho, jak **čípky v lidském oku** vnímají jednotlivé záření.
- Zároveň je RGB **aditivní barevný model**, což znamená, že se jednotlivé barevné složky **míchají** (nové barvy získáváme přidáváním větší intenzity jednotlivých složek) a výsledkem jsou další barevné odstíny, případně vyšší intenzita barvy.
- Když k tomuto modelu definujeme, jak mají být tyto n-tice interpretovány, dostáváme **barevný prostor** – je předem definovaná množina barev, kterou je schopno určité zařízení snímat, zobrazit nebo reprodukovat.
- Barevný prostor je tedy **definován rozsahem barev**, které dokáže zobrazit.
- Tomuto rozsahu se také říká **gamut**. Ten se zpravidla zobrazuje jako oblast v CIE 1931 chromatickém diagramu



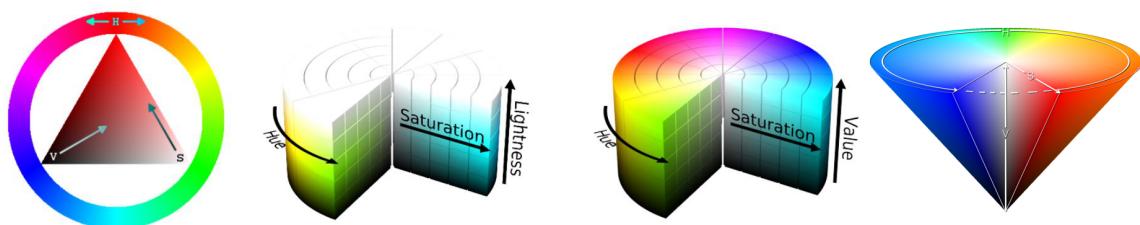
1.1.1 RGB

- Nejrozšířenější barevný prostor postavený na RGB barevném modelu je **sRGB** - standardní RGB.
- Jeho určení je pro zobrazování **na monitorech** nebo **kódování barev** na internetu.
- Pro všechny tři barevné složky má definované barvy v **chromatickém diagramu**, které vymezují jeho gamut.
- Každá barva, kterou tento prostor zobrazuje, je dána zastoupením jednotlivých barevných složek, buďto relativně (hodnoty jsou v rozmezí 0 - 1) nebo absolutně (konkrétní „bitové“ hodnoty, zpravidla 0 - 255, 24-bitů).
- RGB je možné zobrazit jako krychli.
- Často se přidává **Alpha kanál** pro průhlednost - **RGBA** (32-bitů).



1.1.2 HSV a HSL

- **Hue, Saturation, Value/Lightness** - barevný model, který nejvíce odpovídá lidskému vnímání barev.
- Barvy popisuje pomocí 3 hodnot, které však samy barvy nereprezentují:
 - **Hue** - barevný tón, převládající. Neboli **odstín** - barva **odražená** nebo **procházející** objektem. Měří se jako poloha na standardním barevném kole (0° až 360°). Obecně se odstín označuje názvem barvy. 0° - červená, 120° - zelená, 240° - modrá.
 - **Saturation** - sytost barvy, příměs jiné barvy. Někdy též chroma, síla nebo čistota barvy, představuje množství šedi v poměru k odstínu, měří se v procentech od 0% (šedá) do 100% (plně sytá barva). Na barevném kole vzrůstá sytost od středu k okrajům.
 - **Value** - hodnota jasu, množství bílého světla. Relativní světlost nebo tmavost barvy. Jas vyjadřuje **kolik světla barva odráží**, dalo by se také říct přidávání černé do základní barvy.
- Nejčastěji se tato reprezentace (popř. **HSL**) používají v grafických nástrojích jako komponenty pro výběr barvy, protože je mnohem intuitivnější než RGB.
- Vyberu si odstín, jak má být sytý a jasný a hotovo. Není třeba řešit jak smíchat 3 barevné složky, abych dostal to co chci.
- Dále se využívá často v případě detekce objektů, kdy hodnota HUE (odstín), je nezávislý na osvětlení scény. Problém však nastává u bílých a černých objektů (kdy HUE může být různé), ty lze na základě value a saturation mapovat do podobných barev (žlutá a černá).
- Mimo níže uvedené zobrazení **válcem**, lze také zobrazit **kuželem** a



1.1.3 CMY a CMYK

- Substraktivní barevné systémy (barvy se „odečítají“ od bílé, přidáváním jednotlivých složek až po černou), Cyan, Magenta, Yellow a Key (Black).
- Používá se **pro tisk**.
- Černá se přidala, protože smíšení CMY nedává plně černou barvu, navíc je černý inkoust levnější než barevný.
- Nevýhodou je, že **nedokáže správně zobrazit** sytě červenou, zelenou a modrou.
- Při tisku to však není poznat.
- Před tiskem se RGB obraz převádí do CMYK.
- To provádí buďto ovladač tiskárny nebo RIP (Raster Image Processor - u profi tiskáren).

- RGB se používá pro aktivní zdroje světla, CMYK jsou **pasivní** (světlo pouze **odrážejí**), proto nedokáží udělat tak jasné odstíny.

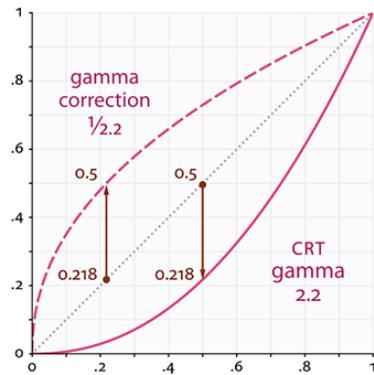


1.1.4 YCbCr

- Barva je reprezentována **jasovou složkou** Y a modrou a červenou **chrominanční** komponentou.
- Není to absolutní barevný model, jedná se o **způsob kódování RGB** informací.
- Využívá se nejčastěji u videa a barevných obrázků, kde je využito faktu, že **lidské oko nejvíce vnímá jas**, který je reprezentovaný složkou Y. Barvy už tak důležité nejsou a proto se můžou například více **komprimovat** bez výraznější ztráty kvality obrazu (JPEG).
- Jasová složka je kódována v intervalu $\langle 0, 1 \rangle$ a chrominanční složky v intervalu $\langle -0.5, 0.5 \rangle$

1.2 Nelinearita grafického výstupu (gamma korekce)

První CRT monitory zobrazovaly jas nelineárně. To znamená, že dvojnásobné napětí neznamená dvojnásobný jas, křivka jasu byla zhruba exponenciální. Tento způsob zobrazení jasu přetrvává i v dnešních monitech. Proto je potřeba upravit i zobrazované barvy. Mapování barev je potřeba provést nelineárně. Nelineární vstup v kombinaci s exponenciální křivkou jasu ve výsledku vede k jasu, který je vnímán jako lineární.



Problém nastává při práci s barvami. Pracovat s barvami je potřeba v lineárním prostoru, aby byly výsledky např. matematických operací korektní. Před zpracováním je tedy nutné převést barvu do lineárního prostoru, po zpracování je potřeba před zobrazením opět převézt barvu zpět do nelineárního prostoru.

$$C_{linear} = C_{sRGB}^{\text{gamma}} \quad (1.1)$$

$$C_{sRGB} = C_{linear}^{\frac{1}{\text{gamma}}} \quad (1.2)$$

1.3 Kompozice rastrových obrazů (alfa kanál)

Alfa kanál v obrazech se používá v počítačové grafice pro průhlednost obrazů. Obrazy mohou obsahovat plnou nebo částečnou průhlednost. Plně transparentní obraz propouští veškeré barvy podkladu, částečně transparentní obraz mixuje část bravy podkladu a část vlastní barvy. Např. výsledná barva obrazu s hodnotou alfa 0.5 vytvoří mix barev tvořený z 50% barvou podkladu a z 50% barvou transparentního objektu. V případě překryvu více transparentních objektů je potřeba objekty setřídit dle vzdálenosti a vypočítat transparentnost postupně.

1.4 HDR

Barvy jsou tradičně reprezentovány trojicí 8bitových hodnot v intervalu $\langle -0, 255 \rangle$. Takto popsány obraz ale může ztráct detaily z důvodu limitovaných možností hodnot pro rozložení barev. Přesnější metodou je ukládání jednotlivých složek jako 16 nebo 32bitových hodnot, pomocí desetiných čísel. Zde ale nastává problém s hodnotami většími než 1. Je teda nutné provést tzv. tone mapping. Proces tone mapping v zásadě převádí hodnoty do intervalu $\langle 0, 1 \rangle$.

Jednou z nejjednodušších metod je Reinhardova metoda, hodnota se vypočítá následujícím vzorcem:

$$C_{out} = \frac{C_{in}}{C_{in} + 1} \quad (1.3)$$

Tato metoda zachová poměrně dobře kontrast pro oblasti obrazu s nízkým jasem, oblasti obrazu s vysokým jasem jsou ale méně kontrastní. Pokročilejší metodou je mapování hodnot pomocí expozice, následujícím vzorcem:

$$C_{out} = 1 - e^{C_{in} * \exp} \quad (1.4)$$

Pomocí hodnoty expozice je pak možné nastavit celkové podání barev obrazu. Hodnota expozice by měla být zvolena v závislosti na aktuálním vstupu, je možné hodnotu expozice volit automaticky.

2 Afinní a projektivní prostor. Afinní a projektivní transformace a jejich matematický zápis. Modelovací a zobrazovací transformace v počítačové grafice.

2.1 Afinní prostor - A_n

- Je to prostor **obsahující body**. Navíc musí být současně dán **přidružený vektorový prostor** (souřadný systém) a nějaké **zobrazení**, které každé dvojici bodů přiřadí vektor (prvek z přidruženého vektorového prostoru).
- Cílem affinního prostoru je mít možnost **jednoznačně specifikovat body** pomocí jejich souřadnice a **manipulovat** s nimi s využitím prostředků lineární algebry.
- **Dimenze** vektorového prostoru určuje dimenzi affinního prostoru.
- Ukázka affinného prostoru:
 - v trojrozměrném affinném prostoru A_3 máme bod X se souřadnicemi $X = (x_1, x_2, x_3)$,
 - vektor \mathbf{x} je prvek onoho přidruženého vektorového prostoru.

2.1.1 Euklidovský prostor - E_n

- Afinní prostor ve kterém je zaveden **skalární součin**(2.1) a **norma**(2.2) (velikost vektoru), to umožňuje měřit délku vektorů a úhly mezi nimi.
- Souřadný systém (kartézský, polární, válcový atd.).

$$\begin{aligned} \text{Vektory } a &= (a_1, a_2, a_3), b = (b_1, b_2, b_3) \\ a \cdot b &= |a| \cdot |b| \cos \alpha, \\ a \cdot b &= a_1 b_1 + a_2 b_2 + a_3 b_3 \end{aligned} \tag{2.1}$$

$$|a| = \sqrt{a_1^2 + a_2^2 + a_3^2} \tag{2.2}$$

2.1.2 Kartézská souřadná soustava

- Souřadné osy jsou **vzájemně kolmé**.
- Protínají se v jednom bodě – **počátku souřadná soustava**.
- Jednotka se obvykle volí na všech osách stejně velká.
- Souřadnice polohy bodu je možno dostat jako kolmé průměty polohy bodu k jednotlivým osám.

2.2 Afinní transformace

- Afinní transformace je **zobrazení bodů jednoho affinního prostoru do jiného** affinního prostoru (speciální případ: zobrazení do téhož affinního prostoru (**bijekce**); tomu se říká **afinita**).
- Afinní transformace souřadnic je **geometrickou transformací** bodu $P = [x, y]$, jehož obrazem je bod $Q = [x', y']$, které spočívá v **posunutí** (translation), **otáčení** (rotation), **změně měřítka** (scaling), **zkosení** (shearing) nebo operaci **vzniklé jejím skládáním**.
- **Afinní** – rovnoběžným přímkám odpovídají opět rovnoběžné přímlky, které však nemusí být rovnoběžné s původními přímkami.
- Geometrické transformace jsou jedněmi z nejčastěji používaných operací v PG.

Když zavedeme následující vektory a matici:

$$y = (y_1, y_2, y_3), x = (x_1, x_2, x_3), t = (t_1, t_2, t_3),$$

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}.$$

Můžeme **afinní transformaci** zapsat jako:

$$y = xA + t,$$

kde y je bod, do kterého je transformován bod x pomocí matice A a **translačního** vektoru t . Vektor t slouží k posunutí středu souřadné soustavy, matice A mění osy souřadné soustavy. Známe-li A a vektor t , můžeme transformaci jednoduše provést. Jindy se musí určit ze zadání.

2.2.1 Ortonormalita affinní transformace

- Ortonormální transformace – jsou takové transformace, které **nemění délky ani úhly**.
- Délky a úhly souvisejí se **skalárním součinem**, když se tento součin po transformaci **nezmění**, jsou zachovány délky i úhly.
- Vlastnosti ortonormální transformace:
 - affinní transformace bude zachovávat hodnotu právě tehdy, když $AA^T = I$, kde I je jednotková matice (**nutná a postačující podmínka ortonormality**),
 - také když uvážíme, že $A^{-1} = A^T$, pak platí $A^{-1}A^T = I$,
 - determinant matice $\det(A)$ musí být roven ± 1 , protože $\det(AA^T) = \det(I) = 1$.

2.2.2 Afinní transformace (2D)

- posunutí** (translation) – posun ve směru x a y je dán hodnotou translačního vektoru,

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} b_x \\ b_y \end{bmatrix}$$

- otáčení** (rotation) – ve směru ručiček (**naopak** prohodíš znaménka u sin),

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

- změna měřítka** (scaling),

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

- zkosení** (shearing),

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & sh_y \\ sh_x & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

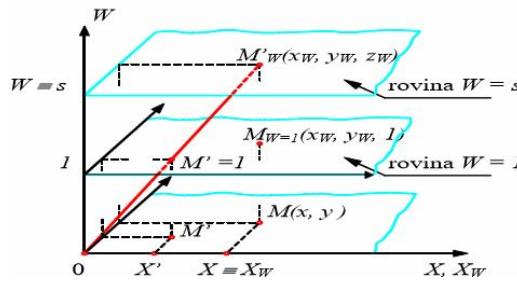
2.3 Projektivní prostor

Při použití affinních prostorů v PG lze v některých situacích narazit na jisté problémy, zejména to, že nelze zobrazovat body v nekonečnu (tzv. nevlastní body). I když jsou praktické scény konečné, mohou i zde vzniknout **nevlastní body** (středové promítání).

Problém spočívá v tom, že vektor reprezentovaný nevlastními body má jednu nebo více složek rovnou $\pm\infty$, kterou nelze v počítači jednoduše reprezentovat. Projektivní prostor navíc umožňuje promítání **středové i rovnoběžné** oproti prostoru affinnímu. Projektivní prostor tedy zavádí **homogenní souřadnice**, které jednotlivé body reprezentují.

2.4 Homogenní souřadnice

- Myšlenkou je reprezentace bodu ve vektorovém prostoru o jednu dimenzi větší – rozšíření o jednu dimenzi (expanze z 2D do 3D, popř. z 3D do 4D).
- Bod (x, y) je v homogenních souřadnicích reprezentován jako (wx, wy, w) kde $w \neq 0$.
- Nejčastěji volíme **homogenní souřadnici** $w = 1$.
- Bod se souřadnicemi (x, y, w) má kartézské souřadnice $x' = x/w$ a $y' = y/w$.



2.5 Projektivní transformace - kolineace

- Kolineace je **zobrazení** bodů jednoho prostoru na body stejného nebo jiného prostoru.
- Kolineaci (projektivní transformaci) lze matematicky **popsat** vztahem $y = xT$, kde vektory x a y reprezentují před a po transformaci a matice T charakterizuje **kolineaci** (transformační matice).
- Sehrává zásadní úlohu v grafických systémech, ty pracují nejčastěji s trojrozměrným projektivním prostorem, kde T je ve tvaru ve tvaru 4×4 a x , y jsou čtyřprvkové vektory homogenních souřadnic.

2.5.1 Základní transformace

U projektivních transformací se můžeme setkat s těmito základními transformacemi:

- **posunutí** (translation),

$$T = \begin{bmatrix} 1 & 0 & 0 & a \\ 0 & 1 & 0 & b \\ 0 & 0 & 1 & c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **otočení kolem osy x** (rotation),

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **otočení kolem osy y** (rotation),

$$T = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **otočení kolem osy z** (rotation),

$$T = \begin{bmatrix} \cos \theta & \sin \theta & 0 & 0 \\ -\sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- **změna měřítka** (scaling),

$$T = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

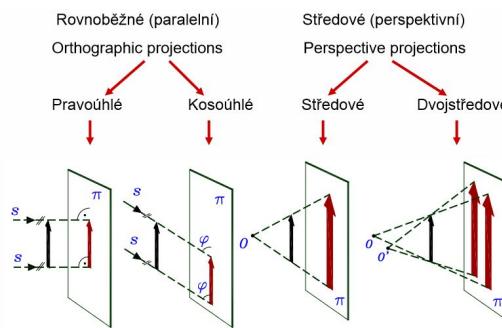
- **zkosení** (shearing),

$$T = \begin{bmatrix} 1 & sh_y & 0 & 0 \\ sh_x & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.6 Promítání

- **Promítání** – zobrazení jednotlivých bodů na předem danou průmětnu.
- V geometrii nejprve volíme promítací metodu a potom v této zobrazujeme objekty (v PG naopak) – nejprve vytvoříme objekt a následně volíme zobrazovací metodu vhodnou pro požadovaný účel.
- Definice promítání:
 - **promítací paprsky** – polopřímka, vycházející z promítacího bodu, směr závisí na typu promítání,
 - **průmětna** (viewing plane) – plocha v prostoru, na kterou dopadají promítací paprsky (paprsky vytvářející průmět),
 - průmětnou nemusí být pouze rovina (polokoule, NURBS plocha...).

2.6.1 Klasifikace promítacích metod

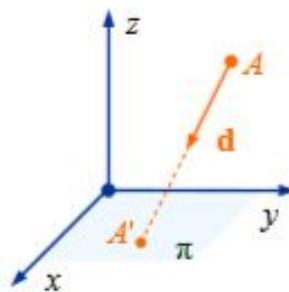


2.6.2 Rovnoběžné promítání

Orthographic nebo orthogonal projection z řeckého „orthos“ rovný a „graphe“ kreslení. Promítání je určeno průmětnou a s směrem s , který není rovnoběžný s průmětnou.

$$\begin{bmatrix} 1 & 0 & -\frac{d_x}{d_z} & 0 \\ 0 & 1 & -\frac{d_y}{d_z} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matice popisuje rovnoběžné promítání na rovinu xy . Směr promítacího paprsku je $d = (d_x, d_y, d_z)$ (viz obr.).

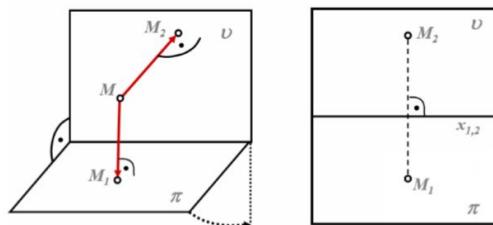


Jednička na pozici 3,3 v matici zajišťuje, že se transformací nezmění souřadnice z bodu. Toho opět využívá řešení viditelnosti.

2.7 Mongeova projekce

- Nejprve promítáme kolmo na vodorovnou rovinu π (**půdorysnu**) – promítací přímky jsou svislé, jde tedy o pohled shora (půdorys).
- Poté promítáme kolmo na svislo rovinu v (**nárysnu**) – promítací přímky jsou kolmé, jde tedy o pohled zpředu (nárys).

- Pohledy kreslíme bez přihlížení k obsahu sklopené druhé průmětny, tudíž se obrazy v jednotlivých průmětních prolínají a jejich polohu v souřadnicovém systému popisuje vzdálenost od základnice (osa Y) pokažmo od nulového bodu.



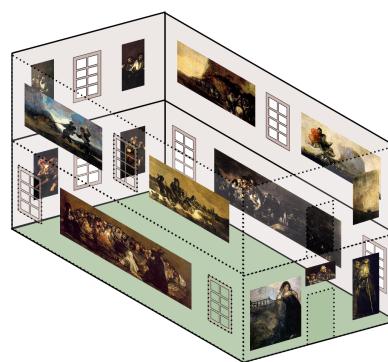
Souřadnice bodů: $M(x, y, z) \dots$ 1. průmět $M1(x, y, 0)$ 2. průmět $M2(x, 0, z)$. V Mongeově projekci je **těleso určeno svým nárysem a půdorysem**.

2.8 Kosoúhlé promítání

- Je rovnoběžné promítání na jednu průmětnu směrem, který má odchulku φ jinou než 90° od průmětny, promítací paprsky S jsou tak rovnoběžné a ne kolmé k průmětně π . Průmětna π je rovnoběžná s některou hlavní rovinou.
- **Výhodou** tohoto způsobu je skutečnost, že **předměty**, které se nacházejí v nárysnu **jsou zobrazeny v reálné velikosti**.

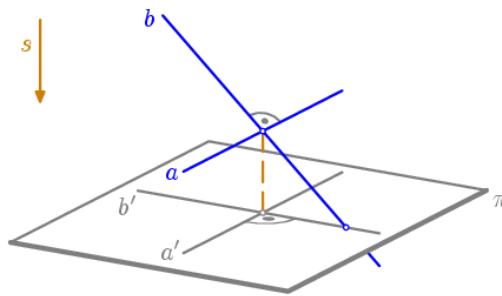
2.9 Axonometrie - rovnoběžné, pravoúhlé promítání

- Axonometrie nebo axonometrické projekce je jednoduchý způsob promítání prostorových těles a trojrozměrných struktur do rovin.
- V rovině se nejprve zvolí tři osy x, y, z , jež spolu svírají stejné nebo nestejně úhly.
- Rozměry těles se pak nanášejí v určitém měřítku rovnoběžně s těmito osami.
- Hlavní výhoda axonometrie proti složitějším metodám promítání je v tom, že průmět se snadno konstruuje, a že se z něho dají rozměry odečíst.
- Nevýhoda může být v tom, že v axonometrické projekci se rovnoběžky nesbíhají a tak je perspektivní dojem nedokonalý (může působit vizuální paradox).



2.10 Ortogonální promítání

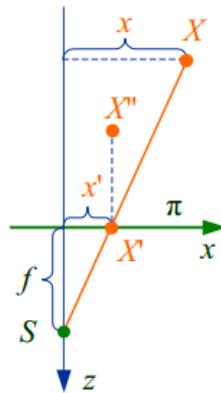
- Směr promítání kolmý k průmětně (jedná se tedy o speciální případ rovnoběžného promítání).
- Zachovávají se všechny vlastnosti rovnoběžného promítání.



2.11 Perspektiva – středové promítání

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \frac{-1}{f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matice popisuje projekci ze středu o souřadnicích $(0, 0, f)$ na rovinu $z = 0$ (tedy na rovinu xy).



Ačkoliv se očekává, že po transformaci bodu bude jeho z souřadnice rovna 0, není to pravda. Tuto nenulovou hodnotu však lze využít například při řešení viditelnosti. Souřadnice x a y slouží k vykreslení na obrazovku.

2.12 Modelovací transformace

Jsou všechny transformace, pomocí nichž se **vytváří scéna**: posun, zkosení, rotace, změna velikosti.

2.13 Zobrazovací transformace

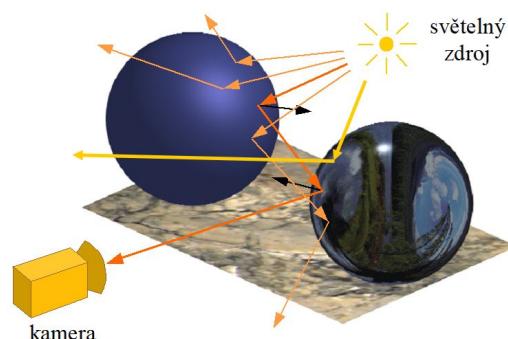
Transformace používané k **zobrazení scény**: středové a rovnoběžné zobrazení. Jsou dělány tak, aby výsledky padly do jednotkového zobrazovacího objemu - souřadnice z intervalu $\langle -1, 1 \rangle$.

3 Metody získávání fotorealistických obrazů, rekurzivní sledování paprsku, radiometrie, zobrazovací rovnice, Monte Carlo přístupy ve výpočtu osvětlení, urychlovací metody.

- Syntetizace fotorealistických obrazů je oblastí PG, která dovoluje vykreslit jakoukoliv uměle vytvořenou scénu tak, jak by vypadala v reálném světě.
- Toho dosahuje díky implementaci optických zákonů, které lze běžně pozorovat.

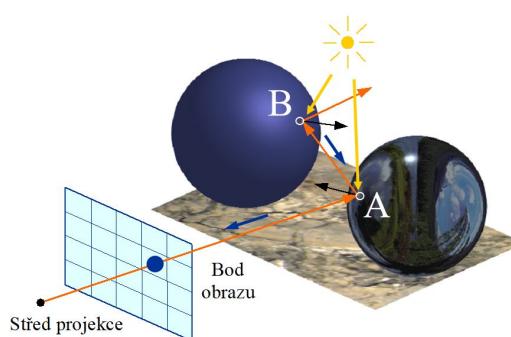
3.1 Sledování paprsku - ray tracing

- Metoda sleduje šíření paprsků ve scéně.
- Tyto paprsky začínají **ve světelném zdroji**, odráží se o tělesa v prostoru a některé z nich nakonec dopadnou do průmětny (obdobně jako světlo v reálném světě).
- Paprsky, které takto prochází scénu, lze znázornit jako strom.
- Tento přístup je však **neefektivní**, protože **velká část paprsků do průmětny nikdy nedopadne**, takže nemají přínos pro výsledný obraz a zbytečně zvyšují výpočetní čas.



3.2 Zpětné sledování paprsku

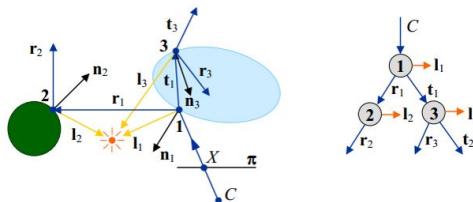
- Funguje stejně jako běžné sledování paprsku, ovšem paprsky jsou **vysílány z kamery do scény**.
- Jakmile paprsky dosáhnou **ukončovacího kritéria** (jdou mimo scénu/maximální počet odrazů), sledujeme zpět jejich pohyb (navrácení z rekurze) a vypočítáváme osvětlení.
- Tím se eliminuje možnost, že by paprsek nepřinesl žádný přínos výslednému obrazu a značně se tak urychluje celý proces ray tracingu.



3.3 Rekurzivní sledování paprsku

- Metoda vyšetřuje „běh“ světelných paprsků ve scéně.
- Světlo je reprezentováno **paprsky**, které jsou do scény vyzařovány světelnými zdroji a **putují prostorem scény**, některé dopadnou na povrchy těles, jiné odletí ze scény.

- Paprsek, který dopadne na povrch tělesa se může **odrazit** (zákon odrazu) nebo pokud je těleso průhledné, může se paprsek **zlomit** (zákon lomu) – oba druhy paprsků mohou opět dopadnout na povrch tělesa, kde se celý proces znovu opakuje.
- Do scény se vyšle **velké množství paprsků**, ale podstatné jsou ty, které projdou objektivem myšlené kamery, pokud na průmětnu dopadne dostatečný počet paprsků, vykreslí se obrázek.
- Metoda je sice jasná a fyzikálně podložená, ale nepoužívá se, protože se obtížně realizuje. (Je potřeba vydat velké množství paprsků, ale k objektivu kamery by jich dorazilo jen malé množství a ostatní by se sledovaly zbytečně).
- Řešením je **otočit paprsky a vyslat je od kamery ke světelnému zdroji**. Principiálně to pak funguje stejně.



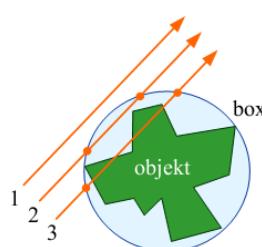
- Rovnice výpočtu lokálního osvětlení: $I = I_l + k_r I_r + k_t I_t$
 $I_l = I_a O_a + \sum_i S_i f_{att,i} I_i (O_d \cos \varphi_i + O_s \cos^n \alpha_i)$. Hodnota S_i představuje viditelnost i-tého zdroje světla v daném bodě.

3.4 Urychlování trasování

Největším problémem je hledání průsečíků paprsků s objekty scény.

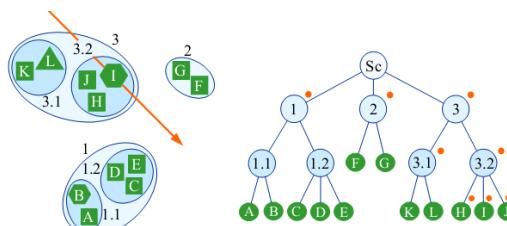
1. Nejjednodušším řešením je využít **ohraničujících ploch** („bounding boxů“). Ohraničující plocha se vytvoří kolem každého objektu ve scéně. Nejlépe když má plocha následující vlastnosti
 - objekt leží **celý uvnitř** ohraničující plochy, ale plocha jej obepíná co **nejtěsněji**,
 - průsečíky paprsků s plochou musí jít spočítat, co nejjednodušším výpočtem,
 - plochu musí být možné pro jednotlivé objekty dostatečně jednoduše nalézt.

Je možné použít **kulovou plochu** (ne moc vhodné, objekty můžou být protáhlé a tato plocha by je neobepínala dostatečně těsně). Další variantou plochy je **kvádr** (taky sice není moc vhodný, protože těleso může být našikmo a taky by jej neobepínal moc natěsně, nicméně nalezení ohraničující plochy je snadné – minimální a maximální hodnoty obepínaného tělesa).



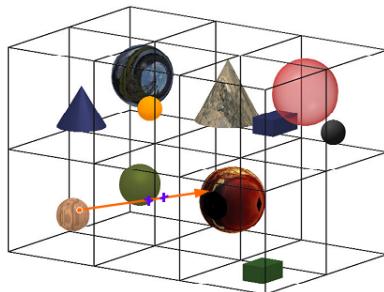
Princip ohraničujících ploch spočívá v tom, že pokud paprsek neprotne ohraničující plochu, pak neprotne ani těleso uvnitř (velmi časté). Odhalením této situace dojde ke značnému zrychlení. Pokud je to naopak, hledají se průsečíky s ohraničeným tělesem.

2. Rozšířením předchozího je **organizování ohraničujících ploch do hierarchických struktur**. Princip je stejný, pokud se neprotne rodičovská plocha, nehledají se dále ani průsečíky s potomky. Nevýhodou je, že nelze jednoduše takovou strukturu automatizovaně nalézt.



3. Další metodou je **Dělení prostoru scény na podprostory**. Obykle se dělí rovinami souřadné soustavy $xy, xz, yz \rightarrow$ vznikají tak velké kvádry (stejně velké / různě velké). Princip metody:

- u neurychlené metody byly všechny objekty organizovány v 1 velkém seznamu,
- nyní je zřízeno tolik seznamů, kolik je objemových elementů vzniklých dělením prostoru, každý element bude mít svůj seznam objektů, které do něj aspoň z části zasahují (pokud objekt zasahuje do více elementů, bude v seznamu každého z nich) – hledání průsečíků začíná v tom elementu, kde je počátek paprsku, při opouštění elementu lze zjistit, do kterého elementu vstupuje, paprsek kontroluje pouze průsečíky s objekty, které jsou v seznamu daného elementu.



4. Dalším metodou je **Adaptivní hloubka rekurze**. Odhaduje se, zda je paprsek pro stanovení intenzity ve zkoumaném obrazovém bodě dostatečně užitečný. Pokud ne, tak se nevyšle (např. u odrazů či průchodu tělesy).

3.5 Vyzařovací metoda - radiozita

Na rozdíl od předchozí metody rekurzivního sledování paprsku (dobře zobrazuje lesklé, dobře osvícený předměty) je tato metoda spíše protikladná. Zaměřuje se na **difúzní odrazy světla** – vhodná pro matné povrchy a rozptýlené světlo (např. interiéry). Princip:

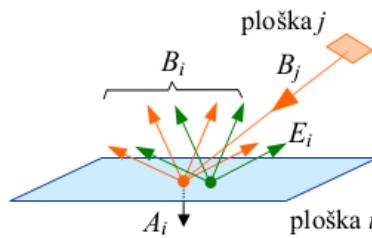
- Vypočítá se, jak jsou osvětlena jednotlivá místa scény.
- Podle toho se povrchy těles pokryjí sítí (v místech kde je komplikovaný průběh osvětlení je síť hustá – zlom světla a stínu).
- Pro každou plošku jsou spočítány hodnoty RGB – vyzařování je konstantní na celém povrchu plošky.
 - **PROBLÉM:** kdyby se takto plošky zobrazovaly, mohly by se sousedící plošky výrazně lišit intenzitou a nebylo by to pěkné.
 - **ŘEŠENÍ:** po výpočtu intenzit plošek se intenzity přenesou do jednotlivých uzlů sítě (zprůměrování intenzit okolních plošek, které obklopují uzel) a následně se intenzity interpolují. (proto i to hustší dělení, kde je přechod světlo–stín ...).
- Nejjednodušším, ale ne zrovna nejsprávnějším zobrazením scény a interpolací je pomocí **Gouradova stínování**.
 - **PROBLÉM:** osvětlení bylo spočítáno v prostoru scény a tam by se měla provádět i interpolace, ale Gouraudovo stínování interpoluje v prostoru obrazu. Problém je, že při středové projekci se nezachovává dělící poměr a proto budou výsledky v prostoru obrazu rozdílné od výsledků z prostoru scény. Nicméně Gouraudovo stínování se používá, protože je rychlé.
- Vytváření sítě probíhá v několika iteracích: nejdřív se hustota odhadne, pak se spočítá osvětlení a dle výsledků se síť dohustí tam, kde je třeba. Základní myšlenou je, že na všech ploškách ustanov **energetická rovnováha**: **výkon vyzařovaný + výkon absorbovaný = výkon na plošku dopadající od jiných ploch + výkon, který ploška sama vyzařuje**

$$B_i = E_i + p_i \sum_{j=i}^n B_j F_{j \rightarrow i} \frac{A_j}{A_i}.$$

kde:

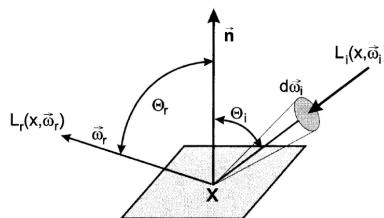
- B_j – výkon vyzářený ploškou j ,
- p_i – míra odrazu (optické vlastnosti materiálu),

- E_i – hodnota výkonu vlastního vyzařování plošky,
- A_i, A_j – velikost plošek (plošný obsah),
- $F_{j \rightarrow i}$ – konfigurační koeficient říká, jaká část výkonu vyzářeného ploškou j dopadne na plošku i (jedná se o $\int \langle 0, 1 \rangle$, záleží na pořadí indexů).



3.6 BRDF (Bidirectional Reflectance Distribution Function)

- Charakterizuje **odrazové schopnosti povrchu materiálu** v určitém bodě x .
- Jedná se o **poměr odraženého záření** ke vstupnímu diferenciálnímu zařízení, promítnutému na kolmou plochu.
- BRDF v daném bodě zůstává stejná i když změníme směr paprsku.
- **Pozitivita BRDF**: funkce není nikdy záporná.
- **Zákon zachování energie**: plocha nemůže odrazit více než je celková přijatá energie
- **Odrazivost** $p(x) = \frac{d\Phi_r(x)}{d\Phi_i(x)}$; $d\Phi_r(x)$ je **odražený světelný tok**, $d\Phi_i(x)$ je **dopadající světelný tok**.
- Obor hodnot odrazivosti je na intervalu $<0, 1>$, $1 = \text{plný odraz}$.
- **PRINCIP**: Vysílá se mnoho paprsků v každém bodě s různými offsety, některé padnou do zdroje světla, některé ne. Výsledná hodnota pixelu je poté průměrem všech hodnot paprsků (čím více paprsků vyšleme, čím lepší je výsledek (méně zrní)).

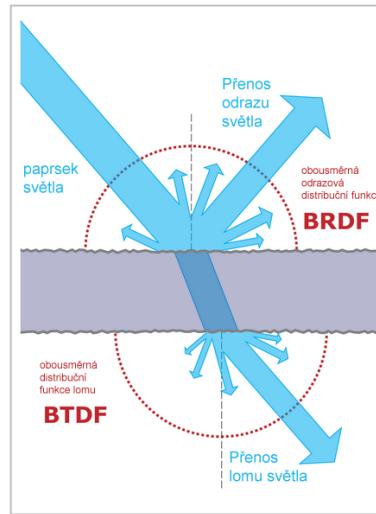


3.7 BTDF (Bidirectional Transmittance Distribution Function)

Dvousměrná distribuční funkce lomu. Popisuje **průchod světla povrchem**.

3.8 BSDF (Bidirectional Scattering Distribution Function)

- Obousměrná distribuční funkce **rozptylu**.
- Je to souhrn dvou distribučních funkcí, a to funkce odrazu (BRDF) a lomu (BTDF).
- **BSDF + BTDF + BRDF**



3.9 Renderovací rovnice

Rekurzivní diferenciální rovnice

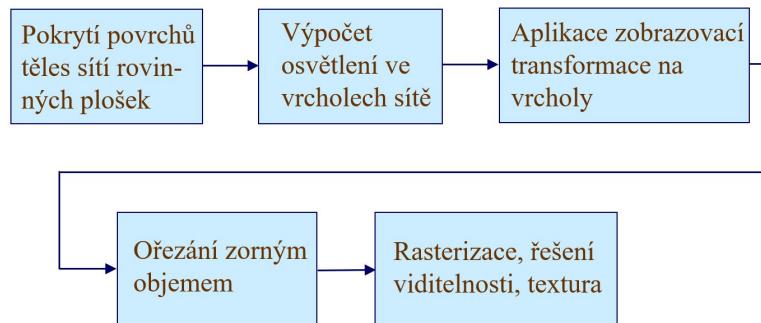
$$L(x, \omega_0) = L_e(x, \omega_0) + \int_{\Omega} L(r(x, \omega_i) - \omega_i) \cdot BRDF(\omega_i, x, \omega_0) \cos \theta_i d\omega_i$$

Zjednodušeně: osvětlení povrchu = samovolně vyzařované světlo + součet příchozího osvěrlení ze všech směrů krát BRDF.

4 Standardní zobrazovací řetězec a realizace jeho jednotlivých kroků, modely osvětlení a stínovací algoritmy, řešení viditelnosti, možnosti výpočtu globálního osvětlení v reálném čase, stručná charakteristika standardu OpenGL.

4.1 Standardní zobrazovací řetěz

Klade důraz na rychlosť nikoli na kvalitu, realizuje ho OpenGL.

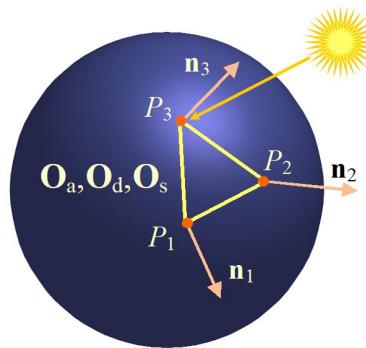


1. Pokrytí povrchu objektů sítí roviných plošek:

- Ploškami bývají nejčastěji **trojúhelníky** nebo čtyřúhelníky.
- Pro objekty ve tvaru mnohostěnu je takové dělení vcelku samozřejmé.
- K **přesnějšímu výpočtu** barev bývá, ale někdy dělení na plošky **jemnější**.
- Někdy síť roviných plošek žádaný povrch pouze approximuje.

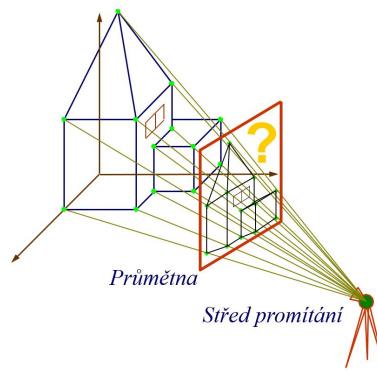
2. Výpočet osvětlení ve vrcholech sítě – k tomu známe:

- Polohu, intenzitu a barvu světelných zdrojů.
- Souřadnice vrcholů (P), normál (n) a konstanty popisující optické vlastnosti materiálu (O_a, O_d, O_s).
- V tomto kroku je počítán barevný vjem každého vrcholu.



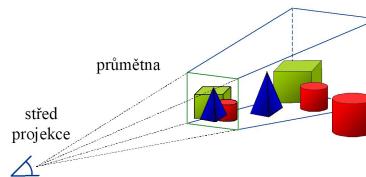
3. Aplikace zobrazovací transformace na vrcholy

- Oblíbenou technikou je středové promítání, to je zadáno:
 - Polohou průmětny.
 - Polohou středu promítání.



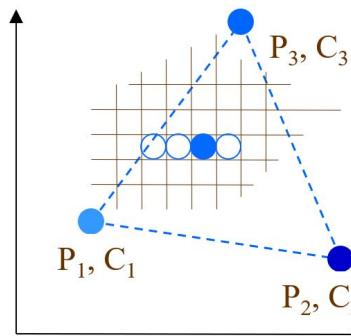
4. Ořezání zorným objemem

- Objekty nebo jejich části, nacházející se mimo zorný objem (obvykle jehlan) jsou odstraněny.



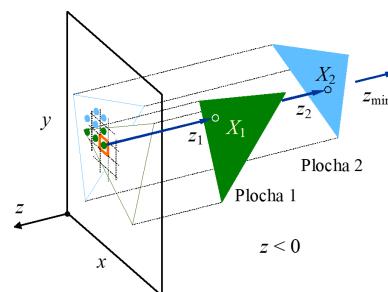
5. Rasterizace plošek

- Postupně se zpracovávají všechny plošky.
- Pro každou plošku jsou „rozsvěceny“ všechny její pixely.
- Barva každého pixelu se stanoví **interpolací mezi hodnotami ve vrcholech**.



5.1 Řešení viditelnosti (z-buffer)

- Pro rozhodnutí viditelnosti se použijí hodnoty souřadnice z (zde je $z_1 > z_2$).
- Před řešením viditelnosti bývá středové promítání převedeno na rovnoběžné.



5.2 Nanášení textury

- Vzhled obrázků lze vylepšit nánášením textury.

4.2 Stínování (shading)

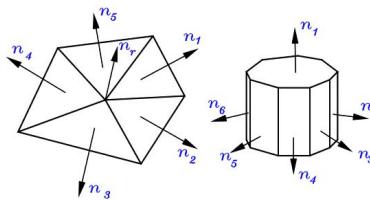
- Vykreslování barevných objektů **různými odstíny barev**.
- Lze odlišit křivosti ploch a tím docílit lepšího prostorového vjemu.
- Neplést s výpočtem vrženého stínu.
- Základní typy: **Konstantní** stínování, **Gouraudovo** stínování (Interpolace barvou), **Phongovo** stínování (Interpolace normálových vektorů).

4.3 Gouraudovo stínování (Interpolace barvou)

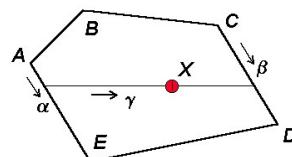
Princip metody spočívá v tom, že pokud budeme znát **normálu** v každém **vrcholu** každé plochy objektu, pak lze vypočítat barvu v tomto vrcholu a **interpolací** vypočítat **barvu pixelu uvnitř plošky** (bilineární interpolace).

Přesto ani tento způsob stínování neposkytuje zcela věrný obraz reálných objektů - interpolace samotného odstínu barvy totiž **nemůže** způsobit místní zvýšení jasu na ploše, stejně jako nemůže kvalitně vytvořit **odlesky** způsobené odraženým světlem. Dá se říci, že tato metoda zahlažuje barevné rozdíly u místních nerovností povrchu.

Normálový vektor pro každý vrchol vypočteme jako aritmetický průměr normálových vektorů plošek, které se v tomto vrcholu stýkají.



1. Vypočteme **normálové vektory pro všechny plošky** ze kterých je objekt složený.
2. Pro každý vrchol spočítáme **normálový vektor** v tomto vrcholu jako **průměr normálových vektorů plošek**, které se v tomto vrcholu stýkají.
3. Z normálových vektorů ve vrcholech a pozice světelného zdroje vypočteme **barvy ve vrcholech plošek**.
4. Provedeme **interpolaci** barvy pro **body jednotlivých plošek**.



$$\mathbf{f}_X = (1-\gamma) \cdot [(1-\alpha) \cdot \mathbf{f}_A + \alpha \cdot \mathbf{f}_E] + \gamma \cdot [(1-\beta) \cdot \mathbf{f}_C + \beta \cdot \mathbf{f}_D]$$

Výhody

- + umožnuje dobře zobrazit i hladké objekty,
- + používá se jako nejčastější metoda stínování.

Nevýhody

- nevznikají ostré odlesky uprostřed polygonů.

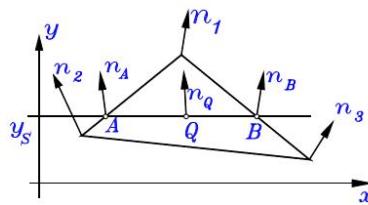
4.4 Phongovo stínování (Interpolace normálových vektorů)

- Provádí se **interpolace normálových vektorů** jednotlivých vrcholů, Interpolaci provádíme po řádcích.
- Tento metodou se **odstraní problém neostrých odlesků**.
- Je **náročnější** na výpočet než goraudovo stínování, jelikož se počítá výsledná barva v každém bodě plošky (ne pouze ve vrcholech).
- Pro normálové vektory lze psát:

$$n_A = n_1 + (n_2 - n_1) \cdot u; u < 0, 1 >,$$

$$n_B = n_1 + (n_3 - n_1) \cdot w; w < 0, 1 >,$$

$$n_Q = n_A + (n_B - n_A) \cdot t; t < 0, 1 >,$$

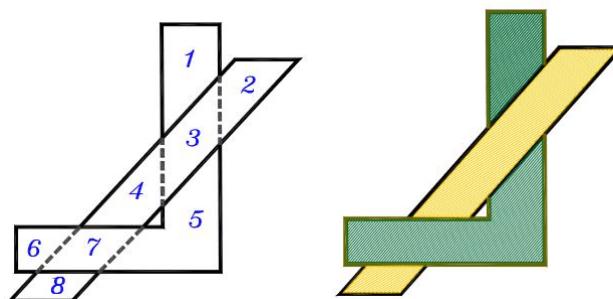


4.5 Řešení viditelnosti

- Podle výsledných dat:
 - Vektorové algoritmy** – geometrické prvky vrcholy, hrany a stěny. Výstupem je vektorové řešení.
 - Rastrové algoritmy** – výsledkem je rastrový obraz (jednotlivé pixely obsahují barvu), většina současných metod.
- Podle místa řešení:
 - Řešení v prostoru objektů** – proovnávání vzájemné polohy těles $O(n^2)$.
 - Řešení v prostoru obrazu** – pracujeme s promítnutými a rasterizovanými objekty. Pro pixely hledáme nejbližší objekty.

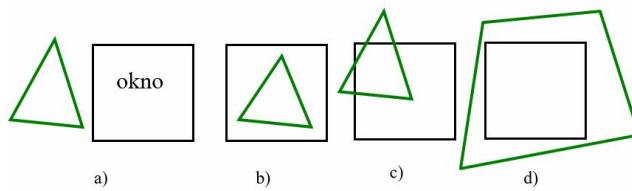
4.5.1 Rastrové algoritmy

- Malířův algoritmus** (Painter's algorithm) – porovnává plochy z hlediska jejich z -tových souřadnic (plocha s menší z -tovou souřadnící bude kreslena první), jestliže se plochy **nepřekrývají**, potom na pořadí kresby nezáleží, pokud se protínají – rozdělit na nepřekrývající se plochy.

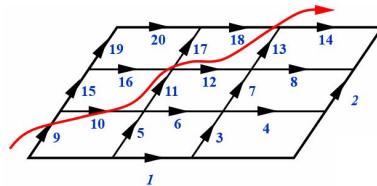


- Dělení obrazovky** (Warnock subdivision)

- Všechny plošky leží mimo zónu - zůstane barva pozadí.
- Oblast obsahuje právě jeden celý n-úhelník. Daná oblast se vyplní barvou a zbytek pozadím.
- Oblast protíná právě jeden n-úhelník. Daná část se vyplní barvou, zbytek pozadí.
- Pokud zobrazovaná část je celá uvnitř jednoho n-úhelníka, potom se celá oblast zobrazí barvou nejbližšího n-úhelníka, který oblast obklopuje.
- Pokud nenastane jeden z vyjmenovaných případů - oblast se rozdělí.

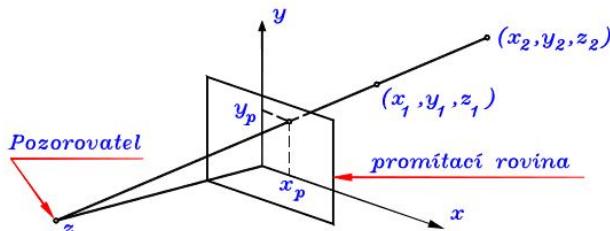


3. **Plovoucí horizont** (Floating Horizon Algorithm) – metoda „zig–zag“, počítáme od „nejbližšího“ rohu plochy k „oku“ pozorovatele.



4. **Paměť hloubky** (Z–buffer, depth–buffer)

- Vyplň obrazovou paměť barvou pozadí.
- Vyplň paměť hloubky – nekonečnem.
- Pro každou plochu najdi její průměr (rasterizaci) nalezenému pixelu $[x_i, y_i]$ přiřaď hloubku z_i .
- Porovnej hloubku a zapiš do paměti, pokud je hloubka menší překreslí se nový pixel daným objektem.



- Nejznámější a nejefektivnější metoda.
- Každá plocha se zpracovává **pouze jednou**.
- **Doba zpracování roste s počtem ploch lineárně** (záleží i na velikosti ploch).
- Není potřeba žádné třídění nebo pomocné datové struktury.
- Možnost **paralelních** procesů.
- Z-buffer je často realizován jako 2D pole, kdy se ukládá pouze aktuální hodnota z (nejmenší).

5. **Z–buffer – paměť hloubky – průhlednost - princip**

- Inicializuj **color buffer** a **depth buffer**.
- Postupně načti všechny plochy, neprůhledné zpracuj, průhledné si zapamatuj a odlož pro následné zpracování.
- Po zpracování neprůhledných ploch setříd průhledné plochy podle vzdálenosti.
- Zpracuj průhledné plochy s použitím **alfa míchání**.

4.6 Grafický standard OpenGL: stručná charakteristika

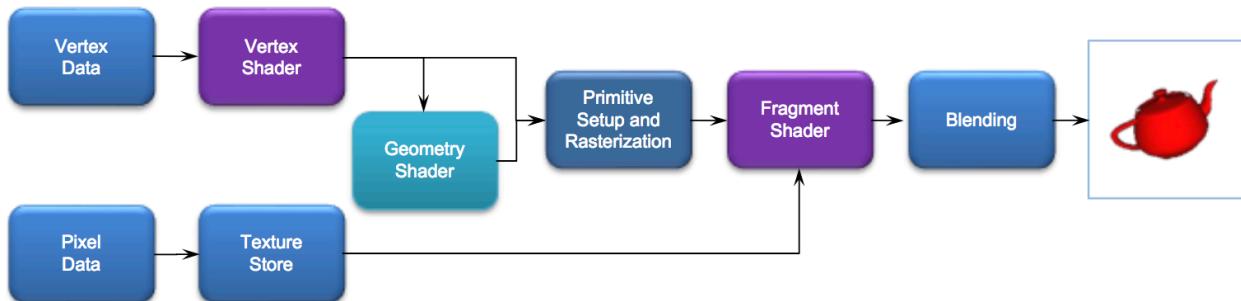
- OpenGL je **multiplatformní** standard poskytující rozhraní (**API**) pro zobrazování 2D a 3D objektů. Je podporován takřka všemi výrobci grafických karet.
- Používá se pro tvorbu **PC Her, CAD programů**, aplikací **virtuální reality** či pro **vědecko-technické vizualizace**.
- Veškerá činnost OpenGL se řídí vykonáváním příkazů pomocí **volání funkcí** a procedur (kterých je v OpenGL cca 250).

- Byla vytvořena tak, aby byla **nezávislá** na použitém **operačním systému** nebo **programovacím jazyce**. Specifikace OpenGL tedy **neříká nic** o řízení kontextu platformy, **správě** a vytváření **oken, audio a vstupu**. Řešení této problematiky je necháno na **podpůrných systémech** (které jsou většinou platformě specifické) a OpenGL se zabývá pouze vykreslováním.
- Z programátorského hlediska se OpenGL chová jako **stavový automat**. To znamená že pokud změníme nějaké nastavení, toto nastavení zůstane až do doby než jej znovu změníme. To se hodí např. když jediným příkazem jsme schopni přepnout program do kreslení ve „wireframe módu“.
- Programátorské rozhraní knihovny OpenGL je vytvořeno tak, aby knihovna byla použitelná v téměř **libovolném programovacím jazyce**. Primárně je k dispozici hlavičkový soubor pro jazyky C a C++. Existují však i podobné soubory s deklaracemi pro další programovací jazyky, například Fortran, Object Pascal či Javu; tyto soubory jsou většinou automaticky vytvářeny z Cékovských hlavičkových souborů.

4.6.1 Core-profile vs Immediate mode

Ve starších verzích OpenGL (před 3.2, kde se stal deprecated) se vyvídely aplikace v tzv. **immediate módu** (non-shader mód). Tento mód byl značně **jednodušší** než core-profile, jelikož zakrýval (abstrahoval) většinu funkcionality OpenGL pod danou knihovnu, za to však zaplatil svou **neefektivitou**. To je způsobeno (mimo jiné) tím, že při každé specifikaci vertexu je jeho pozice odeslána na GPU, což představuje bottleneck v komunikaci mezi CPU a GPU. Tento mód lze stále použít v moderním OpenGL pokud využíváme tzv. **compatibility profile**.

Moderní OpenGL tedy nutí uživatele využívat moderní praktiky, přičemž pokud se snaží využívat starých metod, OpenGL vyhodí chybu a přestane vykreslovat. Přestože je core-profile na první pohled značně **složitější**, je **efektivnější**, umožňuje **větší flexibilitu** a **lepší porozumění grafického programování**. Core-profile přináší **buffer objecty** a je založen převážně na použití **shaderů**, které jsou vždy zkompilované do malých podprogramů na CPU a nahrány na **GPU pouze jednou**.



Od verze 4.1 přibyla ještě mezi vertex a geometry shader fáze **Tessalace**.

```

// ...: Initialization code :: ...
// 1. bind Vertex Array Object
glBindVertexArray(VAO);
// 2. copy our vertices array in a vertex buffer for OpenGL to use
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
// 3. copy our index array in a element buffer for OpenGL to use
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
// 4. then set the vertex attributes pointers
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
 glEnableVertexAttribArray(0);

[...]

// ...: Drawing code (in render loop) :: ...
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0)
glBindVertexArray(0);
  
```

4.6.2 GLSL (OpenGL Shading Language)

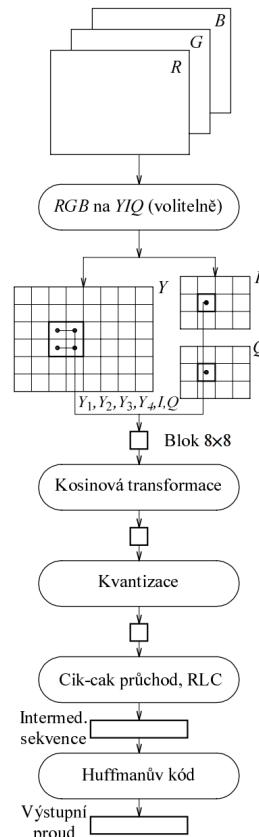
Je součástí OpenGL 2.0 a vyšší. Jedná se o jazyk pro programování shaderů, který je syntaxí velice podobný C. Shadery jsou malé programy, které běží na GPU. Jeho vlastnosti:

- Definuje **speciální proměnné** `gl_Position` (výstupní pozice z vertex shader) a `gl_FragColor` (výstupní barva z fragment shaderu).
- **Datové typy:** vektory, matice, int, float, bool, C-like structures.
- Předávání hodnot mezi vertex/geometry/fragment shadery lze pomocí **vstupních a výstupních proměnných** `in`, `out`, `inout`.
- Možnost vytváření vlastních **funkcí** (default funkce musí být `main()`).

5 Komprese obrazu a videa, principy úprav obrazu v prostorové a frekvenční doméně.

5.1 JPEG

JPEG představuje jeden z nejpoužívanějších obrazových formátů (fotografie), jedná se o **ztrátový** formát obrazků. **Komprese JPEG** pracuje v několika krocích, principem je **redukce vysokofrekvenčních dat** v obrazků při zachování co nejvíce informací o nízkofrekvenčních datech. To je založeno na tom, že lidské oko se zaměřuje spíše na nízké frekvence (malou změnu jasu na ploše) a vysoké frekvence vnímá hůrky (hrany). Můžeme si je tedy dovolit redukovat bez znatelné ztráty kvality obrazu. Zároveň se využívá **podvzorkování barev** → znova z toho důvodu, že lidské oko vnímá více jas než barvy.



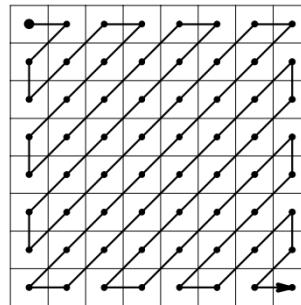
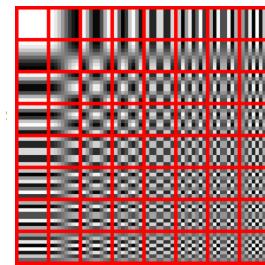
5.1.1 Komprese JPEG

1. **Převod RGB na YCbCr** – vstupní obraz se převede na YCbCr, což jej rozdělí na **jasovou složku** (Y) a **chrominanční** složky (barvonošné Cb a Cr). To nám umožňuje následnou redukci barvonošné složky v jednom ze 3 módů: 4:4:4 (nepodvzorkovává se), 4:2:2 (horizontálně na polovinu, pro 2 jasové 1 barva), 4:2:0 (horizontálně i vertikálně na polovinu, pro 4 jasové 1 barva).
2. **Provedení DCT** – obraz se následně rozdělí na bloky o velikosti 8×8 . Pro každý blok se provede 2D DCT (diskrétní kosinova transformace), ta narozdíl od DFT produkuje pouze reálné komponenty. Výsledek DCT nám vrátí počet jednotlivých frekvencí, které se v obrazu nachází, přičemž **nízké frekvence** se koncentrují vlevo nahore a **vysoké frekvence** vpravo dole (viz obrázek).

$$F(k, l) = \frac{1}{4} c(k) c(l) \sum_{m=0}^7 \sum_{n=0}^7 f(m, n) \cos \frac{(2m+1)k\pi}{16} \cos \frac{(2n+1)l\pi}{16}$$

kde,

$$c(k), c(l) = \begin{cases} 1/\sqrt{2}, & k, l = 0 \\ 1, & \text{jinak} \end{cases}.$$



3. **Kvantizace** – jednotlivé bloky jsou **vyděleny kvantizační maticí a zaokrouhleny**. Koeficienty matice definují **míru komprese** (čím větší hodnoty tím větší komprese a horší obraz). Většinou po aplikaci kvantizace v každém bloku zůstane pouze několik hodnot **vlevo nahore** to vyplývá z toho co jsme zmínil výše → více se redukují vysoké frekvence, které si můžeme dovolit vynechat. Výsledek před a po kvantizaci je vidět obrázku níže.

139	144	149	153	155	155	155	155
144	151	153	156	159	156	156	156
150	155	160	163	158	156	156	156
159	161	162	160	160	159	159	159
159	160	161	162	162	155	155	155
161	161	161	161	160	157	157	157
162	162	161	163	162	157	157	157
162	162	161	161	163	158	158	158

a) Blok obsahující vzorky jasů.

235.6	-1.0	-12.1	-5.2	2.1	-1.7	-2.7	1.3
-22.6	-17.5	-6.2	-3.2	-2.9	-0.1	0.4	-1.2
-10.9	-9.3	-1.6	1.5	0.2	-0.9	-0.6	-0.1
-7.1	-1.9	0.2	1.5	0.9	-0.1	0.0	0.3
-0.6	-0.8	1.5	1.6	-0.1	-0.7	0.6	1.3
1.8	-0.2	1.6	-0.3	-0.8	1.5	1.0	-1.0
-1.3	-0.4	-0.3	-1.5	-0.5	1.7	1.1	-0.8
-2.6	1.6	-3.8	-1.8	1.9	1.2	-0.6	-0.4

b) Výsledek kosinové transformace.

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

c) Kvantizační matice pro složku Y .

15	0	-1	0	0	0	0	0
-2	-1	0	0	0	0	0	0
-1	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

d) Spektrum po kvantizaci.

4. **Zig-zag intermediální sekvence** – obraz po kvantizaci se projde **zig-zagem** a z daných čísel se vytvoří intermediální sekvence (dvojice ve tvaru **Symbol-1, Symbol-2**) pomocí **Run-Length encoding**. RLE kóduje pouze AC složky (1 - 64), DC složka (px na souřadnicích [0, 0] každého bloku) se kóduje **diferenciálně** vzhledem k DC složce v předchozím bloku. Intermediální sekvence vypadá následovně:

- **Symbol-2** – (AMPLITUDE) značí hodnotu na daném pixelu po kvantizaci.
- **Symbol-1** – (RUNLENGTH, SIZE), kde RUNLENGTH značí počet nul, které danému prvku předchází v zig-zag sekvenci (pokud je počet větší než 15, zapíše se 15) a SIZE je počet bitů nutných k reprezentaci AMPLITUDE. **Speciální hodnota** (0, 0) říká, že předchozí nenulová hodnota byla poslední.
- **Kódování – DC** složky se kódují odlišně (SIZE, AMPLITUDE), **AC** složky ((AMPLITUDE), (RUNLENGTH, SIZE)).

Intermediální sekvence se poté zakóduje **Huffmanovým kódem**, kde se kódují pouze kombinace (RUNLENGTH, SIZE) u AC a (SIZE) u DC podle předem daných tabulek. AMPLITUDE se zapisuje pomocí **jednotkového doplňku** (způsob reprezentace záporných čísel).

Intermediální sekvence: (2)(3), (1,2)(-2), (0,1)(-1), (0,1)(-1), (0,1)(-1), (2,1)(-1), (0,0)
(předpokládáme, že v předchozím bloku byla stejnosměrná složka 12).

Kódování hodnot Symbol-1: (2)→011, (0,0)→1010, (0,1)→00, (1,2)→11011, (2,1)→11100.

Kódování hodnot Symbol-2: (-1)→0, (1)→1, (-3)→00, (-2)→01, (2)→10, (3)→11.

Výsledný kód: 011 11 11011 01 00 0 00 0 11100 0 1010.

5.2 MPEG

MPEG je zkratkou pro Moving Picture Expert Group. Cílem práce této skupiny bylo standardizovat metody komprese videosignálu. Existuje několik standardů MPEG-(1, 2, 4, 7).

Úroveň	Profil	Simple	Main	High
Low		4:2:0 352×288 4 Mb/s I,P,B		
Main	4:2:0 720×576 15 Mb/s I,P	4:2:0 720×576 15 Mb/s I,P,B	4:2:0, 4:2:2 720×576 20 Mb/s I,P,B	
High		4:2:0 1920×1152 80 Mb/s I,P,B	4:2:0, 4:2:2 1920×1152 100 Mb/s I,P,B	

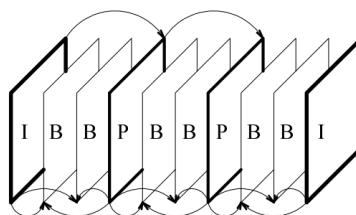
Tab.7.2. MPEG-2: profily a úrovně.

- **MPEG-1** – první standard, dokončen v roce 1991. Navržen zejména pro práci s obrazy 352×288 pixelů, 25FPS (odvozeno od televizní normy PAL) nebo 352×240 , 30FPS (odvozeno od NTSC) při datovém toku **1.5 MBit/s** (optimální tok, mohl být i vyšší).
- **MPEG-2** – dokončen v roce 1994, mnohem velkorysejší implementace, snaží se být co **nejuniverzálnější**. Zavádí několik **profilů** (podmnožina z nejširší možné syntaxe) a **úrovní** (definuje parametry v rámci daného profilu).
- **MPEG-3** – práce na tomto standardu byly zastaveny (měl sloužit pro HDTV) později byl sloučen do MPEG-2.
- **MPEG-4** – metodou komprese se značně liší oproti MPEG-1,2, je určen pro **extrémně nízké datové toky**.
- **MPEG-7** – neříká nic o kódování, jedná se o standard pro popis dat (**metadata**) s multimediálním obsahem.

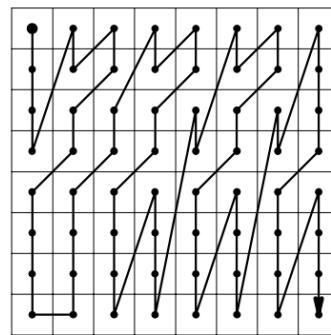
5.2.1 Komprese MPEG

Standard MPEG, rozlišuje **tři typy rámců (I, B, P)**, ve své podstatě pro kódování využívá stejných principů jako JPEG (**rámeček I** se kóduje nezávisle). Oproti JPEG však využívá i **časové koherence**. Tedy k dosažení maximální komprese se předpokládá, že po sobě jdoucí rámce jsou s největší pravděpodobností dosti podobné. Počítá se ovšem s tím, že části obrazů se mohou přemístit, k tomu se využívá vložených rámců P a B, které se kódují **závisle** vzhledem k ostatním.

- **I** – jsou kódovány každý **zvlášť**, bez vazby na rámce předcházející či následující. Princip kódování (komprese) je stejný jako u standardu **JPEG** (i když v detailech existují některé **odlišnosti**: jiná kvantovací tabulka, jiná struktura intermediální sekvence a jiný způsob **zig-zagu** (podle MPEG-2)). **Kvantizační tabulka** může být pro každý makroblok jiná – změnou měřítka lze **řídit tok dat** (některé aplikace mohou vyžadovat konstantní tok dat).
- **P (Predicted)** – tento rámec je kódován **vzhledem k jedinému předcházejícímu rámci typu I nebo P**.
- **B (Interpolated bi-directionally)** – tyto rámce jsou kódovány vzhledem k nejbližšímu **předchozímu** a **nejbližšímu** budoucímu rámci typu **I** nebo **P**. Jejich použití je nepovinné ale z hlediska dosahovaných kompresních poměrů výhodné. **Komplikace** z použití rámci B spočívá v uchovávání v paměti dva kotevní obrazy. Dále je nevyhnutelné jisté časové zpoždění, protože nejprve musí být k dispozici obraz **novější** a teprve potom může být kódován obraz starší.



Obr. 7.5. Sekvence IBBPBBPBB... v proudu MPEG rámciů.

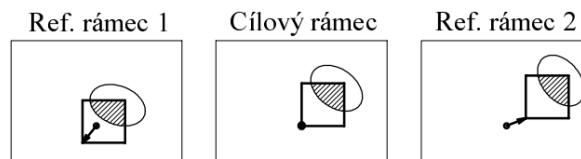


Obr. 7.4. Alternativní postup při kódování bloku v MPEG-2.

Velmi často používané řazení rámciů je IBBPBBPBBI. Kódování rámciů P a B taky probíhá po **makroblokách**. Pro každý makroblok v cílovém (tj. právě kódovaném rámci) jsou sestaveny **vektory pohybu** (pro každý makroblok v P **jeden** vektor, v B jsou vektory **dva**) vzhledem k referenčním rámciům.

Vektor pohybu je definován takto: jestliže o uvedený vektor posuneme kódovaný makroblok a porovnáme s odpovídající částí referenčního obrazu, pak je dosaženo dobré shody. Vektory posunutí se stávají **součástí komprimované sekvence**. Po nalezení vektoru jsou **kódovány difference** – podobně jako u JPEGu mezi odpovídajícími makrobloky bude v dvou rámciích **malý rozdíl** a výsledné data po kvantizaci vyžadují tak **malé sekvence** (komprese).

- Bloky **P** se kódují $T - R$, kde T je makroblok v cílovém rámci,
- bloky **B** se kódují $T - 0.5(R_1 + R_2)$, kde R_1 a R_2 jsou makrobloky v cílových rámciích.



Vektory pohybu:



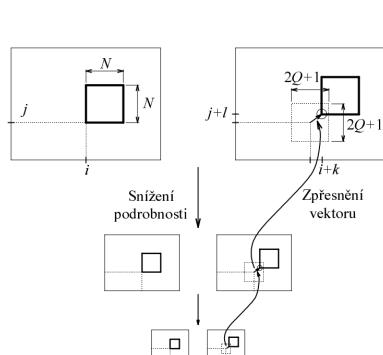
Diference:
$$\frac{1}{2} [\square - \square + \square]$$

Obr. 7.6. Kódování rámciů typu B.

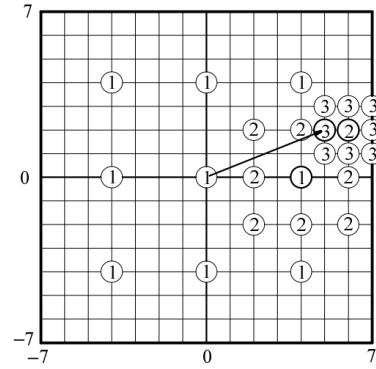
5.2.2 Stanovení pohybového vektoru

Stanovení pohybového vektoru MPEG norma nepředepisuje, jedná se však o jeden z **nejobtížnějších problémů**. Většinou se využívá pouze **jasové** (Y) složky, existuje několik metod:

1. **Porovnání makroblocků** – v referenčním rámci se naleze makroblok, který nejvíce odpovídá zpracovávanému makrobloku. Pro určení shody lze použít např. SSD. **Účinné**, avšak velmi **časově náročné**.
2. **Logaritmické vyhledávání** – vylepšení předchozí metody, má logaritmickou časovou složitost. V prvním kroku algoritmus testuje 9 dvojic. V každém dalším kroku se testuje vždy po 8 dvojicích rozmištěných kolem předchozího bodu, který měl největší shodu. Rychlejší, ale nemusí být tak přesné jak předchozí metoda.
3. **Rekurzivní dělení** – rekurzivně dělíme obraz na menší a menší. Poté nalezneme první odhad v nejmenším obrazu a se zvyšující se velikostí (návrat z rekurze) pozici vektoru jen upřesňujeme. Rychlé a spolehlivé.

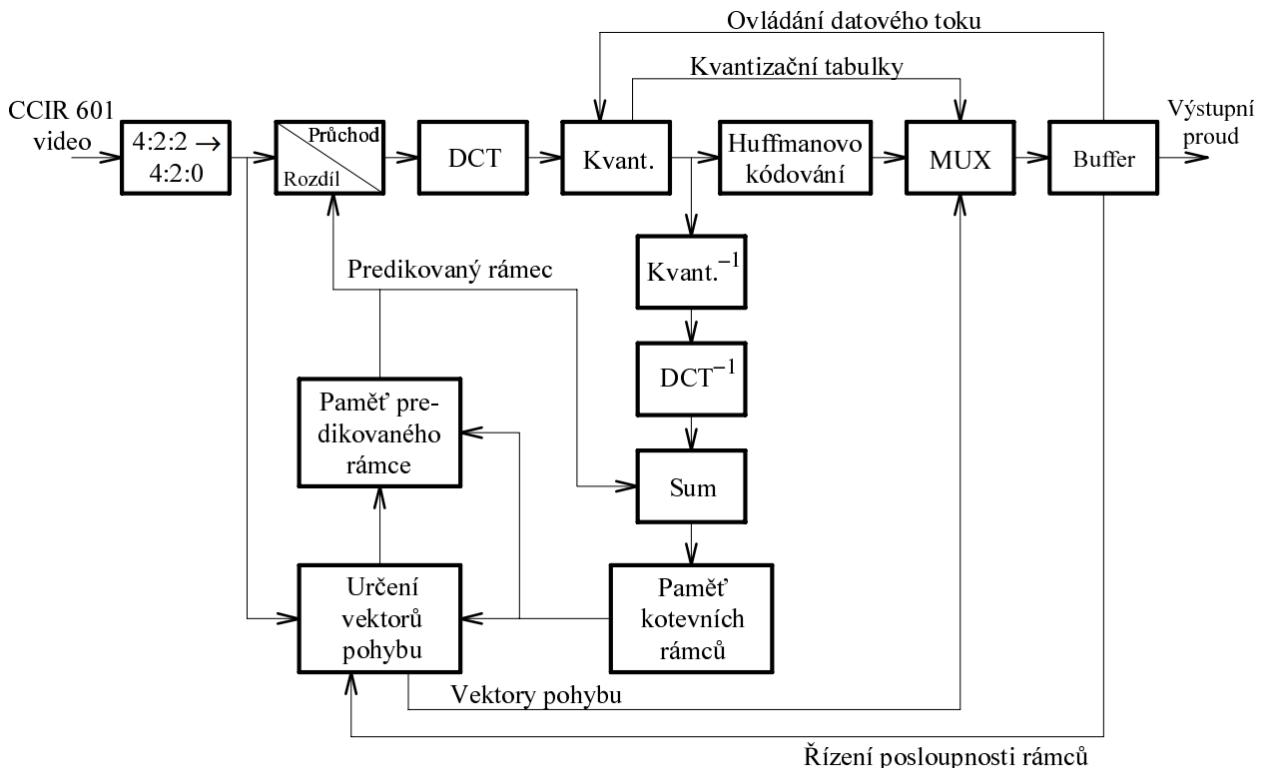


Obr. 7.8. Hledání vektoru pohybu s využitím obrazu s nižší podrobností.



Obr. 7.7. Stanovení pohybového vektoru logaritmickým vyhledáváním.

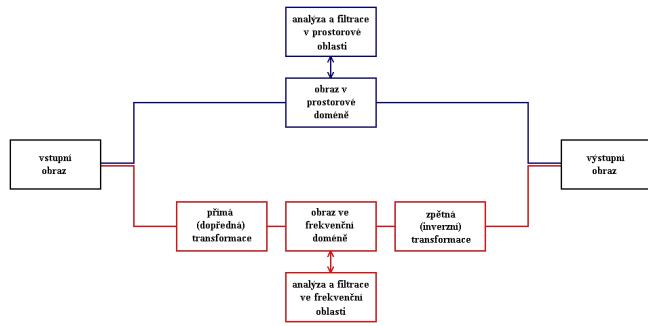
Kódování složek vektorů se provádí **inkrementálně vzhledem k předchozí hodnotě**. Zároveň může nastat situace kdy se vektor nepodaří nalézt, v tom případě je možné kódovat dané makrobloky nezávisle na ostatních (stejně jako I).



Obr. 7.10. Schéma MPEG enkodéru obrazu.

5.3 Principy úprav obrazu v prostorové a frekvenční doméně

Úpravy nad obrazy lze provádět v **prostorové** (obraz je reprezentován souřadnicemi x, y a hodnotou pixelu) nebo **frekvenční** (snímek je popsán harmonickými funkciemi (sin a cos) různé amplitudy, frekvence a fáze) doméně. Většina operací nad prostorem signálů je popsána **operátorem**, pro převod do frekvenční domény se poté používá **(Diskrétní) Fourierova transformace**.



5.4 Operace nad prostorem signálů

- **Lineární operace** – úprava signálu je popsána operátorem \mathcal{O} , velmi často předpokládáme, že operátor \mathcal{O} má nějaké speciální vlastnosti. Platí: $\mathcal{O}\{af(x, y) + bg(x, y)\} = a\mathcal{O}\{f(x, y)\} + b\mathcal{O}\{g(x, y)\}$.
- **Operace invariantní vůči posuvu** – považujeme aby měl operátor kromě linearity ještě další vlastnosti (invarianci vůči posuvu). Operátor \mathcal{O} je invariantní vůči posuvu, pokud pro všechna $f(x, y)$ platí: $g(x-a, y-b) = \mathcal{O}\{f(x-a, y-b)\}$. Méně formálně: pokud provedeme operátor na dva vzájemně posunuté ale jinak shodné signály, jejich výsledek bude také vzájemně posunutý ale jinak **stejný**.
- **Dirakův impulz** – je impulzová funkce $\delta(x, y) = 0$, v bodě $(0, 0)$ je hodnota **nekonečno**, všude jinde nulovou, přičemž platí $\delta(-x, -y) = \delta(x, y)$, a **integrál** přes celou funkci je roven 1. Jinak řečeno se jedná o funkci, která je nekonečně vysoká a nekonečně úzká v bodě $(0, 0)$.

5.5 Konvoluce (lineární sumace bodových zdrojů)

Konvoluce představuje základní matematický **operátor**, který pracuje s dvěma funkcemi (značí se $*$). Umožňuje zpracování obrazových signálů jak v prostorové tak frekvenční doméně (konvoluční teorém). Spojitá konvoluce je definována takto:

$$f(x, y) * h(x, y) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} h(a, b) f(x-a, y-b) da db,$$

kde funkci h nazýváme **konvoluční jádro** (filtr), přesně řečeno se jedná o impulzovou charakteristikou filtru. Při výpočtu se funkce h v rovině x, y otočí o 180° (důsledek členů se zápornými znaménky). Konvoluční operátor má tyto **vlastnosti**:

- **Komutativní** – $f * g = g * f$.
- **Asociativní** – $f * (g * h) = (f * g) * h$.
- **Distributivní** – $f * (g + h) = (f * g) + (f * h)$.
- **Existence jednotky** – $f * \delta = \delta * f = f$, kde δ je dirakův impulz a $\delta(x) = 0, x \neq 0$.
- **Asociativita při násobení skalárem** – $a(f * g) = (af) * g = f * (ag)$
- **Konvoluční teorém** – $\mathcal{F}(f * g) = [\mathcal{F}(f)] \cdot [\mathcal{F}(g)] = F \cdot G$, kde $\mathcal{F}[f(x)]$ značí Fourierovu transformaci. Jinými slovy: Fourierovým obrazem konvoluce funkcí f, g je součin jejich Fourierových obrazů F a G .

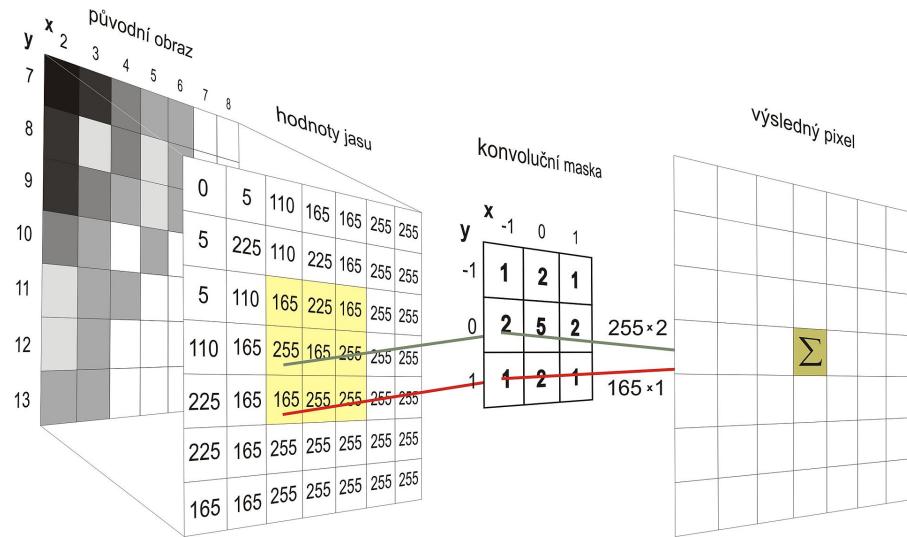
5.5.1 Diskrétní konvoluce

V počítačové grafice využíváme často pro zpracování dvourozměrného diskrétního signálu, je popsána tímto vzorcem:

$$(f * h)(x, y) = \sum_{i=-k}^k \sum_{j=-k}^k f(x-i, y-j) \cdot h(i, j)$$

V případě diskrétní konvoluce lze jádro chápat jako **tabulku** (konvoluční masku), kterou přiložíme na příslušné místo obrazu. Každý pixel překrytý tabulkou vynásobíme koeficientem v příslušné buňce a hodnoty sečteme, ty uložíme na pozici středového pixelu. V případě, že koeficienty již nejsou normalizované, je nutné je **normalizovat** (zprůměrovat), např.: maska 3×3 pokud je všude 1 (uniformní filtr) tak výsledek je nutné vydělit 9.

Hodnoty konvoluční masky mají vliv na výsledek operace, používají se masky pro: **rozostření, zostření, detekci hran**, atd.

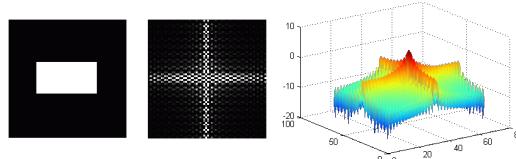


5.6 Fourierova transformace

Slouží pro převod obrazových signálů z prostorové do **frekvenční domény** (na komponenty složené ze sínů a cosínů). Pro analýzu **diskrétního** obrazu se využívá DFT, která je dána vztahem:

$$F(k, l) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m, n) \frac{1}{\sqrt{MN}} \exp \left[-j2\pi \left(\frac{mk}{M} + \frac{nl}{N} \right) \right],$$

kde $k = 0, \dots, M-1$, $l = 0, \dots, N-1$. Symbol $F(k, l)$ značí frekvenční spektrum obrazu s souřadnicemi (k, l) , které nabývá hodnot od 0 - (M, N) . Frekvenční spektrum obsahuje **komplexní čísla**, proto se pro zpracování obrazového signálu ve frekvenční oblasti používají **amplitudy** komplexních čísel (amplitudová frekvenční charakteristika) nebo výkonová spektrální hustota (koeficienty násobené komplexně sdruženým číslem). Příklad amplitudové charakteristiky je na následujícím obrázku.



Tato frekvenční charakteristika obsahuje **koeficienty odpovídající různým frekvenčním složkám**. Analýzou a operacemi s koeficienty ve frekvenční oblasti lze **modifikovat obraz v prostorové oblasti** např. realizovat filtr typu **dolní propust pro vyhlazení obrazu**. Pro získání modifikovaného obrazu je třeba převést koeficienty zpět do prostorové oblasti. Tomuto procesu se říká zpětná nebo někdy **inverzní Fourierova transformace** IDFT, jejíž diskrétní varianta IDFT je dána následujícím definičním vztahem:

$$f(m, n) = \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} F(k, l) \frac{1}{\sqrt{MN}} \exp \left[j2\pi \left(\frac{mk}{M} + \frac{nl}{N} \right) \right].$$

5.7 Filtrace ve frekvenční oblasti

Postup filtrace obrazů ve frekvenční doméně je následující:

1. aplikace diskrétní Fourierovy transformace (DFT) $f(x, y) \rightarrow F(u, v)$,
2. aplikace filtru $G(u, v) = H(u, v)F(u, v)$, kde:
 - $G(u, v)$...výslední snímek
 - $H(u, v)$...funkce filtrace (konvoluční maska?)
 - $F(u, v)$...původní snímek

inverzní Fourierova transformace (IDFT) $G(u, v) \rightarrow g(x, y)$.

5.7.1 Nízkofrekvenční, pásmový, vysokofrekvenční filtr

Bázové funkce jsou síný a cosíný s rostoucí frekvencí. Tedy $F(0,0)$ (střed obrázku) reprezentuje DC složku, tedy **průměrný jas** napříč obrazem a $F(M-1, N-1)$ reprezentuje **nejvyšší frekvenci**, to lze pozorovat na amplitudové charakteristice výše.

Tedy **čím dál od středu, tím je frekvence vyšší**. Zároveň můžeme vidět, že **amplituda je menší pro vyšší frekvence**, nízké frekvence tedy obsahují více informací o obrazu než ty vyšší a obecně se jich ve většině obrazů nachází více (čím rychlejší změna jasu/gradient tím vyšší frekvence). Všech těchto znalostí můžeme využít k návrhu požadovaného filtru:

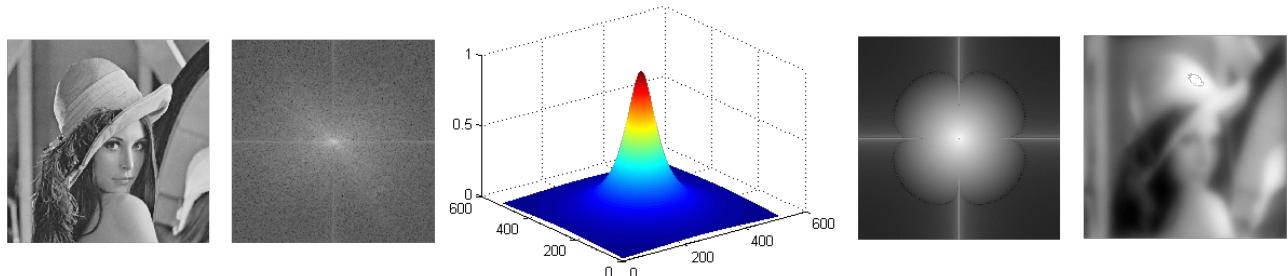
- **Nízkofrekvenční** (lowpass) filtr může například propouštět pouze frekvence kolem středu (tedy ty nízké) a ostatní vynulujem, tento filtr lze navrhnout pomocí binární masky (bílý kruh ve středu) → **minimalizujeme ostré hrany**.
- **Vysokofrekvenční** (highpass) filtr, postup je analogický k předchozímu, pouze tentokrát vynulujeme frekvence na středu a ty na krajích (vysoké) propustíme → **zvýrazníme hrany**.
- **Pásmový** filtr propouští pouze předem definované pásmo.

5.7.2 Butterworthův filtr

Často se pro filtraci ve frekvenční oblasti využívá Butterworthův filtr, který má ze všech běžných filtrov (Gaussian, Chebyshev, Bessel) **nejméně zvlněné frekvenční spektrum** a konverguje k nule u maximální frekvence. Je dán vztahem:

$$H(u, v) = \frac{1}{1 + \left(\frac{D(u, v)}{D_0}\right)^{\frac{2}{n}}}, \quad \text{kde } D(u, v) = \sqrt{(u - u_c)^2 + (v - v_c)^2}$$

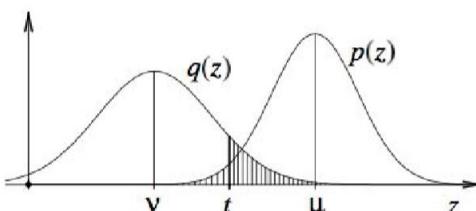
kde $d(x, y)$ představuje běženě používanou L2-normu a n je řád filtru (nejčastěji 1 nebo 2).



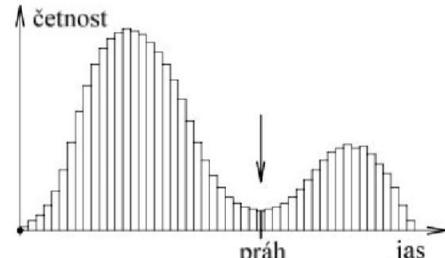
6 Základní metody úpravy a segmentace obrazu (filtrace, prahování, hrany, oblasti, rohy).

6.1 Prahování

- Cílem prahování je **oddělit pozadí od popředí** na základě stanoveného prahu (nějaká reálná hodnota). Výsledkem binární obraz (1 = objekt, 0 = pozadí).
- Práh může být buď **stejný** pro celý obrázek, anebo **adaptivní** pro jednotlivé části obrazu. Další možností je stanovit práh v **intervalu** a, b . Úspěšnost detekci oblastí závisí na správné hodnotě prahu.
- Pokud neznáme hodnotu prahu, snažíme se jí stanovit na základě informací získaných z obrazu, který má být **segmentovaný**.
- **Bimodální histogram** (dva kopce), **multimodální histogram** – práh určit jako **minimum histogramu** mezi vysokými hodnotami, pak lze dále rekursivně dělit (předpokládáme, že v obrazu jsou převážně dva a více druhů pixelů).
- **Obraz lze rekursivně dělit** na menší části, ve kterých se vypočte histogram a dle něho určí práh pro konkrétní část (pokud nelze práh určit, lze ho interpolovat pomocí sousedních prahů).
- **Minimalizace rizika chyby**:
 - stanovení prahu tak, aby se minimalizovala špatná detekce,
 - stanovení dle approximace normálních rozdělení popředí a pozadí $\varepsilon = \theta P(t) + (1 - \theta)[1 - Q(t)]$.
 - nejlepších výsledků lze dosáhnout v **extrému první derivace**
 - pokud je zastoupení pozadí a popředí stejné a má stejný rozptyl $(t - \mu)^2 = (t - v)^2$.



Obr.8.23. Stanovení prahu minimalizací chyby.



Obr.8.22. Bimodální histogram jasu.

- Na levém obrázku je práh označen t a vyšrafovovaná oblast značí chybu, která nastane při prahování, kdy bude špatně rozpoznané popředí/objekt $q(z)$ a pozadí $p(z)$ – minimalizace chyby.

6.2 Detekce hrani

- Každá oblast je obklopena hranicí.
- Hranice se skládá z hran (případně také z jediné zakřivené hrany).
- Hrana se skládá z jednotlivých hranových bodů.
- Většinou se postupuje tak, že se obraz převede do stupně šedi a následně se naleznou jednotlivé body hran.
- Za bod hrany se často považuje místo, kde průběh jasu **vykazuje náhlou změnu**, případně **inflexní bod**.
- Po nalezení jsou jednotlivé nalezené body hran spojovány různými technikami do hran a celých hranic.

6.2.1 Detekce hran s využitím gradientu

- Hrana je v obrazu zastoupeny (prudkou) **změnou jasu**, lze ji tedy najít zkoumáním síly a směru gradientu v jednotlivých bodech.
- Pro určení směru gradientu či hrany (**směr gradientu je kolmý ke směru hrany**) je třeba provést **derivaci** (nejlépe v x i y), která je při výpočtu nahrazena diferencí.
- Diference může být buď **centrální** nebo **dopředná/zpětná**.

$$d_x = \frac{I(x-1, y) - I(x+1, y)}{2}, \quad d_y = \frac{I(x, y-1) - I(x, y+1)}{2}.$$

- Velikost hrany lze určit velikostí gradientu (norma), hrana je tam, kde $e > \text{práh}$. (hrana je kolmá k gradientu)

$$e(x, y) = \sqrt{(f_x(x, y))^2 + (f_y(x, y))^2}$$
- Směr hrany a gradientu lze určit (kde φ – směr gradientu, ψ – směr hrany)

$$\varphi(x, y) = \arctan \left[\frac{f_y(x, y)}{f_x(x, y)} \right], \quad \psi(x, y) = \varphi(x, y) + \frac{\pi}{2}.$$

- Výše uvedené derivace lze nahradit **konvolučními maskami**

- **Sobel** – vážený průměr (Prewittové dělá pouze normální)
- **Kirsch** – počítání hran v 8 směrech

- Robertsův operátor:

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}.$$

- operátor Previtové:

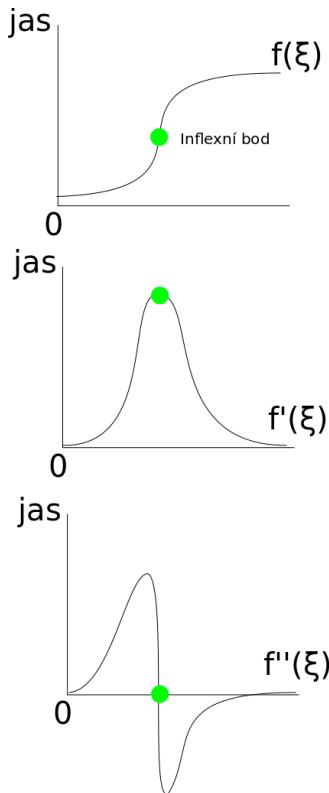
$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

- Sobelův operátor:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

- Kirschův operátor:

$$\begin{bmatrix} -5 & -5 & -5 \\ 3 & 0 & 3 \\ 3 & 3 & 3 \end{bmatrix}.$$



Obrázek 6.1: Velikost gradientu a jeho první a druhá derivace

6.2.2 Detekce hran hledáním průchodu nulou

- První derivace obrazové funkce nabývá svého maxima v místě hrany.
- **Druhá derivace protíná** v místě hrany **nulovou hodnotu**.
- Spolehlivější metoda, než hledání maxima v první derivaci. **NE** v případě, že je obraz postižen šumem. V tomto případě selhává, jelikož druhá derivace ještě více zesílí šum.

Laplaceův operátor (druhá derivace gradientu)

- Pro výpočet se používá symetrická diference nebo konvoluční masky (na krajích je maska ořezaná)

$$d_x = I(x-1, y) - 2I(x, y) + I(x+1, y), \\ d_y = I(x, y-1) - 2I(x, y) + I(x, y+1).$$

0	1	0
1	-4	1
0	1	0

1	1	1
1	-8	1
1	1	1

a) b)

0	1	0
1	-3	1
0	0	0

0	1	0
0	-2	1
0	0	0

a) b)

Obr.8.6. Konvoluční masky pro výpočet Laplaceova operátoru.

Obr.8.7. Příklady masek pro body na krajích a v rozích obrazu.

- Hrana je detekována jako **změna znaménka v průchodu mezi dvěma extrémy**.
- Je **více citlivý na šum než první derivace** (i při malém šumu je detekováno množství falešných hran).
- Pro redukci šumu a zahlazení vysokých frekvencí lze použít **Gaussův operátor**:

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 1 & 2 \\ 1 & 2 & 1 \end{bmatrix}.$$

6.2.3 Cannyho detekce hran

Canny první stanovil požadavky, které by měl detektor splňovat a následně navrhl detektor. Požadavky:

- **Minimalizovat** pravděpodobnost **chybné detekce**.
- Najít polohu hrany, co **nejpřesněji**.
- Bod hrany identifikovat **jednoznačně**.

Postup:

1. Eliminace šumu Gaussovým filtrem.
2. Velikost a směr gradientu – nejčastěji Sobelův operátor (nebo centrální derivace).
3. Nalezení lokálních maxim a stanovení interpolace v osmi okolí. **Redukce na hranu velikosti 1 px**.
4. Eliminace nevýznamných hran (**double thresholding**)
 - Všechny body, kde je velikost hrany $\leq t_{high}$ – „jistá“ hrana
 - Pak ty, které jsou $> t_{low}$ a sousedí s hranou – „jistá“ hrana

7 Základní metody rozpoznávání objektů, příznakové rozpoznávání. Univerzální příznaky pro rozpoznávání (např. HOG), trénovací klasifikátory (např. SVM).

7.1 Příznakové rozpoznávání

- Založeno na tom, že každý objekt lze popsat množinou vhodných číselných hodnot = **příznaků**. Těch může být i více a potom tvoří **vektor příznaků**.
- Vektor příznaků nese všechny podstatné informace o objektu a je **jedinou informací** pro následné **rozpoznání**.
- Počet příznaků a jejich typy bývá **obtížné** určit, je nutné provést experimenty. **Velikost** vektoru příznaků by měla být rozumná (ne příliš velká).
- Obecně by zvolené příznaky měli **splňovat** následující:
 - hodnoty příznaků jsou podobné pro objekty stejných tříd a odlišné pro objekty jiných tříd,
 - měly by být pokud možno nezávislé na hodnotách jiných příznaků.

7.1.1 Momenty

Momenty **různého stupně** patří k často používaným příznakům. Důvodem je, že jejich schopnost od sebe rozlišit objekty různých tříd často vyhovuje a jejich **výpočet je snadný**. Moment vztažený k souřadné soustavě obrazu:

$$m_{p,q} = \int \int_{\Omega} x^p y^q f(x, y) dx dy,$$

kde $f(x, y)$ je obrazová funkce, p, q udávají **stupeň** momentu a Ω je ta část obrazu, kterou považujeme za rozpoznávaný objekt. U diskrétních obrazů se nahrazuje **sumou**. Hodnoty p, q lze zvolit $p \geq 0, q \geq 0$, přičemž: $m_{0,0}$ – **plocha objektu** vážená hodnotou jasu. Máme 3 základní typy momentů:

- **invariantní vůči posunu** (nezávislá na poloze objektu) - $m_{p,q} \rightarrow \mu_{p,q}$,
- **invariantní vůči měřítku** (nezávislá na velikosti objektu, normalizace) - $m_{p,q} \rightarrow v_{p,q}$,
- **invariantní vůči rotaci** dána momentovými invarianty - sestavení θ_n ,

Výpočet polohy těžiště (nezávislost na poloze)

K výpočtu polohy těžiště objektů lze použít momentů $m_{0,0}, m_{1,0}, m_{0,1}$, což dává:

$$x_t = \frac{m_{1,0}}{m_{0,0}}, \quad y_t = \frac{m_{0,1}}{m_{0,0}}.$$

Také **poloha objektu** reprezentována jeho těžištěm může být sama o sobě použita jako **příznak** (x_t, y_t) . Momenty vztažené vzhledem k těžišti, které je činí **nezávislé na poloze objektu** avšak stále závislé na **velikosti a rotaci** objektu, lze vypočítat:

$$\mu_{p,q} = \int \int_{\Omega} (x - x_t)^p (y - y_t)^q f(x, y) dx dy.$$

Momenty nezávislé na velikosti

Pokud není pro rozpoznání důležitá velikost objektu, je možné použít **normalizovaných momentů** $v_{p,q}$ (normalizace pomocí součtu jasových hodnot objektu):

$$v_{p,q} = \frac{\mu_{p,q}}{(m_{0,0})^{\gamma}}, \quad \text{kde } \gamma = \frac{p+q}{2} + 1.$$

Momenty nezávislé na natočení

Pokud nemá rozpoznání záležet ani na orientaci (natočení), je nutné momenty počítat k hlavním osám objektu. Hlavní osy vytvářejí souřadnou soustavu, která má počátek v těžišti rozpoznávaného objektu a vzhledem k souřadné soustavě obrazu je pootočena o úhel Θ . Pootočení θ je dáno podmírkou, aby momenty $\mu'_{2,0}$ a $\mu'_{0,2}$ vypočítány k hlavním osám byly extrémy. Opět lze použít i samotný úhel pootočení jako **příznak**.

7.1.2 Pravoúhlost a podlouhlost

Hranici objektu **postupně rotujeme** v rozsahu $0 - 90^\circ$. Krok úhlu rotace zvolíme v jednotkách stupňů (např. 5°). Po provedení rotace opíšeme kolem objektu pravoúhelník, jehož strany jsou rovnoběžné se stranami obrazu (po provedení rotace stačí nalézt extrémy hranice). Ze všech pravoúhelníků vybereme ten, který má **nejmenší plochu** A_O , a délky jeho stran značíme postupně A_R , a , b . Pravoúhlost (R) a podlouhlost (S) je definována následovně:

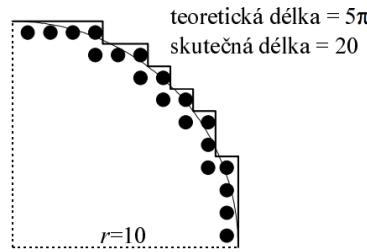
$$R = \frac{A_O}{A_R}, \quad S = \frac{a}{b}.$$

7.1.3 Kruhovost

Nechť P , označují délku hranice objektu a jeho plochu, kruhovost je pak definována vztahem:

$$C = \frac{P^2}{A}.$$

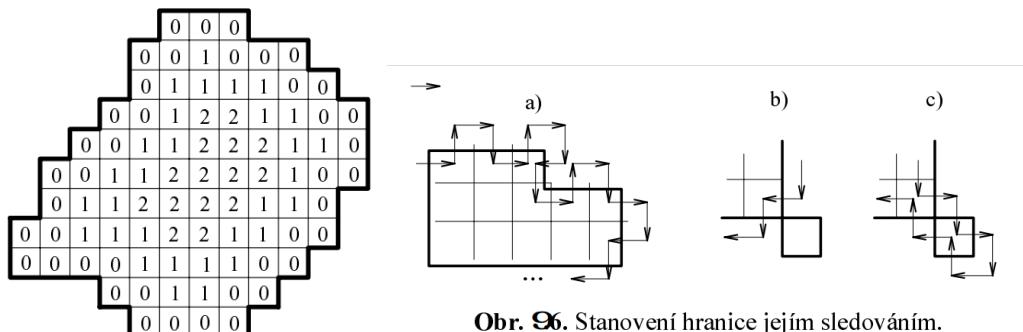
Pro kruh je $C = 4\pi$, pro čtverec $C = 16$, pro objekty nepravidelného tvaru jsou hodnoty vyšší. Tyto hodnoty jsou však teoretické a skutečná hodnota se může lišit (viz obr).



7.1.4 Průměrná vzdálenost pixelu od hranice

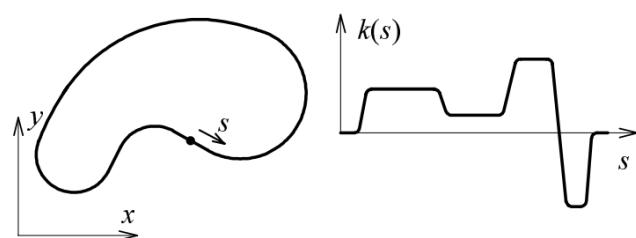
Nechť d_i je vzdálenost i -tého pixelu od hranice objektu, jehož plocha je A (vzdáleností rozumíme, kvůli rychlosti, pouze počet řad pixelů, které leží mezi pixelem a hranicí). Průměrnou vzdálenost μ_d a koncentrovanou informaci o tvaru objektu S získáme jako:

$$\mu_d = \frac{1}{N} \sum_{i=0}^N d_i, \quad S = \frac{A}{\mu_d^2}.$$



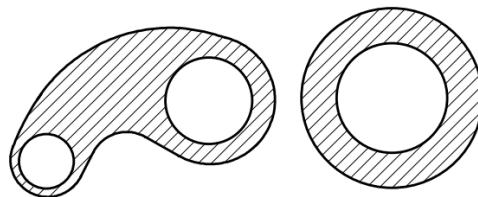
Obr. 96. Stanovení hranice jejím sledováním.

7.1.5 Průběh křivosti hranice objektu



7.1.6 Eulerovo číslo

Nechť C je počet navzájem nesouvislých částí rozpoznávaného objektu a H je počet děr v objektu pak Eulerovo číslo je rozdíl $C - H$, jelikož však většinou detekujeme jeden souvislý objekt, rovnice bude rovna $1 - H$.



Obr. 9.4. Objekt se dvěma nesouvislými částmi a třemi děrami ($C=2$, $H=3$).

7.1.7 Atributy odvozené z histogramu jasu

Možné použití, pokud rozpoznáváme objekty, pro něž je charakteristické jisté rozložení jasu po ploše objektu, nebo pro objekty pokryté texturou. Pokud je N počet pixelů plochy objektu, potom normalizovaný histogram (vydělíme počtem pixelů plochy), v podstatě představuje **pravděpodobnost** toho, že se na daném místě nachází pixel s danou hodnotou. S využitím pozorování, že funkci $p(b)$ (pravděpodobnost, že tam bude pixel) lze interpretovat jako hustotu pravděpodobnosti, lze jako **příznaky** využít **střední hodnotu**, **varianci**, **šikmost**, **energie**, **entropie**.

7.1.8 Atributy odvozené z frekvenčního spektra jasu

Hodí se např. když má objekt nějakou **texturu**. Provede se furierova transformace nad oblastí, kterou objekt zaujímá (pokud je objekt třeba kulatý, vyplníme jej menšími obdelníky a na každý uděláme furiera). Jednotlivá data frekvenčního spektra mohou sloužit jako příznaky, je však žádoucí je nějakým způsobem redukovat – zapisujeme počet frekvencí, která se nachází v nějakém definovaném **rozsahu**.

7.2 Příznakové metody analýzy obrazu (klasifikátory)

Klasifikátor nazveme zobrazení $d : X \rightarrow \Omega$, která každému vektoru příznaků (X) přiřadí identifikátor třídy, do které dané objekt náleží. Rozlišujeme několik základních druhů klasifikace:

1. **Klasifikace pomocí diskriminačních funkcí** – předpokládáme že existuje n reálných diskriminačních funkcí $g_1(x), g_2(x), \dots$ definovaných nad X . Rozpoznání je v tomto případě založeno na stanovení funkce dávající pro zadaný vektor příznaků **maximální hodnotu**. **Problém:** nalezení těchto funkcí \rightarrow , lze pomocí rozdělení hustoty pravděpodobnosti.

Nejjednodušším tvarem diskriminační funkce je funkce lineární, která definuje váhy (a) pro každý příznak:

$$g_r(x) = a_{r,0} + a_{r,1}x_1 + a_{r,2}x_2 + \dots + a_{r,m}x_m$$

2. **Klasifikace pomocí minimální vzdálenosti (etanoly)** – etanol e_r je vektor, který reprezentuje nějakou třídu příznaků. Tento vektor lze nejjednodušši získat jako **průměr** všech ostatních příznaků dané třídy trénovací množiny. Klasifikaci objektu do dané třídy pak lze získat vypočítáním a nalezením **nejmenší Euklidovské vzdálenosti** mezi vektorem příznaků objektu a jednotlivými etanoly.

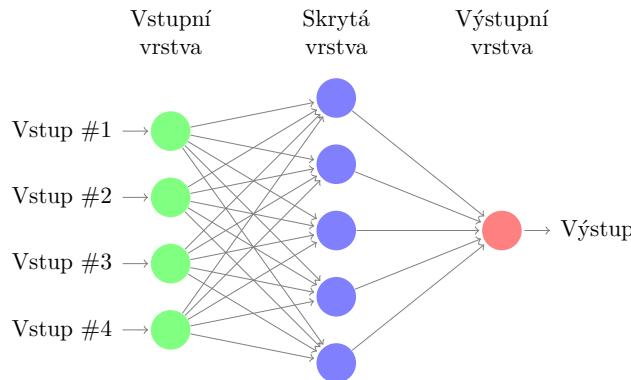
7.2.1 KNN - K-nearest neighbours

- velmi rychlý, jednoduchý (lze použít i jiné shlukovací algoritmy pro vytváření jednotlivých **etanolů**),
- pomocí **k nejbližších sousedů** určíme label dotazovaného ,
- učení je velmi rychlé (jen se uloží data do struktur),
- určení se zpomaluje se zvyšujícím se k .

7.2.2 Neuronové sítě ANN - Artificial Neural Network

- Inspirované **lidským mozkem**, který je složený z různých vzájemně propojených vrstev neuronů, kde každý z nich přijímá informaci z předchozího, zpracovává tuto informaci a odesílá ji do dalšího neuronu, dokud není přijat konečný výstup.
- Může se jednat o výstup **s danou třídou**, jestliže se jedná o kontrolované učení nebo **o určitá kritéria** v případě nekontrolovaného učení.
- Umožňuje klasifikovat více tříd.

Příklad topologie neuronové sítě je na obrázku 7.1.



Obrázek 7.1: Neuronová síť je propojená skupinou uzlů, podobná síti neuronů v mozku.

Typickým příkladem neuronové sítě je **vícevrstvý perceptron** (ANN-MLP). Tato neuronová síť se skládá minimálně ze tří vrstev uzlů (vstupní, výstupní a skrytou). Každý z uzlů je **neuron**, který využívá nelineární aktivační funkci s výjimkou vstupních uzlů:

- **Vstupní vrstva** - jedná se o pasivní vrstvu, která nemodifikuje data, pouze je získává z okolního světa a pošle je dál do sítě. Počet uzlů v této vrstvě závisí na **množství příznaků** nebo deskriptivních informací, které chceme extrahovat z obrázku.
- **Skrytá vrstva** - v této vrstvě probíhá transformace vstupů do něčeho, co může být výstupní nebo jiná skrytá vrstva využít (za předpokladu, že existuje více skrytých vrstev). Počet uzlů je určen složitostí problému a přesnosti, které chceme přidat do sítě.
- **Výstupní vrstva** - tato vrstva musí také vždy existovat v topologii sítě, ovšem počet uzlů v tomto případě bude definován vybranou neuronovou sítí. Pokud detekujeme na obrázku pouze jeden objekt, bude mít vrstva jen jeden uzel (lineární regrese) a bude vracet hodnotu definující pravděpodobnost konkrétního objektu v rozmezí $[-1, 1]$.

Učení:

1. Jednotlivé neurony jsou navzájem propojeny synapsemi (**váhami**), které na začátku náhodně nastřelíme (v intervalu $\angle -0.5, 0.5$).
2. Poté neuronové sítě podáváme trénovací data (vstupní příznaky a odpovídající výstup), která postupně **upravuje váhy** jednotlivých synapsí, aby dávala správný výsledek a minimalizovala chybu.
3. Jednou z metod minimalizace chyby je učení „**back propagation**“, což představuje hledání minima **gradientní metodou**.

Další vlastnosti:

- Vysoká dimenze vstupního vektoru zvyšuje přesnost výsledků (ovšem zvyšuje výpočetní náklady)
- Aktivační funkce pro skrytou vrstvu, která umožňuje přizpůsobit nelineární hypotézy a získat lepší detekci vzoru v závislosti na poskytnutých datech (Sigmoid, tanh, ReLU).
- Hodnoty jsou získávány z předchozí vrstvy, sečteny s určitými váhami a hodnotou zkreslení. (Suma těchto hodnot je transformována pomocí aktivační funkce, může se lišit pro různé neurony)

7.3 Histogram orientovaných gradientů HOG

Základní myšlenkou je, že objekt v obraze může být pomocí vzhledu a tvaru charakterizován pomocí intenzity gradientů, i přestože neznáme jejich přesnou polohu v obraze. Autoři jsou N. Dalal a B. Triggs (2005).

1. před započetím výpočtu je třeba normalizovat, například normalizaci barev a gamy, v případě černobílých obrázků k normalizaci kontrastu (tento krok může být přeskočen, dle Dalal a Triggs → předzpracování má malý vliv na výkon)
2. obraz se **rozdělí** na malé prostorové oblasti (buňky, například 8×8 pixelů)

3. pro každou buňku se vypočítá 1-D **histogram**, který je vypočítán ze všech pixelů z buňky (hodnoty buněk jsou rovnoměrně rozloženy do histogramu o 9 kanálech (binech) po 20° ; rozsah 0° – 180°) → výjde nám vektor o velikosti 9
4. buňky spojíme do větších **propojených bloků** 16×16 z důvodu normalizace osvětlení a kontrastu. Pro chodce se používá L2–norm normalizace, dle vztahu (7.1) → vznikne vektor velikosti $9 \times 4 = 36$ (čtyři 8×8 bloky)
5. vektor normalizovaných histogramů pro jeden blok nazýváme **deskriptor**.
6. spojíme tyto normalizované vektory do jednoho a získáme „trénovací“ vektor příznaků (features)

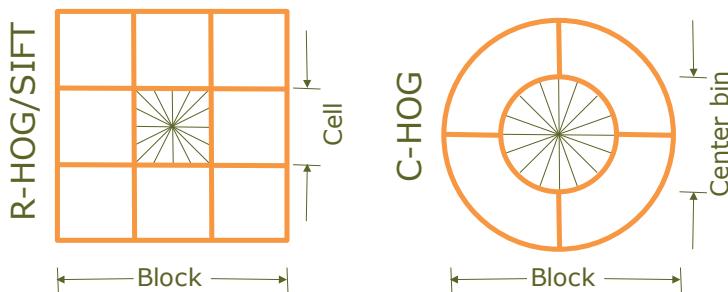
Existují dvě varianty spojení bloků, tzv. obdélníkové bloky (R–HOG) a kruhové bloky (C–HOG). Rozdělení do rozsahu 0 – 180° proto, že se jedná o bezznaménkové gradienty (unsigned) a bylo dokázáno, že fungují lépe než znaménkové (signed) 0 – 360° . Některé implementace HOG umožní určit, zda chceme používat signed gradienty.

$$\begin{aligned} L2 - \text{norm} : \quad f &= \frac{v}{\sqrt{\|v\|_2^2 + e^2}} \\ L1 - \text{sqrt} : \quad f &= \sqrt{\frac{v}{\|v\|_1 + e}} \end{aligned} \quad (7.1)$$

Nechť v je nenormalizovaný vektor obsahující všechny histogramy v daném bloku, $\|v\|_k$ je jeho k –norm pro $k = 1, 2$, a e je malá konstanta.

C–HOG (Kruhové HOG bloky) - lze nalézt ve dvou variantách: *s jedinou, centrální buňkou a úhlově rozdělenou centrální buňkou*. Dají se popsat čtyřmi parametry: počtem úhlů a radiálních kanálů (binů), poloměrem centrálního binu a faktorem roztažení pro poloměr dalších radiálních binů.

R–HOG (Obdélníkové HOG bloky) - tyto bloky jsou v praxi nejčastěji používané a reprezentují se třemi parametry: *počet buněk na blok, počet pixelů na buňku a počet binů (kanálů) na jeden histogram*. R–HOG bloky se také používají pro kódování informací.



Obrázek 7.2: Varianty geometrie spojení bloků

7.4 SIFT/SURF - detektory a popisovače klíčových bodů

Využívá se stejný princip tvoření histogramu jako u metody HOG

- klíčové body jsou **nezávislé** na osvětlení, velikosti, orientaci, pozici
- Octave - úroveň skálování
 - 4 oktávy a 5 rozmazání “ideál” dle prezentace, každá oktava se 5x rozmaže
 - vypočítá se rozdíl mezi rozmazanými obrázky
 - klíčový bod najdeme jako minimum/maximum mezi různými urovněními rozmazání
- poté musíme provést eliminaci slabých bodů
 - odebereme ty s malou intenzitou
 - odstraníme klíčové body, které leží na hraně - využít princip Harrisova detektoru hran - Hessian matice (matice druhých derivací)
- orientace bodu se vypočítá pomocí histogramu směrů gradientů v okolních bodech, zajišťuje nám to invarienci vůči rotaci (36 košů)
- Descriptor

- 16×16 matice okolo klíčových bodů
- rozdělit na 4×4 subbloky (16 bloků v 16×16 okolí)
 - * v nich vypočítat histogram orientací gradientu
 - * poté tento histogram převést do vektoru (viz HOG)
 - * spojit pro všechny bloky a máme výsledný vektor
- SURF má stejné kroky jako SIFT, jen má jiné „implementace“ kroků

7.5 Haarovy příznaky

- Na tomto přístupu je založen objektový detektor **Viola–Jones** (*Viola–Jones object detector framework*).
- Poskytuje v reálném čase **spolehlivou** a **konkurenceschopnou** detekci objektů.
- Může být vytrénován pro detekci různých objektových tříd (primárně určen pro **detekci obličejů**).
- Detektor pracuje s obrazy ve stupních šedi a skládá se ze tří částí. (Integrální obraz, Haar příznaků a AdaBoost algoritmus)

Integrální obraz je takový obraz (obrázek 7.5), kde každý bod x představuje součet hodnot předchozích pixelů doleva a nahoru. Spodní pravý bod obsahuje součet všech pixelů v obrazu. Zápis integrálního obrazu je:

$$I(x, y) = \sum_{\substack{x' \leq x \\ y' \leq y}} i(x', y'),$$

kde $i(x', y')$ je hodnota pixelu na pozici (x, y) .

1	1	1
1	1	1
1	1	1

Obrázek 7.3: Vstupní obraz

1	2	3
2	4	6
3	6	9

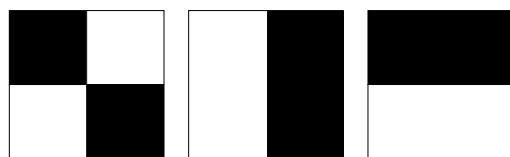
Obrázek 7.4: Integrální obraz

Obrázek 7.5: Převod obrazu na integrální obraz

Princip využití Haarových příznaků v obrazech je založen na pozorování, že lidská těla a obličeje mají některé podobné rysy. Právě tyto rysy mohou být porovnány pomocí Haarových příznaků. Jedná se například o tyto rysy:

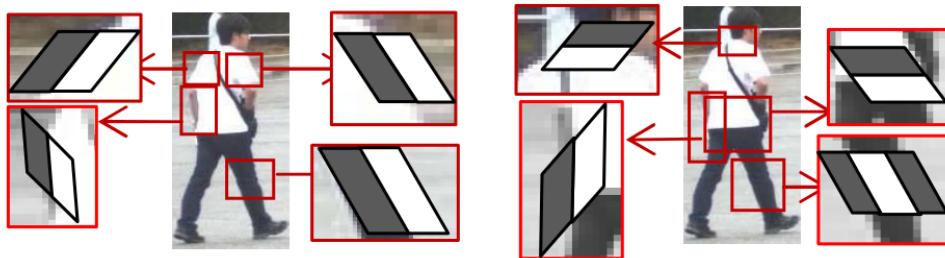
- Oční oblast je tmavší než oblast nosního mostu,
- hlava člověka je tmavší než její okolí,
- oblast mezi dolními končetinami je světlejší než samotné nohy.

Sada Haarových vlnek je na obrázku 7.6, jedná se pouze o základní sadu příznaků.



Obrázek 7.6: Základní sada Haarových příznaků

Pro identifikaci lidských postav se používá rozšířená sada vlnek, tzv. Haar-like příznaky. Klasifikační systém založený na těchto Haar-like příznacích dosahuje nižší falešně pozitivní detekce než původní Haar příznaky. Na obrázku 7.7 je příklad detekce pomocí Haar-like vlnek. **Hodnota příznaku je rozdíl mezi sumou hodnot pixelů v bílé a černé oblasti** Haarových vlnek.



Obrázek 7.7: Použití Haar-like příznaků na chodcích

7.6 LBP Lokální binární vzor

Hlavní myšlenkou LBP je, že struktury obrazu mohou být efektivně zakódovány **porovnáním hodnot jednotlivých pixelů a jejich okolí**. Tato metoda je odolná vůči jasovým změnám obrazu.

1. převod obrazu do stupňů šedi a jeho rozdělení do buněk
2. okolní hodnoty pixelů jsou porovnávány se středovým pixellem, pokud je jejich hodnota rovna nebo větší zapíše se na tuto pozici jednička v opačném případě nula
3. tyto hodnoty seřadíme dle hodinových ručiček nebo naopak a získáme osmimístné binární číslo a převedeme do dekadické soustavy
4. z čísel, které jsme získali kombinací pixelů v buňkách, vypočítáme histogram
5. zřetězíme všechny histogramy buněk a získáme vektor příznaků pro celý obraz (jedná se o 256-dimenzionální vektor příznaků)

Matematicky lze LBP vyjádřit jako:

$$LBP_{P,R} = \sum_{p=0}^P (g_p - g_c) 2^P, s(x) = \begin{cases} 1 & \text{pro } x \geq 0, \\ 0 & \text{pro } x < 0, \end{cases}$$

kde: P je počet bodů v okolí, R vyjadřuje vzdálenost bodů od středového pixelu, g_c je středový pixel, g_p je aktuální pixel.

Následující příklad se vztahuje k obrázku 7.8. Po porovnání pixelů se středovým pixellem jsme získali vzor 11110001. Tento vzor převedeme do dekadické soustavy a sečteme, $1 + 16 + 32 + 64 + 128 = 241$. **Získali jsme hodnotu této buňky do vektoru příznaků.**

6	2	2
7	6	1
9	8	7

Vstupní buňka

1	0	0
1		0
1	1	1

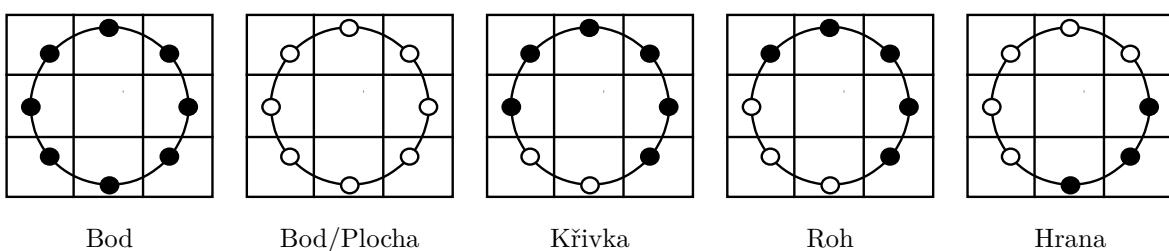
Prahové hodnoty

1	2	4
128		8
64	32	16

Pixely ohodnoceny váhou

Obrázek 7.8: Výpočet příznaku

Výhoda této metody je její rychlý a snadný výpočet a odolnost vůči různým osvětlením. Na druhou stranu je těžší na trénování, protože výsledné dekadické číslo může mít obrovské množství možností (podle parametru P). K omezení lze využít uniformní vzory (obrázek 7.9). Pro parametr $P = 8$, získáme 59 vzorů.



Obrázek 7.9: Lokální okolí LBP metody

7.7 Klasifikátory

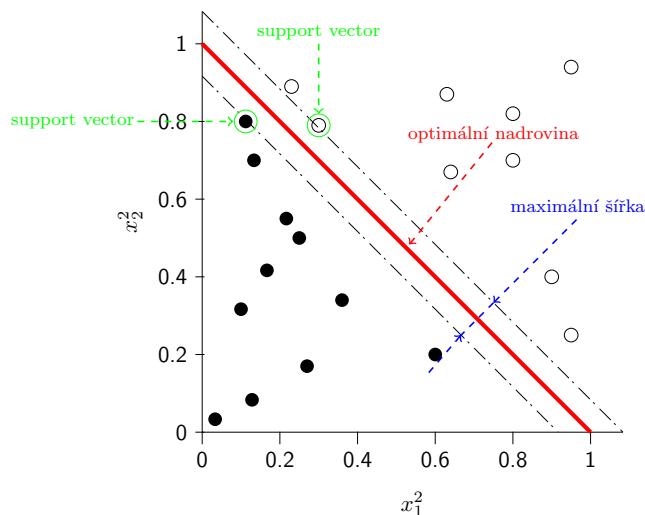
Klasifikace je obecný proces kategorizující objekty do určitých tříd. Termín klasifikátor někdy odkazuje také na matematickou funkci, implementovanou klasifikačním algoritmem.

SVM Support vector machines

- První algoritmus přisuzován Vladimíru Vapnikovi (1963)
- Učební modely, které jsou velmi populární v oblasti strojového učení.
- Založena na tzv. **jádrových algoritmech** (kernel machines) s využitím **podpůrných vektorů** (support vectors).
- Původně tato technika sloužila k vytvoření optimálního binárního klasifikátoru, později byla rozšířena na řešení **problému regrese a shlukování**.
- Byly úspěšně použity ve třech hlavních oblastech: kategorizace textu, rozpoznání obrazu a bioinformatika (např. třídění novinových zpráv, rozpoznávání ručně psaných čísel nebo například vzorky rakovinových tkání).

Primárním cílem SVM je nalézt **nadrovinu**, která **optimálně rozděluje prostor** příznaků tak, aby trénovací data náležela do konkrétních tříd. Tuto nadrovinu ilustruje obrázek 7.10. Pokud mezera mezi oddělující nadrovinou a nejbližšími vektory příznaků z obou kategorií (v případě binárního klasifikátoru) je maximální, jedná se o optimální řešení. Vektory příznaků v blízkosti této nadroviny se nazývají podpůrné vektory, což znamená, že pozice ostatních vektorů nemá vliv na nadrovinu (rozhodovací funkce).

Jinými slovy, se jedná o diskriminační klasifikátor formálně definovaný rozdělovací nadrovinou, která kategorizuje nové příklady.



Obrázek 7.10: Optimální oddělovací hranice

Implementaci SVM lze nalézt v již existujících knihovnách, jako jsou například LIBSVM, kernlab, scikit-learn, SVMLight..

- **C-Support vektorová klasifikace (C-SVC)** – Umožnuje nedokonalé oddělení tříd pro n -tříd ($n > 2$) s postihovým multiplikátorem C , pro odlehlé hodnoty ($C > 0$).
- **ν -Support vektorová klasifikace (ν -SVC)** – n -třídní klasifikace s možností nedokonalé separace. Tato klasifikace přidává nový parametr $\nu \in (0; 1)$, čím větší je jeho hodnota, tím hladší je rozhodovací funkce.
- **Distribuční odhad (Jednotřídní SVM)** – Distribution Estimation (One-class SVM), jak již název sám o sobě napovídá všechny trénovací data pocházejí z jedné třídy, SVM vytvoří hranici, která odděluje třídu od zbývající části.
- **ε -Support vektorová regrese (ε -SVR)** – Vzdálenost mezi vektory příznaků a rozdělovací nadrovinou musí být menší než mez tolerance ε . Pro odlehlé hodnoty opět použijeme multiplikátor C . Musí tedy platit: $C > 0$ a $\varepsilon > 0$.

- **ν –Support vektorová regrese (ν –SVR)** – Tato klasifikace je podobná jako ε –SVR. Na místo ε se použije parametr $\nu \in (0; 1)$.

Účinnost SVM závisí na výběru správného jádra a jeho parametrů. Často se používá Gaussovo jádro s jedním parametrem γ . Díky jeho přesnosti, ale je časově náročné. V této knihovně se můžeme setkat s následujícími jádry.

- **Lineární jádro** – Použití tohoto jádra je velmi rychlé (bez jakékoliv transformace), jedná se o lineární diskriminaci a rozdělovací nadrovina bude vždy přímka. Pro toto jádro platí

$$K(x_i, x_j) = x_i^T x_j,$$

kde x_i a x_j jsou vektory vstupního prostoru.

- **Polynomické jádro** – Polynomické jádro umožňuje učení nelineárních modelů

$$K(x_i, x_j) = (\gamma x_i^T x_j + c)^d, \gamma > 0,$$

kde: $c \geq 0$, volný parametr, který vylučuje vliv vyššího rádu oproti polynomu nižšího rádu (pokud $c = 0$, jádro je homogenní), rád polynomu určuje parametr d .

- **Gaussovo jádro** – Gaussovo neboli RBF (Radial Basis Function) jádro se řadí mezi nejpoužívanější a je definované jako

$$K(x_i, x_j) = e^{-\gamma \|x_i - x_j\|^2}, \gamma > 0,$$

kde: $\|x_i - x_j\|^2$ značí kvadratickou euklidovskou vzdálenost mezi dvěma vektory příznaků.

- **Sigmoidní jádro** – toto jádro je podobné sigmoidní funkci v logistické regresi

$$K(x_i, x_j) = \tanh(\gamma x_i^T x_j + r),$$

kde r je volitelný parametr.

- **Exponenciální jádro** – Exponenciální jádro χ^2 je podobné RBF jádru a využívá se převážně na histogramy

$$K(x_i, x_j) = e^{-\gamma \chi^2(x_i, x_j)}, \chi^2(x_i, x_j) = \frac{(x_i - x_j)^2}{(x_i + x_j)}, \gamma > 0,$$

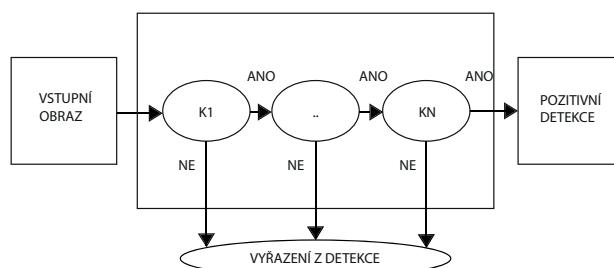
- **Jádro histogramu průsečíků** – Toto jádro je také známé jako *Min Kernel*, jedná se o nejnovější jádro v této knihovně a je velmi rychlé a užitečné při klasifikaci

$$K(x_i, x_j) = \min(x_i, x_j).$$

7.7.1 Kaskádové klasifikátory

- Skládá z více slabších klasifikátorů umístěných v **kaskádách** za sebou.
- Požadavky na tento druh klasifikátoru byly **rychlosť** detekce, aby mohl být implementován na procesorech s nižším výkonem. (v kamerách, v telefonech..)
- Klasifikátory si mezi sebou **předávají všechny** informace o vstupním obrazu (může se redukovat čas, nutný pro detekci v daném obrazu).
- Prvním takovým klasifikátorem byl detektor obličeje **Viola–Jones**

Klasifikátor na první vrstvě může vyfiltrovat většinu negativních oken. Na druhé vrstvě se mohou odfiltrovat „těžší“ negativní okna, která přežila z první vrstvy a tak dále. Subokno, které přežije všechny vrstvy, bude označeno jako pozitivní detekce. Příklad řetězce kaskádového klasifikátoru je ilustrován na obrázku 7.11, kde $K1–KN$ je klasifikátor první až n –té vrstvy.



Obrázek 7.11: Ukázka pipeline kaskádového klasifikátoru

7.7.2 AdaBoost

- AdaBoost, neboli Adaptive Boosting
- klasifikátor **kombinuje** slabé klasifikátory k vytvoření jednoho silného klasifikátoru

V kombinaci více klasifikátorů s výběrem trénovací sady v každé iteraci algoritmu a přidělení správné váhy na konci trénování, docílíme klasifikátoru s dobrou přesností. Klasifikátory v tomto řetězci, které mají klasifikační přesnost menší než 50%, jsou ohodnoceny zápornou vahou. Váhou nula jsou ohodnoceny klasifikátory, které mají přesnost 50%. Pouze ty, které mají přesnost vyšší než 50%, jsou přínosné do této kombinace a můžeme hovořit o zesílení (boosting) klasifikace.

8 Hluboké neuronové sítě (např. konvoluční, popis jednotlivých vrstev).

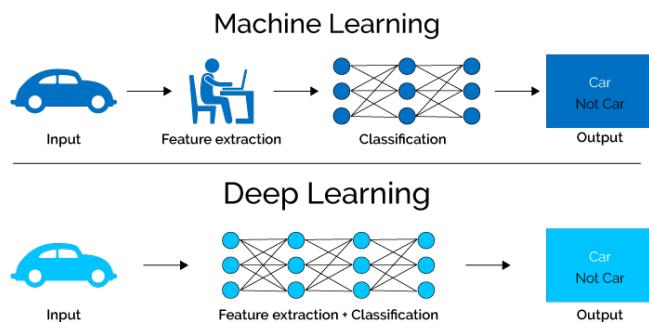
- Deep learning neboli **hluboké učení**, známé také jako hierarchické učení, je **sbírka algoritmů** používaných ve strojovém učení.
- Používají se k modelování abstrakcí na vysoké úrovni v datech za pomocí modelových architektur, které se skládají z několika nelineárních transformací.
- Hluboké učení je součástí široké skupiny metod používané pro strojové učení, které jsou založeny na učení reprezentace dat.

Hluboké strukturované učení může být:

- **Kontrolované (s učitelem)** - všechna data jsou kategorizovaná do tříd, algoritmy se učí předpovídat výstup ze vstupních dat.
- **Částečně kontrolované** - data jsou částečně kategorizovaná do tříd. Pří tomto přístupu učení lze využít kombinaci kontrolovaného a nekontrolovaného přístupu učení.
- **Nekontrolované (bez učitele)** - data nejsou kategorizovaná do tříd, algoritmy se učí ze struktury vstupních dat.

Hluboké učení je specifický přístup, používaný k budování a učení neuronových sítí, které jsou považovány za velmi spolehlivé rozhodovací uzly. Jestliže vstupní data algoritmu procházejí řadou nelinearit a nelineárních transformací, tak tento algoritmus je považován za „deep“ algoritmus.

Odstraňuje také ruční identifikaci příznaků (obrázek 8.1) z dat a místo toho se spoléhá na jakýkoliv trénovací proces, které má za úkol zjistit užitečné vzory ve vstupních příkladech. To dělá neuronovou síť jednodušší a rychlejší, a může přinést lepší výsledky než z oblasti umělé inteligence.



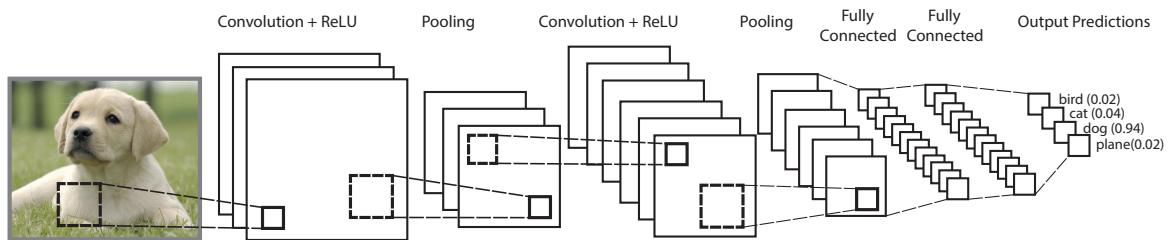
Obrázek 8.1: Hlavním rozdílem mezi strojovým a hlubokým učením je ten, že u strojového se příznaky musí extrahovat manuálně.

8.0.1 Konvoluční neuronové sítě *CNN* - Convolution neural network

- Speciálním druhem vícevrstvých neuronových sítí a jsou navrženy tak, aby rozpoznaly vizuální vzory přímo z pixelu obrazu s minimálním předzpracováním.
- Mohou rozpoznat vzory s extrémní variabilitou (například ručně psané znaky) a odolnost vůči deformacím a jednoduchým geometrickým transformacím.
- Síť využívá matematickou operaci zvanou konvoluce alespoň v jedné jejich vrstvě.

Nejznámější a nejvíce používanou konvoluční neuronovou sítí jsou modely LeNet. Hlavní kroky LeNet sítě jsou:

- **Konvoluce** - tyto vrstvy provádějí konvoluci nad vstupy do neuronové sítě.
- **Nelinearity (ReLU)** - tato vrstva je použita po každé konvoluční vrstvě a jejím cílem je nahrazení všech negativních pixelů nulou ve výstupu této vrstvy (příznaková mapa).
- **Pooling/sub sampling** - ze vstupního obrazu vyextrahuje pouze zajímavé části pomocí některých matematických operací (max, avg, sum), a tím se **redukuje jeho dimenzionalita**.
- **Fully connected layer/klasifikace** - tato vrstva vychází z původních umělých neuronových sítí, konkrétně z vícevrstvého perceptronu. Tato vrstva je typicky umístěna na konci sítě a je propojena s klasifikační vrstvou pro predikci.



Obrázek 8.2: Řetězec LeNet konvoluční neuronové sítě

9 Rekonstrukce 3D objektů z 2D obrazů (základní principy).

9.1 Structure From Motion

Metoda Structure From Motion se zabývá převodem sady snímků objektu do jeho reprezentace ve 3D. Pro získání hloubkové mapy z obrazu jsou potřeba minimálně dva snímky, pro kompletní rekonstrukci 3D modelu je poté potřeba pracovat s co největší datovou sadou. Je potřeba mít snímky vytvořené z různých úhlů, různých vzdáleností.

V podstatě se celý proces dá shrnout do následujících kroků: Po vytvoření snímků je potřeba danou datovou sadu zpracovat. Nejprve jsou v obrazu nalezeny klíčové body - tzn. nějaké body zájmu. Např. hrany, rohy... Následně je potřeba najít korespondence mezi body zájmu v jednotlivých obrazech. Po nalezení korespondencí je možné zpětně vypočítat pozici kamer vůči jednotlivým body. Výsledkem je poté mračno bodů, ze kterého lze pak approximovat povrch tělesa.

Lidé vnímají mnoho informací o trojrozměrné struktuře ve svém prostředí tím, že se kolem ní pohybují. Když se pozorovatel pohybuje, objekty kolem nich se pohybují různě, v závislosti na jejich vzdálenosti od pozorovatele. Toto je známé jako pohybová paralaxa a z této hloubky lze informace použít ke generování přesné 3D reprezentace světa kolem nich.

Hledání struktury z pohybu představuje podobný problém jako hledání struktury ze stereofonního vidění. V obou případech je třeba najít shodu mezi obrázky a rekonstrukcí 3D objektu.

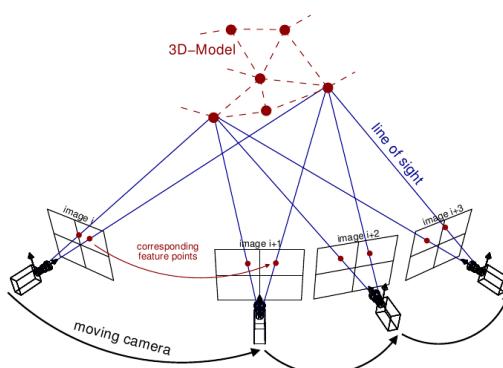
Chceme-li najít korespondenci mezi obrázky, sledují se příznaky - body zájmu jako rohové body (hrany s přechody ve více směrech) z jednoho obrázku na druhý. Jedním z nejpoužívanějších detektorů příznaků je transformace příznaků invariantních vůči měřítku (SIFT - scale-invariant feature transform). Jako rysy používá maxima z pyramidy DOG (Difference-of-Gaussians). Prvním krokem v SIFT je nalezení dominantního směru gradientu. Aby byla rotace invariantní, deskriptor se otočí tak, aby odpovídal této orientaci. Dalším běžným detektorem vlastností je SURF (speeded-up robust features). V SURF je DOG nahrazen detektorem tzv. blobů na bázi hesenské matice. Místo vyhodnocení gradientních histogramů SURF také počítá pro součty složek gradientu a součty jejich absolutních hodnot. Jeho použití integrálních obrazů umožňuje extrémně rychle detektovat příznaky s vysokou mírou detekce. Ve srovnání s SIFT je tedy SURF rychlejším detektorem vlastností s nevhodou menší přesnosti v pozicích příznaků. Dalším typem příznaku, jsou obecné křivky (např. lokální hrany s přechody v jednom směru, součást technologie známé jako pointless SfM) užitečné, když jsou bodové příznaky nedostatečné, běžné v umělých prostředích.

Příznaky detekované ze všech obrázků budou poté porovnány. Jedním z algoritmů shody, který sleduje příznaky z jednoho obrázku na druhý, je Lukas – Kanade tracker.

Někdy jsou některé z uzavřených příznaků nesprávně spárovány. Z tohoto důvodu by měly být zápasy filtrovány. RANSAC (konsensus náhodných vzorků) je algoritmus, který se obvykle používá k odstranění odlehlých korespondencí. V příspěvku Fischlera a Bollese se RANSAC používá k řešení problému určování polohy (LDP), kde cílem je určit body v prostoru, které se promítají na obraz do souboru orientačních bodů se známými místy.

Trajektorie příznaků v čase se poté použijí k rekonstrukci jejich 3D pozic a pohybu kamery. Alternativu dávají takzvané přímé přístupy, kdy jsou geometrické informace (3D struktura a pohyb kamery) přímo odhadovány z obrazů, bez mezilehlé abstrakce na příznaky nebo rohy.

Existuje několik přístupů ke struktuře z pohybu. V příručkovém SfM jsou pozice kamer řešeny a přidávány jeden po druhém do datové sady. V globálním SfM jsou pozice všech kamer řešeny současně. Kompromisem je out-of-core SfM, kde se počítá několik částečných rekonstrukcí, které jsou poté integrovány do globálního řešení.



9.2 Structured Light

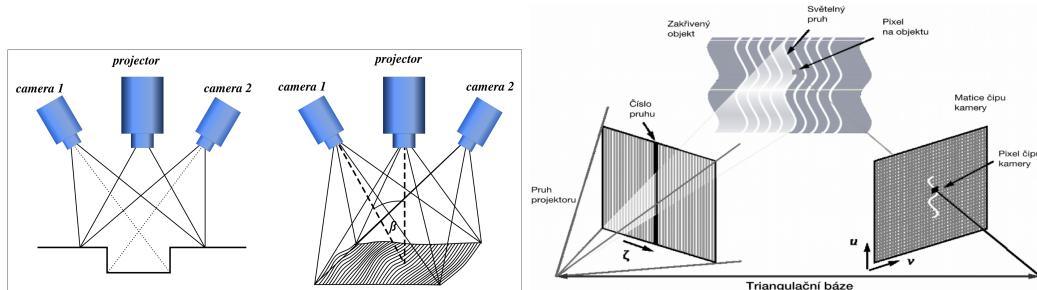
Na sledovaný objekt jsou vysílány různě velké liniové paprsky, které se zpětně snímají kamerou. Podle zakřivení povrchu je vyslaný paprsek deformován. Ze snímků deformovaných paprsků je pak možné provést triangulaci

a následně zpětně rekonstruovat povrch tělesa.

3D skener strukturovaného světla je zařízení pro měření trojrozměrného tvaru objektu užitím promítaného světelného vzoru a kamerového systému. V dnešní době je strukturované světlo běžně využíváno pro různé trojrozměrné profilometrické zkoumání povrchů díky jeho nízké cenně a vysoké rychlosti, proto je jedním z nejčastějších způsobů, jak velmi rychle a efektivně zjistit žádané informace o povrchu objektu. Měřící čidla na bázi strukturovaného světla (SL – structured light) jsou využívány v mnoha odvětvích, kromě strojírenství a kontroly kvality je to např. k uchování uměleckých děl, v zábavném průmyslu, medicíně nebo zabezpečení. Je to především pro jeho bezkonkurenční rozlišení, bezkontaktní zajištění rekonstrukce celého pole objektů ve vysokém rozlišení a rychlost. K výhodám se dá také jistě zařadit kompaktnost skeneru, jelikož měřící proces je realizován pouze profilometrickým systémem, který se skládá z jednotky pro zpracování a analýzu (PC), projekční jednotky (zpravidla videoprojektoru) a vizuální jednotky (CCD/CMOS kamera). Při zaznamenávání stojícího objektu je nutné, aby byl objekt stabilní. Využitím zařízení se strukturovaným světlem lze zaznamenat i geometrii pohybujících se těles, to se však děje na úkor snížení kvality digitalizace. Při tomto dynamickém skenování objektů se ukázalo, že základní vzory jsou nedostačující, proto začaly vznikat strukturálně složitější vzory.

9.2.1 Princip

Promítání uzkých pásů světla na trojrozměrně tvarovaný povrch vytváří linie osvětlení, které jsou zkreslené z jiného úhlu než se nachází projektor a mohou být použity pro přesnou geometrickou rekonstrukci tvaru povrchu. Používá se mnoho různých variant strukturovaného světla, ale vodorovné pruhy jsou nejčastější.



Modulovaný vzor je porovnán se vzorem promítaným, tj. porovnávají se odpovídající si pixely projektoru a snímacího zařízení. Výsledkem tohoto porovnání a využitím vhodného algoritmu se vytvoří body orientované v prostoru, tzv. mračna bodů. S těmito body lze dále pracovat a vytvořit souvislý povrch.

Část II

Matematické základy informatiky

10 Konečné automaty, regulární výrazy, uzávěrové vlastnosti třídy regulárních jazyků.

10.1 Konečné automaty (KA)

Konečný automat (KA) tvoří množina stavů, vstupní abeceda, přechodová funkce, počáteční a koncové stavы. Můžeme jej znázornit jako **tabulkou**, **graf** či **strom**.

Konečné automaty se dělí na **deterministické** a **nedeterministické**. Deterministický konečný automat má pouze jeden počáteční stav a přechodová funkce vrací jeden stav. Zatímco nedeterministický KA může mít více počátečních stavů a přechodová funkce vrací množinu stavů.

- **Slovo** přijaté automatem je taková sekvence symbolů (ze vstupní abecedy), pro kterou automat skončí v koncovém stavu.
- **Regulární jazyk** je takový jazyk (množina slov) který lze popsat konečným automatem.

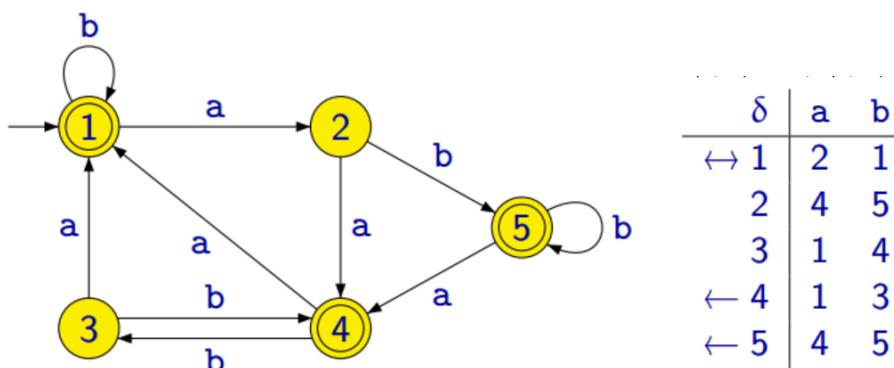
10.1.1 Deterministický konečný automat (DKA)

Skládá se ze **stavů** a **přechodů**. Jeden ze stavů je označen jako **počáteční stav** a některé jsou označeny jako **přijímací**. Je definován jako **uspořádaná pětice** $(Q, \Sigma, \delta, q_0, F)$, kde:

- Q je konečná neprázdná množina **stavů**.
- Σ (*sigma*) je konečná neprázdná množina vstupních symbolů, tzv. **vstupní abeceda**.
- δ (*delta*) je **přechodová funkce**, $\delta : Q \times \Sigma \rightarrow Q$.
- q_0 je **počáteční stav**, $q_0 \in Q$.
- F je neprázdná množina **koncových** neboli **přijímacích stavů**, $F \subseteq Q$.

Příklad

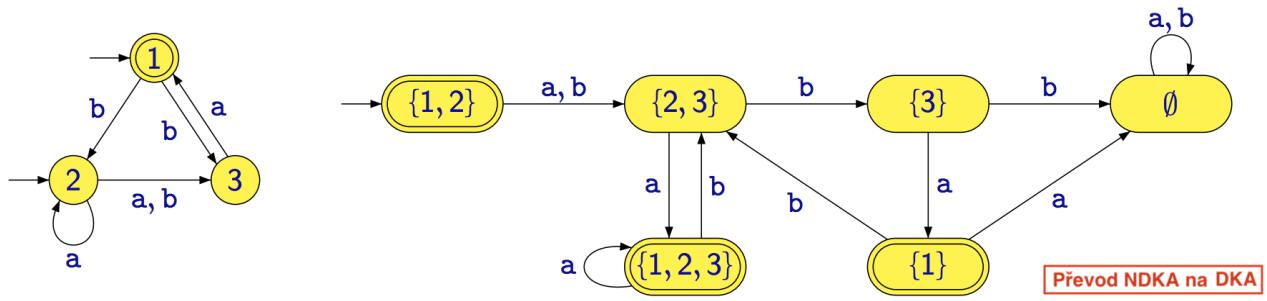
- $Q = \{1, 2, 3, 4, 5\}$, $\Sigma = \{a, b\}$, $F = \{1, 4, 5\}$
- $\delta(1, a) = 2$; $\delta(1, b) = 1$; $\delta(3, a) = 1$; $\delta(3, b) = 4$; $\delta(2, a) = 4$; $\delta(2, b) = 5$; $\delta(4, a) = 1$; $\delta(4, b) = 3$; $\delta(5, a) = 4$; $\delta(5, b) = 5$



10.1.2 (Zobecněný) Nedeterministický konečný automat ((Z)NKA)

Formálně je NKA definován jako pětice $A = (Q, \Sigma, \delta, I, F)$, s tím rozdílem, že oproti deterministickému KA má **více počátečních stavů** a **přechodová funkce vrací množinu** stavů. V případě ZNKA zde existují navíc **nulové epsilon** (ϵ) přechody:

- δ je přechodová funkce, vrací množinu stavů, $\delta : Q \times \Sigma \rightarrow P(Q)$, v případě **ZNKA** $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow P(Q)$.
- I je konečná množina počátečních stavů, $I \subseteq Q$.



Na rozdíl od deterministického automatu:

- Může z jednoho stavu vést **libovolný počet přechodů** označených stejným symbolem (i **nulové ϵ** v případě ZNKA).
- Není zde nutné, aby z každého stavu vystupovaly všechny symboly, které do něj vstoupily → **nemusí ošetřovat všechny varianty**, pouze odhadne, kterou cestou půjde.
- Nedeterministický automat přijímá dané slovo, jestliže **existuje alespoň jeden jeho výpočet**, který vede k přijetí tohoto slova.
- V automatu může být **víc než jeden počáteční stav**.
- Lze ho **převést na deterministický** (formou tabulky). Při převodu automatu, který má n stavů může mít výsledný nedeterministický až 2^n stavů.

10.1.3 Normovaný tvar

Začnu v počátečním stavu a procházím navštívené stavky a vytvářím tabulku. Každý KA má **právě 1** normovaný tvar. Také lze tímto způsobem zjistit, zda jsou automaty **ekvivalentní**.

10.2 Regulární výrazy

Regulární výraz je řetězec **popisující celou množinu řetězců**, konkrétně **regulární jazyk**. Regulární výrazy také můžeme chápat jako jednoduchý způsob, jak **popsat konečný automat** umožňující generovat všechna možná slova patřící do daného jazyka.

V regulárních výrazech využíváme znaky **abecedy** a symboly pro **sjednocení**, **zřetězení** a **iterace** regulárních výrazů. Za regulární výraz se považuje i samotný znak abecedy (např. a) stejně jako **prázdné slovo ϵ** a **prázdný jazyk \emptyset** .

10.2.1 Definice regulárních výrazů

Regulární výrazy popisují jazyky nad abecedou $A = \Sigma : \emptyset, \epsilon, a$ (kde $a \in \Sigma$) jsou regulární výrazy:

- \emptyset označuje **prázdný jazyk**,
- ϵ označuje jazyk $\{\epsilon\}$,
- a označuje jazyk $\{a\}$.

Dále, jestliže α, β jsou regulární výrazy, pak i $(\alpha + \beta)$, $(\alpha \cdot \beta)$, (α^*) jsou regulární výrazy, kde:

- $(\alpha + \beta)$ označuje **sjednocení** jazyků označených α a β ,
- $(\alpha \cdot \beta)$ označuje **zřetězení** jazyků označených α a β ,
- (α^*) označuje **iteraci** jazyka označeného α .

Neexistují žádné další regulární výrazy než ty definované podle předchozích dvou bodů.

Příklady

Ve všech případech je $\Sigma = \{0, 1\}$:

- **01** (0 a 1) ...jazyk tvořený jedním slovem 01,
- **0+1** (0 nebo 1) ...jazyk tvořený dvěma slovy 0 a 1,
- **(01)*** ...jazyk tvořený slovy $\epsilon, 01, 0101, 010101, \dots$,
- **(0+1)*** ...jazyk tvořený všemi slovy nad abecedou $\{0, 1\}$,
- **(01)*111(01)*** ...jazyk tvořený všemi slovy obsahující podslovo 111, předcházení i následované libovolným počtem slov 01,
- **(0+1)*00+(01)*111(01)*** ...jazyk tvořený všemi slovy, která budou končit 00 nebo obsahují podslovo 111 předcházené i následované libovolným počtem slov 01,
- **(0+1)*1(0+1)*** ...jazyk tvořený všemi slovy obsahujícími alespoň jeden symbol 1,
- **0*(10*10*)*** ...jazyk tvořený všemi slovy obsahujícími sudý počet symbolů 1.

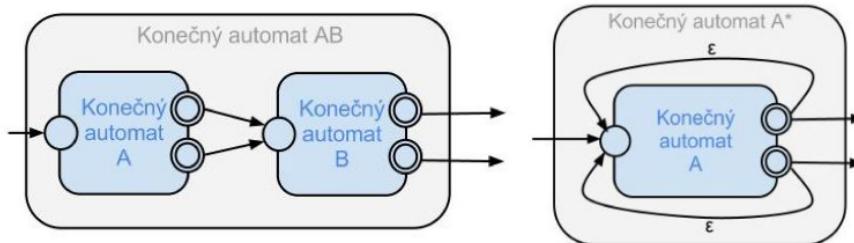
10.3 Uzávěrové vlastnosti třídy regulárních jazyků

Uzavřenost množiny nad operací znamená, že výsledek operace s libovolnými prvky z množiny bude opět spadat do dané množiny. Třídu regulárních jazyků značíme **REG**. Regulární výrazy (tedy i KA) jsou uzavřené vůči operacím:

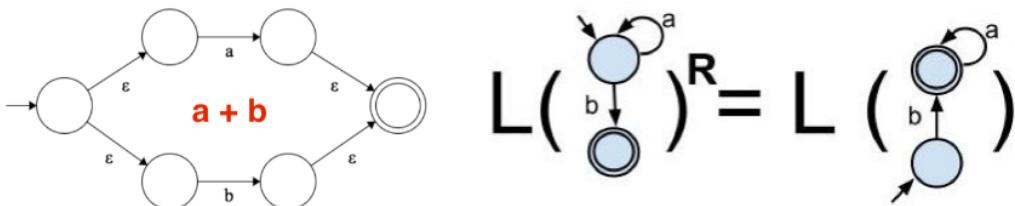
- **Sjednocení, průnik, doplněk** – je-li $L_1, L_2 \in \text{REG}$, pak také $L_1 \cup L_2, L_1 \cap L_2, L_1'$ jsou v REG .
- **Zřetězení, iterace** – je-li $L_1, L_2 \in \text{REG}$, pak také $L_1 \cdot L_2, L_1^*$ jsou v REG .
- **Zrcadlový obraz** – je-li $L \in \text{REG}$, pak také L^R jsou v REG .

10.3.1 Operace sjednocení, zřetězení, iterace a zrcadlový obraz u KA

- **Iterace – spojíme koncové stavy** jednoho KA s **počátečními** druhého KA ϵ přechodem. Na obrázku generuje automat A^* jazyk $L(A^*) = L(A)^*$, který je iterací jazyku generovaného modrého automatu A .
- **Zřetězení – spojíme koncové stavy jednoho s počátečními stavami druhého**. Na obrázku generuje konečný automat AB jazyk $L(AB) = L(A) \cdot L(B)$.



- **Sjednocení** – $L(A + B) = L(A) + L(B)$ získáme tak, že vytvoříme **nový počáteční stav**, ze kterého vedeme ϵ přechody do počátečních stavů obou automatů. Poté obdobě z koncových stavů obou automatů vedeme ϵ přechody do **nového koncového**.
- **Zrcadlový obraz** – pustíme automat pozpátku, celý jej převrátíme. **Přehodíme orientaci všech přechodů**, z počátečních stavů uděláme koncové a naopak.



- **Doplněk** – u DKA provedeme prohození označení příjmajících a ostatních stavů, u NKA je nejprve nutné provézt převod na DKA.

11 Bezkontextové gramatiky a jazyky. Zásobníkové automaty, jejich vztah k bezkontextovým gramatikám.

11.1 Bezkontextové gramatiky (BG)

Bezkontextová gramatika definuje **bezkontextový jazyk**. Je tvořena **neterminály** (proměnné), **terminály** (konstanty) a **pravidly**, které každému neterminálu definují přepisovací pravidla. Jeden neterminál označíme jako **startovní**, kde začínáme a podle pravidel je dál přepisujeme na výrazy složené z terminálu a neterminálu. Jakmile už není co přepisovat, výraz obsahuje už jen neterminály, získali jsme **slovo**.

- Je **uzavřená** vůči operacím **sjednocení**, **zřetězení**, **iteraci** a **zrcadlový obraz**.
- Ke každé bezkontextové gramatice existuje **ekvivalentní zásobníkový automat**.

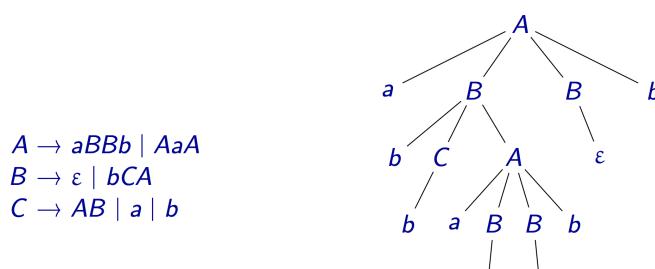
11.1.1 Formální definice BG

Bezkontextová gramatika je definována jako uspořádaná čtveřice $G = (\Pi, \Sigma, S, P)$, kde:

- Π (*velké pí*) je konečná množina **neterminálních** symbolů (neterminálů).
- Σ je konečná množina **terminálních** symbolů (terminálů), $\Pi \cap \Sigma = \emptyset$.
- S je **počáteční neterminál**, $S \in \Sigma$.
- P je konečná množina **přepisovacích pravidel**, $P \subseteq \Pi \times (\Pi \cup \Sigma)^*$.

11.1.2 Základní pojmy

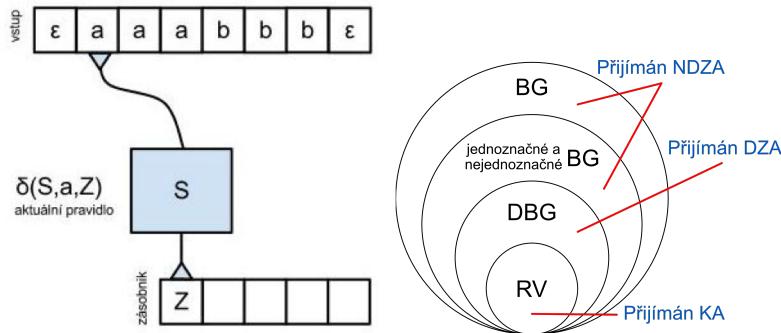
- **Bezkontextový jazyk** – formální jazyk, který je akceptovaný nějakým zásobníkovým automatem.
- **Derivace slova** – jedno konkrétní odvození slova pomocí gramatiky, tedy záznam postupných přepisů od startovního neterminálu po konečné slovo. Derivace se podle postupu při přepisování dělí na:
 - **levou** – přepisujeme nejprve levé neterminály,
 - **pravou** – přepisujeme nejprve pravé neterminály.
- **Derivační strom** – grafické znázornění derivace slova stromem. Pro všechny možné derivace (levou, pravou, moji) by měl derivační strom být **stejný**. Není-li tomu tak jedná se o **nejednoznačnou gramatiku**, což je nežádoucí jev.
 - **Špatně** = $A \rightarrow A \mid \epsilon$ (lze generovat až N způsoby), **Správně** = $A \rightarrow \epsilon$
- **Chomského normální forma** – gramatika může obsahovat pouze pravidla typu: $A \rightarrow BC$ nebo $A \rightarrow a$ nebo $S \rightarrow \epsilon$ (pokud gramatika generuje pouze prázdný řetězec).
- **Nevypouštějící gramatika** – neobsahuje ϵ (*epsilon*) přechody.



$A \Rightarrow aBBb \Rightarrow abCABb \Rightarrow abCaBBbBb \Rightarrow abCaBbBb \Rightarrow abbaBbBb \Rightarrow abbaBbb \Rightarrow abbabb$

11.2 Zásobníkové automaty (ZA)

Slouží k **rozpoznání bezkontextových jazyků**. S využitím zásobníků si může pamatovat kolik a jaké znaky přečetl, což je potřeba právě k rozpoznání bezkontextového jazyka. Zásobníkový automat je v podstatě konečný automat rozšířený o zásobník.



- ZA na základě **aktuálního znaku** na pásce, **prvního znaku v zásobníku** a **aktuálního stavu** změní svůj stav a **přepíše** znak v zásobníku podle daných pravidel.
- ZA **přijímá** dané slovo, jestliže skončí v konfiguraci (q, ϵ, ϵ) , tedy když se přečte celé vstupní slovo a zásobník je **prázdný**.
- Konfigurace** je dána: aktuálním stavem, obsahem pásky a obsahem zásobníku.
- Deterministický** – nesmí se objevit dvě pravidla se stejnou levou stranou, a pokud existuje pravidlo (q, ϵ, X) , tak nesmí zároveň existovat pravidlo (q, a, Y) . Pokud se deterministický ZNKA ocitne ve stejné konfiguraci více než jednou, tak obsahuje nekonečný cyklus.

11.2.1 Formální definice zásobníkového automatu

Zásobníkový automat M je definován jako šestice $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$, kde:

- Q je konečná neprázdná množina **stavů**.
- Σ je konečná neprázdná množina **vstupních symbolů** (vstupní abeceda).
- Γ (velká gamma) je konečná neprázdná množina **zásobníkových symbolů**.
- δ je **přechodová funkce** (konečná množina instrukcí), $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow P_{fin}(Q \times \Gamma^*)$.
- q_0 je **počáteční stav**, $q_0 \in Q$.
- Z_0 je **počáteční zásobníkový symbol**, $Z_0 \in \Gamma$.

11.2.2 Definice instrukcí (pravidel) v ZA

Instrukce (sady instrukcí reprezentují přechodovou funkci δ) definují **chování automatu**:

$$(q, a, X) \rightarrow (q', \alpha), \text{ kde } a \in \Sigma. \quad (11.1)$$

Tato instrukce je aplikovatelná jen v situaci (neboli konfiguraci), kdy **řídicí jednotka** je ve stavu q , **čtecí hlava** na vstupní pásce čte symbol a a na vrcholu zásobníku je symbol X . Pokud je **instrukce aplikována**, vykoná se následující:

- řídicí jednotka **přejde do stavu** q' ,
- čtecí hlava na vstupní pásce se **posune o jedno políčko doprava**,
- vrchní symbol v zásobníku se **odebere** (vymaže),
- na vrchol zásobníku se přidá** řetězec α tak, že jeho nejlevější symbol je aktuálním vrcholem zásobníku.

Pravidlo	Akce (Z = zásobník)	Význam
$\delta(q_1, a, X) \rightarrow (q_1, YX)$	přidání prvku do Z	na začátek zásobníku se vloží Y
$\delta(q_1, a, X) \rightarrow (q_1, Y)$	přepsání prvku v Z	první prvek zásobníku se přepíše na Y
$\delta(q_1, a, X) \rightarrow (q_1, \epsilon)$	smazání prvku ze Z	první prvek zásobníku se smaže neboli nahradí prázdným slovem ϵ
$\delta(q_1, a, X) \rightarrow (q_2, X)$	změna stavu	stav q_1 se změní na stav q_2
$\delta(q_1, a, X) \rightarrow \emptyset$	pád automatu	ukončení výpočtu, slovo nebylo přijato

11.3 Převod BG na zásobníkový automat

Využívá se tzv. metody shora-dolů, která obsahuje pouze **1 stav**:

1. pro všechny **neterminály** vypíšu pravidla typu: $(q, \epsilon, A) \rightarrow \{(q, B), (q, C)\}$,
2. všechny **terminály** přepíšu na pravidla typu: $(q, a, a) \rightarrow (q, \epsilon)$.

Vstupní gramatika:

$$S \rightarrow A|B$$

$$A \rightarrow a$$

$$B \rightarrow (c)$$

$$\Sigma = \{A, B, S\}$$

$$\Gamma = \{a, c, (,)\}$$

Instrukce, převedené dle výše uvedených pravidel:

$$(Q, \epsilon, S) \rightarrow \{(q, A), (q, B)\}$$

$$(Q, \epsilon, A) \rightarrow (q, a)$$

$$(Q, \epsilon, B) \rightarrow (q, (c))$$

$$(Q, a, a) \rightarrow (q, \epsilon)$$

$$(Q, (, ()) \rightarrow (q, \epsilon)$$

$$(Q, c, c) \rightarrow (q, \epsilon)$$

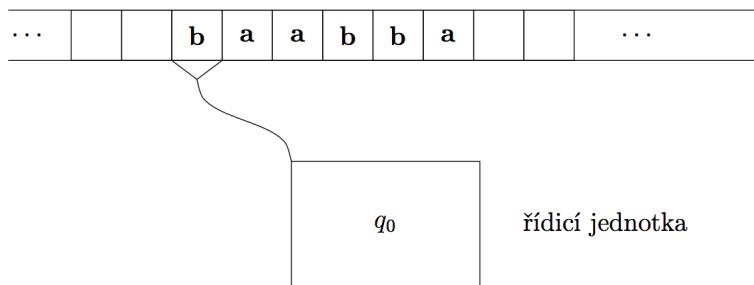
$$(Q, (,)) \rightarrow (q, \epsilon)$$

12 Matematické modely algoritmů -Turingovy stroje a stroje RAM. Složitost algoritmu, asymptotické odhady. Algoritmicky nerozhodnutelné problémy.

Ve snaze **popsat jakýkoliv algoritmus** si vymysleli matematici Turingovy a RAM stroje. Jde o dva různé přístupy (modely) univerzálních počítačů/programovacích jazyků. Jinými slovy těmito stroji lze **definovat a provést libovolný algoritmus**.

Historicky prvním „univerzálním programovacím jazykem“ byl Turingův stroj. Byl popsán dříve, ještě před rozmachem počítačů, proto se od reálného počítače (programování) podstatně liší, na rozdíl od RAM stroje. Turingův stroj například pracuje s **celou abecedou** zatímco RAM (podobně jako počítač) s **čísly**.

12.1 Turingův stroj (TS)



Turingův stroj je podobný konečnému automatu, ale má **oboustranně nekonečnou pásku** (je na ni zapsáno vstupní slovo), místo symbolu ϵ pro prázdné znaky se používá \square , **hlava** je **čtecí i zapisovací** a pohybuje se po pásmu v **obou směrech**.

12.1.1 Formální definice TS

Turingův stroj je definován jako šestice $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, kde:

- Q je konečná neprázdná množina **stavů**.
- Σ je konečná neprázdná množina **vstupních symbolů** (vstupní abeceda).
- Γ je konečná neprázdná množina **páskových symbolů**, kde $\Sigma \subseteq \Gamma$ a $\Gamma - \Sigma$ je (přinejmenším) speciální znak \square (prázdný znak [Blank]).
- δ je přechodová funkce, $\delta : (Q - F) \times \Gamma \rightarrow Q \times \Gamma \times \{-1, 0, +1\}$.
- q_0 je **počáteční stav**, $q_0 \in Q$.
- F je množina **konecových stavů**, $F \subseteq Q$.

12.1.2 Definice instrukcí (pravidel) v TS

Podobně jako ZA lze konkrétní Turingův stroj zadat seznamem instrukcí. Tyto instrukce jsou opět dány přechodovou funkcí, význam instrukce: $(q, a) \rightarrow (q', a', m)$ je tento:

$$(\text{akt. stav } [q], \text{ znak na pásmu } [a]) \rightarrow (\text{nový stav } [q'], \text{ nový znak } [a'], \text{ posun } [\{-1; 0; +1\}])$$

12.1.3 Příklad

Tento příklad invertuje slovo, které je uvedené na úvodním obrázku u TS:

$$\begin{array}{ll} Q = \{q_1, q_2\} & (q_1, a) \rightarrow (q_1, b, +) \\ \Sigma = \{a, b, c, \square\} & (q_1, b) \rightarrow (q_1, a, +) \\ q_0 = q_1 & (q_1, \square) \rightarrow (q_2, \square, 0) \\ F = \{q_2\} & \end{array}$$

12.1.4 Modifikace TS

- **N-páskový TS** – čte a zapisuje do více pásek najednou, jediná změna je v přechodové funkci: $\delta : Q \times \Gamma^n \rightarrow Q \times (\Gamma \times \{L, R, N\})^n$.
- **N-hlavový TS** – má více čtecích hlav než klasický TS, každá hlava zapisuje/čte a pohybuje se **nezávisle na ostatních**.
- **Nedeterministický TS** – umožňuje výběr z více možností, pro jednu konfiguraci můžeme definovat **více pravidel**.

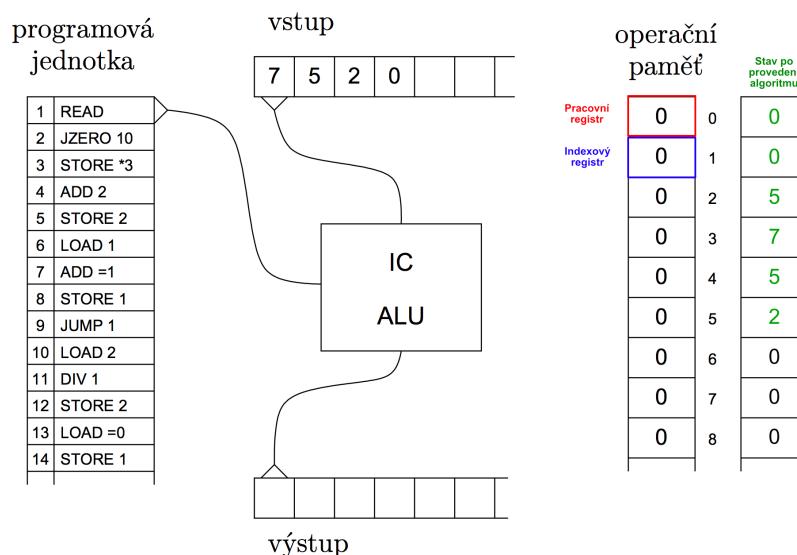
12.1.5 Základní pojmy

- **Turingovsky úplný** – stroj (počítač, programovací jazyk, úloha, ...), která má stejnou výpočetní sílu jako TS. Lze v něm **odsimulovat** libovolný jiný TS zadaný na vstupu.
- **Church-Turingova teze** – říká, že jakýkoliv výpočet lze úspěšně uskutečnit algoritmem běžícím na počítači, tedy „ke každému algoritmu existuje ekvivalentní TS“.

12.2 Model RAM (Random Access Machine)

RAM stroje již vycházejí ze skutečných počítačů, dá se tedy říct, že se jedná o jednoduchou abstrakci reálného procesoru s jeho strojovým kódem pracujícím s lineárně uspořádanou pamětí. Tento model slouží zejména k analýze algoritmů z hlediska (**paměťové, časové**) **složitosti**. Skládá se z těchto částí:

1. **Programová jednotka** – uchovává program, tvořený konečnou posloupností instrukcí.
2. **Neomezená pracovní paměť** – neomezená lineárně uspořádaná paměť, tvořená buňkami, do které lze zapisovat/číst celá čísla (\mathbb{Z}), adresovaná přirozenými čísly (\mathbb{N}) (0 = **pracovní** registr, 1 = **indexový** registr).
3. **Vstupní a výstupní páska** – lze na ně sekvenčně zapisovat/číst celá čísla (\mathbb{Z}).
4. **Centrální jednotka** – obsahuje programový register ukazující, která instrukce má být provedena. Ta se provede a programový registr se příslužně změní (zvýší o 1, o více v případě skoku).



Výše uvedený program vypočítá **aritmetický průměr**, který následně uloží do buňky paměti pod indexem č. 2. Výsledek po dělení je roven 4, 666 a po zaokrouhlení 5.

12.2.1 Instrukce a typy operandů RAM

Typ	Hodnota operandu
$= i$	přímo číslo udané zápisem i
i	číslo obsažené v buňce s adresou i
$*i$	číslo v buňce s adresou $i + j$, kde j je aktuální obsah indexového registru

Zápis	Význam
READ	do pracovního registru (PR) se načte vstup a hlava se posune doprava
WRITE	na výstup se zapíše hodnota PR
LOAD <i>op</i>	do PR se načte hodnota dána operátorem <i>op</i>
STORE <i>op</i>	hodnota PR se uloží na do registru daného operátorem <i>op</i>
ADD <i>op</i>	k hodnotě PR se přičte hodnota daná operátorem <i>op</i>
SUB <i>op</i>	od hodnoty v PR se odečte hodnota daná operátorem <i>op</i>
MUL <i>op</i>	PR se vynásobí hodnotou danou operátorem <i>op</i>
DIV <i>op</i>	PR se vydělí hodnotou danou operátorem <i>op</i>
JUMP <i>návěští</i>	provede se skok na instrukci danou <i>návěštím</i>
JZERO <i>návěští</i>	pokud je hodnota v PR rovna 0 , provede se skok na <i>návěští</i>
JGTZ <i>návěští</i>	pokud je hodnota v PR větší než 0 , provede se skok na <i>návěští</i>
HALT	korektní ukončení programu

12.3 Složitost algoritmů

Abychom mohli **porovnávat** různé algoritmy řešící stejný problém, zavádí se pojem **složitost algoritmu**. Složitost je jinak řečeno **náročnost algoritmu** – čím menší složitost tím je algoritmus lepší. Přičemž nás může zajímat složitost z pohledu času, či **paměti**:

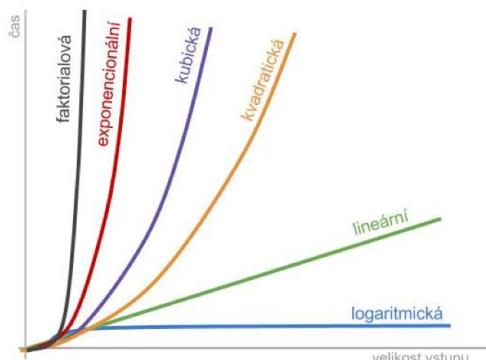
- **Časová složitost** – sleduje jak závisí **doba** výpočtu alg. na množství vstupních dat.
- **Prostorová složitost** – sleduje jak závisí **množství použité paměti** výpočtu alg. na množství vstupních dat.

Jelikož konkrétní čísla (čas, paměť) se liší v **závislosti vstupních datech**, množství zpracovávaných dat a použitém programovacím jazyku, neudává se složitost čísla, nýbrž **funkcí závislou na velikosti vstupních dat**. Tato funkce se získá počítáním proběhlých instrukcí algoritmu sestaveném v univerzálním RAM stroji. A počítá se s nejhorším možným případem vstupu. To je důležité například u třídicích algoritmů, kde hraje velkou roli to, jak moc už je vstupní pole setříděné (vstupuje-li do algoritmu už setříděná posloupnost čísel, algoritmus skončí okamžitě, zatímco s opačně seřazenými čísly se bude trápit dloho).

12.4 Asymptotická notace

Je **způsob klasifikace počítačových algoritmů**. Ve většině případů nemusíme znát přesný počet provedených instrukcí a spokojíme se pouze s odhadem toho, jak rychle tento počet narůstá se zvyšujícím se vstupem. Asymptotická notace nám umožní **zanedbat méně důležité detaily** a **odhadnout** přibližně, **jak rychle daná funkce roste**. V souvislosti s asymptotickými odhady složitosti se používají tyto zapisy:

- $f \in O(f)$ – f roste **nejvýše tak rychle** jako g (f je ohraničena g shora) $[\leq]$.
- $f \in o(g)$ – f roste **(striktně) pomaleji** než g (f je ohraničena g shora ostře) $[<]$.
- $f \in \Theta(g)$ – f roste **stejně rychle** jako g $[=]$.
- $f \in \omega(g)$ – f roste **(striktně) rychleji** než g (f je ohraničena g zdola ostře) $[>]$.
- $f \in \Omega(g)$ – f roste **rychleji** než g (f je ohraničena g zdola) $[\geq]$.



Seřazeno podle složitosti:

- $f(n) \in \Omega(\log n)$ – logaritmická funkce (složitost),
- $f(n) \in \Omega(n)$ – lineární funkce (složitost),
- $f(n) \in \Omega(n^2)$ – kvadratická funkce (složitost),
- $f(n) \in O(n^k)$ pro nějaké $k > 0$ – polynomiální,
- $f(n) \in \Omega(k^n)$ pro nějaké $k > 1$ – exponenciální.

12.4.1 Úskalí asymptotické notace

Při používání asymptotických odhadů časové složitosti je třeba si uvědomit některá úskalí:

- Asymptotické odhady se týkají pouze toho, **jak roste čas s rostoucí velikostí vstupu** → neríkají nic o **konkrétní době výpočtu**. V asymptotické notaci mohou být **skryty velké konstanty**.
- Algoritmus, který má lepší asymptotickou časovou složitost než nějaký jiný algoritmus, **může být ve skutečnosti rychlejší** až pro nějaké hodně velké vstupy.
- Většinou analyzujeme složitost v **nejhorším případě**. Pro některé algoritmy může být doba výpočtu v nejhorším případě mnohem větší než doba výpočtu na „typických“ instancích (typicky Quicksort → nejhorší: $O(n^2)$, průměrná: $O(n \log n)$).

12.5 Algoritmicky nerozhodnutelné problémy

Rozhodovací problém je rozhodnutelný (řešitelný) pokud pro libovolný vstup z množiny vstupů, skončí algoritmus svůj výpočet a vydá správný výstup (tedy jestliže **existuje turingův stroj, který jej řeší**).

Pokud nalezneme takový vstup, pro který všechny dosavadní algoritmy nejsou schopny nalézt výstup, můžeme tento problém označit za **nerozhodnutelný**. Speciální případ jsou **doplňkové problémy**, které vracejí přesně opačné výsledky než původní problém.

12.5.1 Definice problému

Problém je určen **trojicí** (IN, OUT, p) , kde:

- IN je množina (přípustných) **vstupů**,
- OUT je množina **výstupů**,
- $p : IN \rightarrow OUT$ je **funkce** přiřazující každému vstupu odpovídající výstup.

12.5.2 Ano/Ne problémy

Jsou to problémy, jejichž **výstupní množina obsahuje dva prvky** $OUT = \{\text{ano}, \text{ne}\}$. Na ano/ne problémy se dají převést ostatní problémy nepotřebujeme-li znát přesný výsledek:

- Nepotřebuji najít v poli nejmenší číslo, stačí mi vědět **zda pole obsahuje číslo menší než nula**.
- Nepotřebuji znát nejkratší cestu grafem, stačí mi najít cestu, **která je kratší než 8**.

12.5.3 Riceova věta

Tato věta ukazuje nerozhodnutelnost celé třídy problémů, její znění je následující „*Každá netriviální vstupně/výstupní (I/O) vlastnost programů je nerozhodnutelná*“.

- Vlastnost **X je vstupně/výstupní** právě tehdy, když každé dva programy se stejnou I/O tabulkou budou vlastnost **X** mají nebo ji oba nemají.
- Připomeňme tedy ještě, že vlastnost **V je triviální**, když ji mají buď všechny programy nebo ji nemá žádný program; taková vlastnost je podle definice také **vstupně/výstupní**.

Problém	1	2	3
Je triviální?	A	N	N
Je I/O?	A	A	N
Je nerozhodnutelný	N	A	?

12.5.4 Částečná rozhodnutelnost

Částečně rozhodnutelný problém, je takový problém, pro který jsme v případě vstupů, u nichž očekáváme odpověď ANO, schopni vrátit odpověď ANO, a v případě NE vrátit buď NE nebo \perp (program se nezastaví a nejsme schopni zjistit, zda by odpověď byla opravdu NE).

12.5.5 Převeditelnost mezi nerozhodnutelnými problémy

Důkaz neřešitelnosti lze provést skrze jiné, **už dokázané**, problémy. Řekneme, že problém P_1 je převeditelný na problém P_2 (značíme $P_1 \rightsquigarrow P_2$), jestliže alg., který k instanci I_1 problému P_1 sestrojí instanci I_2 problému P_2 tak, že odpověď P_1, I_1 je stejná jako P_2, I_2 . Např.: DHP je převeditelný na HP. Z toho vyplývá, že pokud P_1 je nerozhodnutelný tak i P_2 je **nerozhodnutelný**.

12.5.6 Příklady nerozhodnutelných problémů

1. Eq-CFG (Ekvivalence bezkontextových gramatik)

- **Vstup:** Dvě bezkontextové gramatiky $G1, G2$.
- **Otzáka:** Platí $L(G1) = L(G2)$? Generují obě gramatiky stejný jazyk?

2. HP (Problém zastavení [Halting Problem])

- **Vstup:** Turingův stroj M a jeho vstup w .
- **Otzáka:** Zastaví se M na w (tzn. je výpočet stroje M pro vstupní slovo w konečný)?

12.6 Optimalizační problémy

Optimalizační problémy **hledají nejlepší řešení** v množině různých řešení. Příkladem je například: hledání nejkratší cesty, nejmenší kostry, apod.

12.6.1 Příklady optimalizačních problémů

1. Hledání nejkratší cesty v grafu

- **Vstup:** Orientovaný graf $G = (V, E)$ a dvojice vrcholů $u, v \in V$.
- **Výstup:** Nejkratší cesta z u do v .

2. Hledání minimální kostry v grafu

- **Vstup:** Neorientovaný souvislý graf $G = (V_G, E_G)$ s ohodnocenými hranami.
- **Výstup:** Souvislý graf $H = (V_H, E_H)$, kde $V_H = V_G$ a $E_H \subseteq E_G$, který má součet hodnot všech hran minimální.

13 Třídy složitosti problémů. Třída PTIME a NPTIME, NP-úplné problémy.

13.1 PTIME (PSPACE)

Třída všech problémů, které lze řešit algoritmy s **polynomiální časovou (prostorovou) složitostí**, tj. s časovou složitostí $O(n^k)$, kde k je nějaká konstanta.

Tuto třídu problému považujeme za zvládnutelnou. Je **robustní** – mezi jednotlivými výpočetními modely (RAM, TS) existují vzájemné polynomické simulace, nezáleží tedy jakým modelem budeme algoritmus simulovat, vždy bude patřit do třídy PTIME.

13.1.1 Problémy pařící do třídy PTIME

- Třídění a vyhledávání.
- Nejkratší cesta v grafu a minimální kostra grafu.
- Ekvivalence deterministických konečných automatů.
- Přijatelnost slova bezkontextovou gramatikou.

1. Výběr aktivit

- **Vstup:** Množina aktivit s časovými intervaly, kdy je lze vykonávat.
- **Výstup:** Největší možný počet kompatibilních aktivit (aktivit, které se nekryjí).

2. Optimalizace násobení řetězce matic

- **Vstup:** Posloupnost matic.
- **Výstup:** Plně uzávorkovaný součin.

3. LCS - problém nejdelší společné posloupnosti

- **Vstup:** Dvě posloupnosti v, w v nějaké abecedě Σ .
- **Výstup:** Nejdelší společná podposloupnost posloupností v, w .

13.2 NPTIME

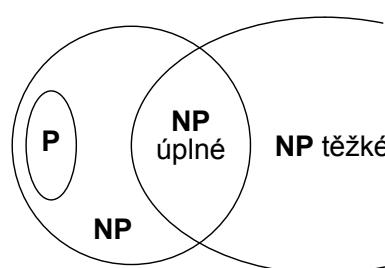
Třída všech rozhodovacích problémů (ano/ne), které jsou rozhodovány **nedeterministickými algoritmy s polynomiální časovou složitostí**.

Pokud je odpověď ANO, tak stačí nalézt jedno řešení s touto odpovědí. Pokud je odpověď NE, je potřeba ukázat, že žádné řešení nevrací ANO.

13.3 Třída NP-úplných problémů

NP-úplné problémy jsou takové problémy, na které jsou **polynomiálně redukovatelné všechny ostatní problémy z třídy NP**. To znamená, že třídu NP-úplných úloh tvoří v jistém smyslu ty **nejtěžší úlohy** z NP.

Pokud by byl nalezen deterministický polynomiální algoritmus pro nějakou NP-úplnou úlohu, znamenalo by to, že všechny nedeterministicky polynomiální problémy jsou řešitelné v polynomiálním čase (tedy že $NP = P$). Otázka, zda nějaký takový algoritmus existuje, zatím nebyla rozhodnuta, předpokládá se však, že $NP \neq P$ (je však zřejmé, že $P \subseteq NP$).



13.3.1 NP-těžký problém

Problém P nazveme NP-těžkým, pokud pro libovolný problém A ze třídy NP platí, že A je polynomickně převeditelné (redukované) na problém P , tedy pokud platí: $\forall P \in \text{NP-ptime} : P \triangleright Q$.

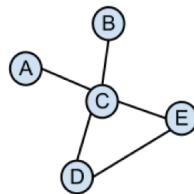
13.3.2 NP-úplný problém

Třída nejtěžších NPTIME problémů. Problém P je NP-úplný, pokud patří do třídy **NP** a je **NP-těžký**.

13.3.3 NP-úplné problémy

1. IS (problém nezávislé množiny)

- **Vstup:** Neorientovaný graf G (o n vrcholech), číslo k ($k \leq n$).
- **Otzáka:** Existuje v G nezávislá množina velikosti k (tj. množina k vrcholů, z nichž žádné dva nejsou spojeny hranou)?



Program vybere náhodně k vrcholů z grafu a ověří zdali nejsou některé spojeny hranou. Když mezi nimi hranu **nenaře**, vrátí odpověď „ANO, v grafu existuje nezávislá množina o velikosti k “. Když hranu mezi zvolenou množinou **najde**, vrátí odpověď „NE“. Přičemž odpověď ne **může být chybná**.

Výstup pro $k > 3$: NE.

Výstup pro $k = 3$: ANO (ABD, ABE).

Výstup pro $k = 2$: ANO (AB, AD, AE, BD, BE).

2. Isomorfismus grafů

- **Vstup:** Dva neorientované grafy G a H .
- **Otzáka:** Jsou grafy G a H izomorfní?

3. CG (Barvení grafu)

- **Vstup:** Neorientovaný graf G a číslo k .
- **Otzáka:** Je možné graf G barvit k barvami (tj. existuje přiřazení barev vrcholům tak, aby žádné dva sousední vrcholy nebyly barveny stejnou barvou)?

4. SAT (problém splnitelnosti booleovských formulí)

- **Vstup:** Booleovská formule v konjunktivní normální formě.
- **Otzáka:** Je daná formule splnitelná (tj. existuje pravdivostní ohodnocení proměnných, při kterém je formule pravdivá)?

5. 3-SAT (problém SAT s omezením na 3 literály)

- **Vstup:** Formule v konjunktivní normální formě, kde každá klauzule obsahuje pravě 3 literály.
- **Otzáka:** Je formule splnitelná?

6. HK (problém hamiltonovské kružnice)/HC (problém hamiltonovského cyklu)

- **Vstup:** Neorientovaný graf G /Orientovaný graf G .
- **Otzáka:** Existuje v G hamiltonovská kružnice (uzavřená cesta, procházející každým vrcholem právě jednou)?

7. Subset-Sum

- **Vstup:** Množina přirozených čísel $M = x_1, x_2, \dots, x_n$ a přirozené číslo s .
- **Otzáka:** Existuje podmnožina množiny M , pro niž součet jejích prvků je roven s ?

8. Problém obchodního cestujícího (TSP) ANO/NE verze

- **Vstup:** Neorientovaný graf G s hranami ohodnocenými přirozenými čísly a číslo k .
- **Otzáka:** lze objet k měst a neujet víc než danou vzdálenost?

9. Vrcholové pokryti (vertex cover)

- **Vstup:** Neorientovaný graf G a přirozené číslo k .
- **Otzáka:** Existuje v grafu G množina vrcholů velikosti k taková, že každá hrana má alespoň jeden svůj vrchol v této množině?

14 Jazyk predikátové logiky prvního řádu. Práce s kvantifikátory a ekvivalentní transformace formulí.

Predikátová logika (PL) pracuje s primitivními formulemi (**predikáty**) vypovídajícími o **vlastnostech** a **vztazích** mezi **předměty** jistého **univerza** (individui). Je rozšířením výrokové logiky. Na rozdíl od výrokové logiky si všimá i struktury vět samotných a obsahuje predikáty a kvantifikátory. Pouze jen malá část úsudku může být formalizována pomocí výrokové logiky:

$$\begin{array}{c} \text{Všechny opice mají rády banány} \\ \text{Judy je opice} \\ \hline \text{Judy má ráda banány} \end{array}$$

Z hlediska VL jsou to jednoduché výroky p, q, r a z p, q nevyplývá r .

14.1 Predikátová logika 1. řádu

Predikátová logika umožňuje uvažování nad **vlastnostmi**, jež jsou **sdíleny mnoha objekty**, díky použití **proměnných** a **kvantifikátorů**. V predikátové logice by byl výše uvedený úsudek formalizován takto:

$$\begin{array}{c} \text{Každé individuum, je-li Opice pak má rádo Banány.} \\ \text{Judy je individuum s vlastností být Opice.} \\ \hline \text{Judy je individuum s vlastností mít rádo Banány.} \end{array}$$

$\forall x(O(x) \rightarrow B(x)); O(J) \neq B(J)$, kde x je **individuová proměnná**; O, B **predikátové symboly** a J **funkční symbol**.

Poznámka

Pokud bychom chtěli formalizovat úsudky, které navíc vypovídají i o vlastnostech vlastností a vztahů a o vztazích mezi vlastnostmi a vztahy, museli bychom použít predikátovou logiku druhého řádu a vyššího. Tou se ale nebudeme zabývat.

14.1.1 Formální jazyk PL1 – Abeceda

Logické symboly:

- Individuové proměnné: x, y, z, \dots
- Logické spojky: \wedge konjunkce, \vee disjunkce, \rightarrow implikace, \leftrightarrow ekvivalence, \neg negace.
- Kvantifikační symboly: \forall, \exists .

Speciální symboly: (n-arita = počet argumentů)

- Predikátové: P^n, Q^n, \dots
- Funkční: f^n, g^n, h^n, \dots

Pomocné symboly: závorky a jiná interpunkční znaménka (,), ...

14.1.2 Formální jazyk PL1 – Gramatika

Termy:

- každý symbol proměnné x, y, \dots je **term**,
- jsou-li t_1, \dots, t_n ($n \geq 0$) termy a je-li f n-ární **funkční symbol**, pak výraz $f(t_1, \dots, t_n)$ je term; pro $n = 0$ se jedná o **individuovou konstantu** (značíme a, b, c, \dots),
- jen výrazy dle 1. a 2. jsou termy.

Atomické formule:

- je-li P n-ární predikátový symbol a jsou-li t_1, \dots, t_n termy, pak výraz $P(t_1, \dots, t_n)$ je **atomická formule** (na vstupu jsou pouze termy).
- jsou-li t_1 a t_2 termy, tak $t_1 = t_2$ je atomická formule.

Formule:

- každá atomická formule je formule,
 - je-li výraz A formule, pak $\neg A$ je formule,
 - jsou-li výrazy A a B formule, pak výrazy $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$, $(A \leftrightarrow B)$ jsou formule, je-li x proměnná a A formule, pak výrazy $\forall x A$ a $\exists x A$ jsou formule.

14.2 Převod z přirozeného jazyka do PL1

- \forall – „všichni”, „žádný”, „nikdo”, ...
 - \exists – „někdo”, „něco”, „některí”, „existuje”, ...

Větu musíme často ekvivalentně přeformulovat, pozor: v češtině **dvojí zápor**

- Žádný student není důchodce: $\forall x(S(x) \rightarrow \neg D(x))$.
 - Ale, „všichni studenti nejsou důchodci“ čteme jako „ne všichni studenti jsou důchodci“: $\neg \forall x(S(x) \rightarrow D(x)) \leftrightarrow \exists x(S(x) \wedge \neg D(x))$

Jako pomůcka k řešení může sloužit tato zásada:

- Po **všeobecném** kvantifikátoru následuje formule ve tvaru implikace: $\forall \dots \rightarrow \dots$
 - Po **existenčním** kvantifikátoru formule ve tvaru konjunkce: $\exists \dots \wedge \dots$

14.2.1 Volné a vázané proměnné

$$\forall x \exists y P(x, y, t) \wedge \neg \exists x Q(y, x)$$

\bigvee | / \
 vázané, volné volné, vázané

14.3 Ekvivalentní úpravy

Při ekvivalentních úpravách se používají **de Morganovy** zákony v PL1: $\neg\forall x A \leftrightarrow \exists x \neg A$ $\neg\exists x A \leftrightarrow \forall x \neg A$.

Příklady

- Není pravda, že všichni vodníci jsou zelení. \leftrightarrow Některí vodníci nejsou zelení.
 $\neg\forall x(V(x) \rightarrow Z(x)) \leftrightarrow \exists x(V(x) \wedge \neg Z(x))$
 - Není pravda, že některí vodníci jsou zelení. \leftrightarrow Žádný vodník není zelený.
 $\neg\exists x(V(x) \wedge Z(x)) \leftrightarrow \forall x(V(x) \rightarrow \neg Z(x))$
 - Everybody loves somebody sometimes.
 $\forall x\forall y\forall zL(x, y, z)$
 - Marie má ráda pouze vítěze.
 $\forall x(R(m, x) \rightarrow V(x))$

14.4 Ekvivalentní transformace

- Aplikace negace: $\neg\forall x[V(x) \rightarrow Z(x)] \leftrightarrow \exists x[V(x) \wedge \neg Z(x)].$
 - **De morganovy** zákony: $\forall x[((P(x) \wedge Q(x)) \vee D(x)] \leftrightarrow \forall x[(P(x) \vee D(x)) \wedge (Q(x) \vee D(x))].$
 $\neg(A \vee B) \iff (\neg A) \wedge (\neg B)$
 $\neg(A \wedge B) \iff (\neg A) \vee (\neg B)$
 - Převod **implikace**: $\forall x(P(x) \rightarrow G(x)) \leftrightarrow \forall x(\neg P(x) \vee G(x)).$

Patří zde i část z převodu do **Skolemovy Klauzárni formy**:

- ## 1. eliminace nadbytečných kvantifikátorů.

2. eliminace spojek $\rightarrow, \leftrightarrow,$
3. přesun negace dovnitř,
4. přejmenování proměnných,
5. přesun kvantifikátorů doprava,
6. přesun všeobecných kvantifikátorů doleva,
7. použití distributivních zákonů.

14.5 Sémantika v PL1

- Při substituci termů za proměnné ($A(x/t)$), je třeba dbát na nahrazování pouze **volných proměnných**.
- Při definici formule, je nutné vysvětlit co **znamenají** jednotlivé predikátové symboly, termy atd.

15 Pojem relace, operace s relacemi, vlastnosti relací. Typy binárních relací. Relace ekvivalence a relace uspořádání.

15.1 Relace

- **N-ární relace** nad množinami A_1, \dots, A_n je **libovolná podmnožina kartézského součinu** $A_1 \times \dots \times A_n$ (tyto množiny jsou **nosičemi** relace).
- **Kartézský součin** množin A a B , označovaný $A \times B$, je množina všech uspořádaných dvojic, kde první prvek z dvojice patří do množiny A a druhý do množiny B . Příklad: $\{a, b\} \times \{a, b, c\} = \{(a, a), (a, b), (a, c), (b, a), (b, b), (b, c)\}$

15.1.1 Typy relací

- **homogenní** – jediný druh nosiče ($A \times A$),
- **heterogenní** – alespoň dva různé druhy nosiče ($A \times B$),
- **unární** ($n = 1$), **binární** ($n = 2$), **ternární** ($n = 3$), **n-ární** – podle arity,
- **triviální – úplná** ($\rho = A_1 \times A_n$), **prázdná** ($\rho = \emptyset$),
- **netriviální** – $\emptyset \subset \rho \subset A_1 \times \dots \times A_n$.

15.1.2 Vlastnosti relací

Binární relace $R \subseteq A \times A$ je:

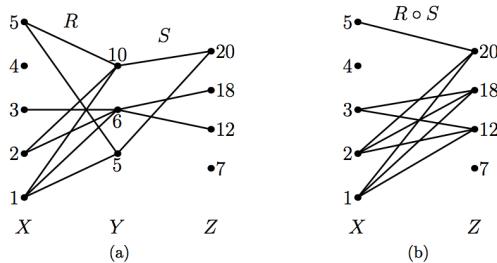
- **Reflexivní** – $\forall x \in A : (x, x) \in R$.
- **Ireflexivní** – $\forall x \in A : (x, x) \notin R$.
- **Symetrická** – $\forall x, y \in A : (x, y) \in R \Rightarrow (y, x) \in R$.
- **Asymetrická** – $\forall x, y \in A : (x, y) \in R \Rightarrow (y, x) \notin R$.
- **Antisymetrická** – $\forall x, y \in A : (x, y) \in R \wedge (y, x) \in R \Rightarrow x = y$.
- **Tranzitivní** – $\forall x, y, z \in A : (x, y) \in R \wedge (y, z) \in R \Rightarrow (x, z) \in R$.

Příklad

- Relace „=” na \mathbb{N} je reflexivní, symetrická, antisymetrická a tranzitivní, ale není ireflexivní ani asymetrická.
- Relace „ \leq “ na \mathbb{N} je reflexivní, antisymetrická a tranzitivní, ale není ireflexivní, symetrická ani asymetrická.
- Relace „ $<$ “ na \mathbb{N} je ireflexivní, asymetrická, antisymetrická a tranzitivní, ale není reflexivní ani symetrická.

15.2 Operace s relacemi

- **Průnik** – Prvek x náleží do průniku relací $R1 \cap R2$, pokud patří do množiny $R1(x \in R1)$ a **zároveň** do $R2(x \in R1)$.
- **Sjednocení** – Prvek x náleží do sjednocení relací $R1 \cup R2$, pokud patří do množiny $R1(x \in R1)$ **nebo** $R2(x \in R1)$.
- **Doplnek** – Doplňkem $R1'$ k relaci $R1$ rozumíme **všechny prvky které nepatří** do $R1$.
- **Inverze** – Relace $R^{-1} \subseteq B \times A$ je inverzní k relaci $R \subseteq A \times B$, pokud $xR^{-1}y \Leftrightarrow yRx$.
- **Skládání relací** – Výsledkem je množina dvojic, kde pokud existují dvojice $(a, b) \in R$ a $(b, c) \in S$, pak jejich složení $(a, c) \in R \circ S$.

Obrázek 1.2: (a) Relace R a S , (b) jejich složení.

15.3 Typy binárních relací

Mezi nejznámější typy binárních relací patří **ekvivalence** (=) [Re, Sy, Tr], **uspořádání** ($<$, $>$, \leq , \geq) [Re, An, Tr] a **tolerance** [Re, Sy].

15.3.1 Ekvivalence [Re, Sy, Tr]

Relace ekvivalence představuje jakési zjemnění relace rovnosti. Vždy můžeme rozhodnout, že jsou dva prvky množiny stejné, tj. že $a = a$. Ale někdy se nám hodí zjistit, zda jsou si dva prvky **pouze podobné**, ne nutně stejné. Neboli zda mají stejnou nějakou zásadní vlastnost. Například dvě knihy můžeme považovat za podobné, pokud mají stejný žánr nebo **pomocí ekvivalence**: dvě knihy jsou ekvivalentní pokud mají stejný žánr.

- Binární relace na množině X je ekvivalentní, pokud je R na A : **reflexivní**, **symetrická** a **tranzitivní**.
- **Třída ekvivalence** prvku a je množina všech prvků ekvivalentních s daným prvkem a .
- **Průnik** dvou ekvivalence je zase ekvivalence.
- **Sjednocení** dvou ekvivalence nemusí znamenat, že výsledek bude ekvivalentní.

15.3.2 Uspořádání [Re, An, Tr]

- Binární relace na množině X je **neostrým uspořádáním**, pokud je R na A : **reflexivní**, **antisymetrická** a **tranzitivní**.
- Binární relace na množině X je **ostrým uspořádáním**, pokud je R na A : **ireflexivní**, **antisymetrická** a **tranzitivní**.
- Uspořádání je **úplné** pokud neexistují neporovnatelné prvky.

16 Pojem operace a obecný pojem algebra. Algebry s jednou a dvěma binárními operacemi.

16.1 Algebra

Algebra je naukou o **algebraických strukturách**, tedy **množinách**, na nichž jsou zavedeny nějaké **operace**. Slouží pro popis objektů reálného světa a operací prováděných s těmito objekty. Příklady algebr:

- $(\mathbb{N}, \{+\})$ - sčítání nad množinou přirozených čísel,
- $(2^M, \{\cup, \cap\})$ - množina všech podmnožin M s operací průnik a sjednocení.

16.1.1 Definice

Každý objekt algebry je reprezentován **datovým nosičem** (množina popisující data, se kterými pracujeme). A **operacemi** – nejjednoduššími transformacemi, které nad daty můžeme realizovat. **Algebraická struktura** je definována jako (A, \circ) , kde:

- A – nosič algebry (množina objektů – čísel, proměnných, ...),
- \circ – množina operací nad nosičem X .

16.2 Operace

Operace na množině A je definována jako zobrazení

$$f : A^n \rightarrow A, \quad (16.1)$$

tedy zobrazení, které každé n -tici prvků množiny A , jednoznačně přiřazuje prvek z množiny A . Číslo n nazýváme **arita operace** a podle něj operace označujeme jako **nulární** ($n = 0$), **unární** ($n = 1$), **binární** ($n = 2$), **ternární** ($n = 3$).

16.3 Algebraické struktury s jednou binární operací

Definována jako (A, \circ) s jedním nosičem (A) a jednou homogenní binární operací (\circ). Nejprve je nutné zmínit **vlastnosti binárních operací**:

- **asociativita:** $a * (b * c) = (a * b) * c$,
- **komutativita:** $a * b = b * a$.

Kromě již zmíněné asociativity a komutativity algebraické struktury také zavádí existenci:

- **Jednotkového prvku:** e takové, že $\forall x \in X : x \circ e = e \circ x = x$. Tedy prvek, který nezmění výsledek (1 u násobení, 0 u sčítání).
- **Inverzního prvku:** \bar{x} takové, že $\forall x \in X : x \circ \bar{x} = \bar{x} \circ x = e$. Tedy prvek, který převede výsledek na jednotkový prvek.

neutrálního či **inverzního prvku**, a další charakteristiky.

16.3.1 Klasifikace algebraických struktur

Všechny níže uvedené klasifikace algebraické struktury (A, \circ) zahrnují i ty co jsou pod nimi. Tedy pokud je nějaká algebraická struktura (AS) Monoid, je i Pologrupa a Grupoid.

1. **Grupoid – uzavřenost** (univerzalita) na nosiči (po výpočtu je výsledek stále v množině A).
2. **Pologrupa** – splňuje vlastnost **asociativity**.
3. **Monoid** – existence **jednotkového prvku**.
4. **Grupa** – existence **inverzního prvku**.
5. **Abelova grupa** – splňuje vlastnost **komutativity** (symetrická podle diagonály).

Kongruence – označuje ekvivalenci na algebře, která je slučitelná se všemi operacemi na algebře.

16.3.2 Morfismy

- **Homomorfismus** – zobrazení, které převádí jednu algebraickou strukturu na jinou: $f(a_1 \cdot a_2) \rightarrow f(a_1) \circ f(a_2)$.
- **Izomorfismus** – bijektivní homomorfismus.
- **Epimorfismus** – surjektivní homomorfismus.
- **Monomorfismus** – injektivní homomorfismus.
- **Endomorfismus** – homomorfismus z objektu do sebe sama (stejná množina).
- **Automorfismus** – endorfismus, který je izomorfní.

16.4 Okruhy (Algebraické struktury s dvěma binárními operací)

Okruh je algebraický systém $(A, +, \cdot)$ se dvěma základními binárními operacemi, kde první $(A, +)$ je **abelova grupa** a druhá (A, \cdot) je alespoň **pologrupa**. Podobně jako u předchozí AS, i zde se zavádí nový pojem:

- **Existence dělitele nuly** – říká, že ve struktuře existují 2 nenulové prvky, pro něž platí $a \circ b = 0$.

U všech typů okruhů musí být splněna podmínka první struktury, která musí být **abelova grupa**, a druhá musí být:

- **Okruh** - uzavřená (U), asociativní (A) [pologrupa].
- **Unitární okruh** - U, A, existence jednotkového prvku (J) [monoid].
- **Obor Integrity** - U, A, J [monoid] + **nesmí** obsahovat dělitele nuly.
- **Těleso** - U, A, J a existence inverzního prvku (I) [grupa] + **nesmí** obsahovat dělitele nuly.
- **Pole** - U, A, J, I a komutativita [grupa] + **nesmí** obsahovat dělitele nuly.

17 FCA – formální kontext, formální koncept, konceptuální svazy.

17.1 Formální konceptuální analýza (FCA)

Metoda analýzy tabulkových dat (objektů a jejich vlastností), umožňuje jiný pohled na data (využívá se např. u data miningu). **Vstupem** pro FCA jsou **tabulková data**, která jsou uspořádána následovně: **objekty** (řádky) a **atributy** (sloupce). Tyto tabulková data vytváří tzv. **kontexty**.

	červené	bílé
jablko	×	
zelí	×	×

17.2 Formální Kontext

Formální kontext K obsahuje objekty z množiny O a atributy z množiny A . Vztahy mezi objekty a atributy jsou charakterizovány binární relací R . Obecně se pro popis kontextu používá výraz:

$$K = (O, A, I). \quad (17.1)$$

Takto vymezený formální kontext je dobře zobrazitelný tabulkou, ve které jsou **řádky** obsazeny **objekty**, **sloupce** **atributy** a incidenční data ($I \subseteq O \times A$) vyjadřují relaci R (I je relace incidence).

17.2.1 Galoisovy konexe

Umožňují přecházet z množiny objektů na jejich společné atributy (\uparrow **intent**) a naopak (\downarrow **extent**).

- **Intent** $\uparrow - 2^X \rightarrow 2^Y; A \subseteq X; A^\uparrow = \{y \in Y; \forall x \in A(x, y) \in I\}$ [z objektu na atributy].
- **Extent** $\downarrow - 2^Y \rightarrow 2^X; B \subseteq Y; B^\downarrow = \{x \in X; \forall y \in B(x, y) \in I\}$ [z atributů na objekt].

Příklad

$$\begin{aligned} A_1 &= \{\text{jablka, zelí}\}, A_1^\uparrow &= \{\text{červené}\} & A_2 &= \{\text{zelí}\}, A_2^\uparrow &= \{\text{červené, bílé}\} \\ B_1 &= \{\text{červené, bílé}\}, B_1^\downarrow &= \{\text{zelí}\} & B_2 &= \{\text{červené}\}, B_2^\downarrow &= \{\text{zelí, jablko}\} \end{aligned}$$

17.3 Formální koncept

Formální koncept je dvojice (A, B) , kde A je množina objektů, a B je množina atributů, které jsou **společné pro všechny objekty** z množiny A . Koncept (A, B) je tedy dán jako: $(A, B) \Leftrightarrow A = B^\downarrow \wedge B = A^\uparrow$, kde $A \subseteq X$, $B \subseteq Y$, kde X a Y jsou objekty a atributy výše uvedené tabulky.

17.3.1 Uzávěrový operátor $\uparrow\downarrow$

Jak již znační $A^{\uparrow\downarrow} = C(A)$ vypovídá, k určení uzávěru atributů množiny A se nejprve provede **intent** a poté **extent**. Uzávěr má tyto vlastnosti:

1. **Idempotence** – $C(C(A)) = C(A)$.
2. **Extensionalita** – $A \subseteq C(A)$.
3. **Monotonie** – $A_1 \subseteq A_2 \Rightarrow C(A_1) \subseteq C(A_2)$.

17.4 Konceptuální svazy

Uspořádaná množina konceptů tvoří tzv. konceptuální svaz. Ten lze graficky znázornit **Hasseovým diagramem** – každý vrchol grafu reprezentuje **jeden koncept**. Koncepty K_i a K_j jsou v grafu spojeny, pokud $K_i \leq K_j$, přičemž K_j je umístěn výše než K_i a neexistuje takové K_k , že $K_i \leq K_k \leq K_j$.

17.4.1 Návod k vytvoření konceptuálního svazu

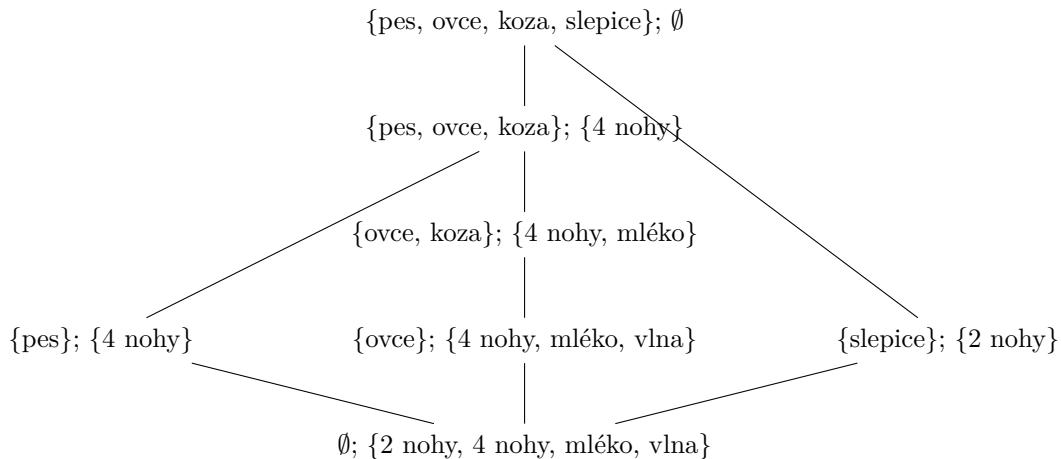
1. Vytvořím a postupně si zapíšu množinu všech extentů na jednotlivých atributech (v grafu zapisuji **zdola nahoru**).
2. Vytvořím jedinečné průniky extentů.
3. Nesmím zapomenout na zahrnutí průniku s prázdnou množinou $= \emptyset$.
4. Přidám extent zahrnující všechny objekty.
5. Pro odpovídající extenty vytvořím intenty (v grafu zapisuji **shora dolů**).

Příklad

	2 nohy	4 nohy	mléko	vlna
pes		×		
ovce		×	×	×
koza		×	×	
slepice	×			

$$\begin{aligned}
 e_7 &= \{\text{pes, ovce, koza, slepice}\} \\
 e_2 &= \{4 \text{ nohy}\}^\downarrow = \{\text{pes, ovce, koza}\} \\
 e_3 &= \{\text{mléko}\}^\downarrow = \{\text{ovce, koza}\} \\
 e_1 &= \{2 \text{ nohy}\}^\downarrow = \{\text{slepice}\} \\
 e_4 &= \{\text{vlna}\}^\downarrow = \{\text{ovce}\} \\
 e_5 &= e_2 \cap e_3 = \{\text{ovce, koza}\} \\
 e_6 &= \emptyset
 \end{aligned}$$

$$\begin{aligned}
 i_7 &= e_7^\uparrow = \emptyset \\
 i_2 &= e_2^\uparrow = \{4 \text{ nohy}\} \\
 i_3 &= e_3^\uparrow = \{4 \text{ nohy, mléko}\} \\
 i_1 &= e_1^\uparrow = \{2 \text{ nohy}\} \\
 i_4 &= e_4^\uparrow = \{4 \text{ nohy, mléko, vlna}\} \\
 i_5 &= e_5^\uparrow = \{4 \text{ nohy, mléko}\} \\
 i_6 &= e_6^\uparrow = \{2 \text{ nohy, 4 nohy, mléko, vlna}\}
 \end{aligned}$$



17.5 Svazy (pro doplnění)

Svaz je algebra (L, \cap, \cup) s dvěma základními binárními operacemi $x \cap y$ **spojení (suprénum)** ($\text{sup}(x, y)$) a $x \cup y$ **průsek (infimum)** ($\text{inf}(x, y)$), které mají následující vlastnosti:

1. **Univerzalita (jednoznačnost)** – $\forall x, y \exists z \ x \cap y = z \quad | \quad \forall x, y \exists z \ x \cup y = z$.
2. **Asociativita** – $x \cap (y \cap z) = (x \cap y) \cap z \quad | \quad x \cup (y \cup z) = (x \cup y) \cup z$.
3. **Komutativita** – $x \cap y = y \cap x \quad | \quad x \cup y = y \cup x$.
4. **Absorbce** – $x \cap (x \cup y) = x \quad | \quad x \cup (x \cap y) = x$.

17.5.1 Typy svazů

1. **Distributivní** – platí zde axiomy distributivity a neobsahuje ani **diamond** ani **pentagon**: $x \cup (y \cap z) = (x \cup y) \cap (x \cup z)$ | $x \cap (y \cup z) = (x \cap y) \cup (x \cap z)$.



2. **Modulární** – slabší reprezentace distributivity, **nesmí** obsahovat **pentagon**, $a \geq c : a \wedge (b \vee c) = (a \wedge b) \vee c$.
3. **Komplementární** – platí zde, že pro každý prvek x existuje komplement x' , kdy: $x \cap x' = [\text{svazová 0}]$ a $x \cup x' = [\text{svazová 1}]$
4. **Booleovský svaz** – komplementární \wedge distributivní

17.5.2 Vlastnosti svazů

Pro každé dva prvky v množině existuje **sup** a **inf**. **Úplný svaz** – nastane tehdy, zda pro **libovolné neprázdné podmnožiny** existuje sup a inf. U svazů můžeme dále získat tyto vlastnosti:

- **Minimum a maximum** – žádný není menší/větší než a (vrchol diagramu).
- **Nejmenší a největší** – **pouze jeden** nejmenší/největší prvek, pokud je jich více, **neexistuje** největší/nejmenší prvek.
- **Dolní (L(A)) a horní (U(A)) závora** – všechny prvky jsou \leq / \geq než A ,
- **Infimum** – nejmenší prvek dolní závory.
- **Supremum** – nejmenší prvek horní závory.

18 Asociační pravidla, hledání často se opakujících množin položek.

Termín asociační pravidla široce zpopularizoval počátkem 90. let v souvislosti s analýzou nákupního košíku. Při této analýze se zjišťuje, jaké druhy zboží si současně kupují zákazníci v supermarketech (např. pivo a párek). **Jde tedy o hledání vzájemných vazeb (asociaci) mezi různými položkami** sortimentu prodejny. Přitom není upřednostňován žádný speciální druh zboží jako závěr pravidla.

18.0.1 Základní charakteristika pravidel

U pravidel vytvořených z dat nás obvykle zajímá kolik příkladů splňuje **předpoklad** a kolik **závěr** pravidla, kolik příkladů splňuje předpoklad i závěr **současně**, kolik příkladů splňuje předpoklad a **nesplňuje** závěr.... Tedy, zajímá nás, jak pro pravidlo:

$$Ant \Rightarrow Suc, \quad \text{kde } Ant, Suc \subseteq I \text{ (položky)} \quad (18.1)$$

kde **Ant** (**předpoklad**, levá strana pravidla, **antecedent**) a **Suc** (**závěr**, pravá strana pravidla, **sukcedent**) jsou kombinace kategorií, pro něž příslušná **kontingenční tabulka** vypadá následovně:

		<i>Suc</i>	$\neg Suc$
		a	b
<i>Ant</i>	$\neg Ant$	c	d

- $Ant \wedge Suc$ – **a** je počet objektů pokrytých současně předpokladem i závěrem,
- $Ant \wedge \neg Suc$ – **b** je počet objektů pokrytých předpokladem a nepokrytých závěrem,
- $\neg Ant \wedge Suc$ – **c** je počet příkladů nepokrytých předpokladem ale pokrytých závěrem,
- $\neg Ant \wedge \neg Suc$ – **d** je počet příkladů nepokrytých ani předpokladem ani závěrem.

18.0.2 Základní charakteristiky asociačních pravidel

- **Support (podpora)** – relativní četnost objektů splňující předpoklad i závěr, jinými slovy $\frac{\text{počet splňující výstup}}{\text{počet položek}}$:

$$sup(Ant \Rightarrow Suc) = \frac{a}{a + b + c + d}, \in \langle 0; 1 \rangle. \quad (18.2)$$

- **Confidence (spolehlivost)** – podmíněná pravděpodobnost závěru pokud platí předpoklad, tedy $\frac{\text{podpora obou}}{\text{podpora závěru}}$:

$$conf(Ant \Rightarrow Suc) = \frac{sup(Ant \cup Suc)}{sup(Suc)} = \frac{a}{a + b}. \quad (18.3)$$

- Další: **pokrytí, zajímavost, závislost**.

18.1 Hledání často se opakujících množin položek)

Frequent item set je množina, kde $sup(K) \geq \gamma$, máme tedy stanovenou **minimální podporu**. Pokud je např. $\gamma = 0,3$ pak je minimální podpora 30%.

18.1.1 Generování kombinací

Základem všech algoritmů pro hledání asociačních pravidel je **generování kombinací (konjunkcí) hodnot atributů**. Při generování vlastně procházíme (prohledáváme) prostor všech přípustných konjunkcí. Metod je několik:

- do **hloubky**,
- do **šířky**,
- **heuristicicky**,

18.1.2 Algoritmus apriori

Jedná se o nejznámějším algoritmus pro hledání asociačních pravidel. Jádrem algoritmu je **hledání často se opakujících množin** položek (frequent itemsets). Jedná se kombinace (konjunkce) kategorií, které dosahují předem zadané četnosti (**minimální podpora**) v datech.

Algoritmus apriori	
1.	do L_1 přiřaď všechny kategorie, které dosahují alespoň požadované četnosti
2.	polož $k=2$
3.	dokud $L_{k-1} \neq \emptyset$ <ol style="list-style-type: none"> 3.1. pomocí funkce <i>apriori-gen</i> vygeneruj na základě L_{k-1} množinu kandidátů C_k 3.2. do L_k zařaď ty kombinace z C_k, které dosahly alespoň požadovanou četnost 3.3. zvětš počítadlo k
Funkce <i>apriori-gen</i> (L_{k-1})	
1.	pro všechny dvojice kombinací $Comb_p, Comb_q$ z L_{k-1} <ol style="list-style-type: none"> 1.1. pokud $Comb_p$ a $Comb_q$ se shodují v $k-2$ kategoriích přidej $Comb_p \wedge Comb_q$ do C_k 1.2. pro každou kombinaci $Comb$ z C_k <ol style="list-style-type: none"> 2.1. pokud některá z jejich podkombinací délky $k-1$ není obsažena v L_{k-1} odstraň $Comb$ z C_k

Při hledání kombinací délky k , které mají vysokou četnost se využívá toho, že **již známe kombinace délky $k-1$** . Při vytváření kombinace délky k spojujeme kombinace délky $k-1$.

Jde tedy o **generování kombinací „do šířky“**. Přitom pro vytvoření jedné kombinace délky k požadujeme, aby všechny její podkombinace délky $k-1$ **splňovaly požadavek na četnosti**. Tedy např. ze tříčlenných kombinací $\{A_1 A_2 A_3, A_1 A_2 A_4, A_1 A_3 A_4, A_1 A_3 A_5, A_2 A_3 A_4\}$ dosahujících požadované četnosti vytvoříme **pouze jedinou čtyřčlennou** kombinaci $A_1 A_2 A_3 A_4$. Kombinaci $A_1 A_3 A_4 A_5$ sice lze vytvořit spojením $A_1 A_3 A_4$ a $A_1 A_3 A_5$, ale mezi tříčlennými kombinacemi chybí $A_1 A_4 A_5$ i $A_3 A_4 A_5$.

18.1.3 Algoritmus Next Closure

Slouží k vytváření formálních kontextů, **vyhledáváním nejmenších intentů**, postup:

- Začnu s následující tabulkou:

A	i	$A \cap \{1 \dots i-1\} \cup \{i\} = B'$	$cl(B') = B$	$B \setminus A$	je-li $B \setminus A = \{i\}$ nebo větší \rightarrow ANO je-li $B \setminus A = \{j\}$, kde $j < i \rightarrow$ NE
\emptyset	5				

- Do A vložím prázdnou množinu a i nastavím na nejvyšší intent (pořadí).
- Udělám průnik s $A \cap \{1 \dots i-1\} \cup \{i\} = B' \rightarrow$ průnik Ačka s intenty od 1 do $i-1$, k tomu přidám i . Př.: $A = \{1, 3, 4\}, i = \{3\} \Rightarrow B' = \{1, 3\}$.
- Udělám closure (B') \rightarrow intent na extent \rightarrow extent na intent $\Rightarrow B'^{\downarrow\uparrow}$.
- Od B odečtu A .
- Je-li:
 - $B \setminus A = \{i\}$ a větší tak ANO [je-li nejmenší prvek z $B \setminus A$ roven nebo větší než $\{i\}$],
 - $B \setminus A = \{j\}$ kde $j < i$ tak NE [je-li nejmenší prvek z $B \setminus A$ menší než i pak NE].
- Pokud:
 - ANO \rightarrow do A dosadíme $cl(B') = B$ a i nastavíme na nejvyšší intent,
 - NE \rightarrow do A neměním a snížíme i o -1 .
- Skončím když je A rovno celé množině extentů.

DŮLEŽITÉ

V i přeskakuju hodnoty, které jsou v A [výjde pro ně $\emptyset \rightarrow$ neřeším].

Příklad

	y_1	y_2	y_3	y_4	y_5
x_1	0	1	0	1	1
x_2	0	1	1	0	0
x_3	1	1	0	1	1
x_4	1	1	1	0	0
x_5	1	0	1	1	0
x_6	1	0	0	1	0

A	i	$A \cap \{1 \dots i-1\} \cup \{i\} = B'$	$cl(B') = B$	$B \setminus A$	ANO / NE
\emptyset	5	5	2, 4, 5	2, 4, 5	N [2 < 5]
\emptyset	4	4	4	4	A [4 \geq 4] $\leftarrow i_n$
4	5	4, 5	2, 4, 5	2, 5	N [2 < 5]
4	3 [skip i]	3	3	3	A [3 \geq 3] $\leftarrow i_{n-1}$
3	5	3, 5	1, 2, 3, 4, 5	1, 2, 4, 5	N [1 < 5]
		\vdots			
1, 2, 3, 4, 5		KONEC			A i_1

19 Metrické a topologické prostory – metriky a podobnosti.

19.1 Metrický prostor

Metrický prostor je **matematická struktura**, pomocí které lze formálním způsobem definovat pojem **vzdálenosti**. Na metrických prostorech se poté definují další topologické vlastnosti jako např. **otevřenost** a **uzavřenosť** množin, jejichž zobecnění pak vede na ještě abstraktnější matematický pojem **topologického prostoru**.

19.1.1 Formální definice

Metrický prostor je dvojice (M, ρ) , kde M je libovolná neprázdná množina a ρ je **metrika**, což je zobrazení:

$$\rho : M \times M \rightarrow \mathbb{R},$$

které splňuje následující axiomy:

1. **Nezápornost**: $\forall x, y \rho(x, y) \geq 0$.
2. **Totožnost**: $\forall x, y \rho(x, y) = 0 \Leftrightarrow x = y$.
3. **Symetrie**: $\forall x, y \in M : \rho(x, y) = \rho(y, x)$.
4. **Trojúhelníková nerovnost**: $\forall x, y, z \in M : \rho(x, y) + \rho(y, z) \geq \rho(x, z)$.

19.1.2 Metriky v \mathbb{R}^n

U metrik v \mathbb{R}^n platí:

- Každý normovaný vektorový prostor je metrickým prostorem.
- Množina reálných čísel spolu s metrikou $\rho(x, y) = |x - y|$, kde x, y jsou libovolné body množiny \mathbb{R} tvoří **úplný metrický prostor**.

Mezi nejpoužívanější **metriky** na Euklidovském prostoru \mathbb{R}^n patří:

1. **Manhattanská** – $\rho_1(x, y) = \sum_{i=0}^n |x_i - y_i|$.
2. **Euklidovská** – $\rho_2(x, y) = \sqrt{\sum_{i=0}^n |x_i - y_i|^2}$.
3. **Minkowského** – $\rho_3(x, y) = (\sum_{i=0}^n |x_i - y_i|^P)^{1/P}$, kde $P \geq 1$; $P \in \mathbb{R}$.
4. **Čebyševova (Maximova)** – $\rho_{\max}(x, y) = \max_{\forall i} |x_i - y_i|$, speciální případ Minkowského metriky pro $P = \infty$.

19.1.3 Podobnosti a nepodobnosti

- **Cosinova podobnost** – míra podobnosti dvou vektorů, která se získá výpočtem kosinu úhlu těchto vektorů:

$$S_c(x, y) = \frac{x \cdot y}{\|x\| \|y\|} = \frac{\sum_{i=0}^n x_i y_i}{\sqrt{\sum_{i=0}^n x_i^2} \sqrt{\sum_{i=0}^n y_i^2}}$$

- **Jaccardova podobnost** – $S_j(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$ používá se pro porovnání podobnosti dvou množin.
- **Jaccardova nepodobnost** (svým způsobem vzdálenost) – $d = 1 - S$, $d = \frac{1}{S} - 1$ pro $S \neq 0$.

19.1.4 Vzdálenost mezi slovy

K určení vzdálenosti mezi textovými řetězci definujeme tyto **vzdálenosti**:

1. **Hammingova** – počet rozdílných písmen na stejných pozicích: $d_h(\text{Karolin}, \text{Kathrin}) = 3$, lze použít pouze pro **stejně dlouhá slova!**.
2. **LCS (Longest common subsequence)** – počet operací **vkládání** a **mazání** nutných k převodu jednoho slova na druhé: $d_{LCS}(\text{kitten}, \text{sitting}) \Rightarrow \text{itten} [-\text{K}] \rightarrow \text{sitten} [+S] \rightarrow \text{sittn} [-\text{E}] \rightarrow \text{sittin} [+I] \rightarrow \text{sitting} [+G] = 5 \text{ operací}$.
3. **Levenshteinova** – počet operací **vkládání**, **mazání**, **substituce** nutných k převodu jednoho slova na druhé: $d_l(\text{kittne}, \text{sitting}) \Rightarrow \text{sitten} [\text{K} \sim \text{S}] \rightarrow \text{sittin} [\text{E} \sim \text{I}] \rightarrow \text{sitting} [+G] = 3 \text{ operace}$.

Editační vzdálenost patří zde LSC a Levenshteinova vzdálenost (při určení se používají úpravy).

19.1.5 Normalizace

Snažíme se dostat všechny hodnoty atributů ve sloupci do intervalu $\langle 0, 1 \rangle$, proto dělíme celý sloupce jeho maximem (dělíme v rámci daného sloupce, pro každý sloupec zvlášť vlastním maximem).

Příklad

d'_1 a d'_2 reprezentují **normovaný tvar** vzdáleností d_1 a d_2 :

	t_1	t_2	t_3	t_4	t_5	t_6	t_7
d_1	1	0	0	1	3	0	1
d_2	1	0	0	2	2	0	0
d'_1	1	0	0	$\frac{1}{2}$	1	0	1
d'_2	1	0	0	1	$\frac{2}{3}$	0	0

	$i = 1$	$i = 2$	$i = 3$	$i = \infty$	Cosinova	Jaccard
$\rho_i(d_1, d_2)$	3	$\sqrt{3}$	$\sqrt{3}^3$	1	$\frac{1}{6\sqrt{3}}$	$\frac{3}{4}?$
$\rho_i(d'_1, d'_2)$	$\frac{11}{6}$	$\frac{7}{6}$	$\frac{3\sqrt{251}}{6}$	1	$\frac{5\sqrt{154}}{3\sqrt{2}}$	$\frac{3}{4}?$

19.2 Topologický prostor

Jedná se o **rozšíření (zobecnění) metrického prostoru**. Cílem topologie je studium **vlastností prostorů**. Na rozdíl od teorie metrických prostorů se v topologii **nezajímáme o vzdálenosti mezi body** prostoru a prostory považujeme za stejné, pokud **se na sebe dají vzájemně přeměnit** nějakou spojitou deformací. Takže např. nerozlišujeme mezi koulí a krychlí, ostatně koule se změní v krychli již při přechodu mezi dvěma ekvivalentními metrikami v \mathbb{R}^3 .

Základním pojmem, který se v topologii studuje je **spojitost zobrazení**. Proto není úplně potřeba vědět přesně, jak jsou od sebe které body daleko. Vystačíme si s informacemi, že jisté body se nekonečně blíží k nějakému bodu prostoru.

19.2.1 Formální definice

Topologickým prostorem nazveme množinu X společně s kolekcí τ podmnožin X , tedy **dvojici** (X, τ) , splňující následující axiomy:

1. $\emptyset, X \in \tau$,
2. $\forall A, B \in \tau \Rightarrow A \cup B \in \tau$, tedy **sjednocení** libovolného počtu (tj. konečného, spočetného i nespočetného) množin z τ leží v τ ,
3. $\forall A, B \in \tau \Rightarrow A \cap B \in \tau$, tedy **průnik** konečného počtu množin z τ leží v τ .

19.2.2 Uzavřená, otevřená množina

Pro topologický prostor (X, τ) je každá množina $A \in \tau$ otevřená množina, a její doplněk je uzavřená množina.

19.2.3 Souvislost nesouvislost

Pokud sjednocením dvou neprázdných množin z τ získáme všechny prvky topologie (celé X), tak je topologie **nesouvislá**. Příklad:

$$\begin{aligned} X &= \{a, b, c\} \\ \tau &= \{\emptyset, X, \{a, b\}, \{c\}\} \\ \{a, b\} \cup \{c\} &= \{a, b, c\} \Rightarrow \tau \text{ je nesouvislá} \end{aligned}$$

Poznámka: sjednocované množiny musí být **disjunktní**, tedy nesmí mít společné prvky, např.: $\{a, b\}$ a $\{a, c\}$ již disjunktní nejsou, ale nenaruší souvislost.

Topologický prostor τ , je **souvislý** právě tehdy, když jedině podmnožiny v τ , které jsou současně otevřené i uzavřené jsou X a \emptyset . V opačném případě je τ **nesouvislý**.

19.2.4 Uzávěrový systém

Uzávěrový systém C nad množinou X obsahuje X a $\forall A, B \in C$ platí, že $A \cap B \in C$.

$$R_i \subseteq A \times A \quad R_i^* = [\text{tranzitivně-reflexivní uzávěr}]$$

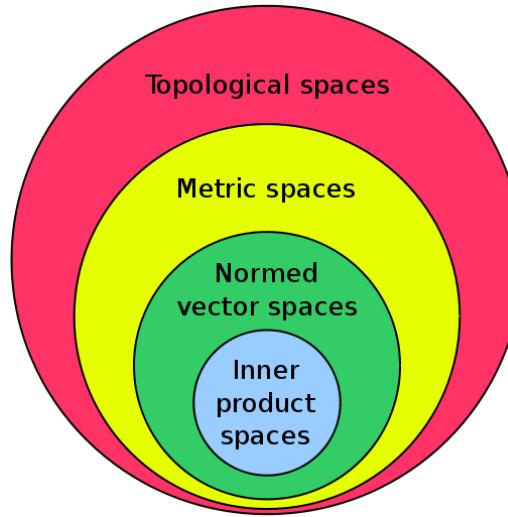
pro R_i^* platí $R_1^* \neq R_2^*$; $R_1^* \cap R_2^* \in G$; $R_1^* \cup R_2^* \notin G$

19.2.5 Uzávěrový operátor $cl(A)$

Uzávěr (*closure*) $A \cup C \rightarrow cl(A)$ je **nejmenší uzavřená množina obsahující daný prvek**. Analogie u konceptů $cl(B) = B^{\downarrow\uparrow}$ a $cl(A) = A^{\uparrow\downarrow}$. Vlastnosti uzávěrového operátoru:

1. **Idempotence** – $cl(cl(A)) = cl(A)$.
2. **Extensionalita** – $A \subseteq cl(A)$.
3. **Monotónost** – $A \subseteq B \Rightarrow cl(A) \subseteq cl(B)$.

Platí-li navíc $cl(\emptyset) = \emptyset$ a $cl(A \cup B) = cl(A) \cup cl(B)$, pak je uzávěr $A = cl(A)$. Jinými slovy se jedná o nejmenší komplement (doplňek) množin TP obsahující daný prvek.



20 Shlukování.

Shluková analýza je vícerozměrná statistická metoda, která se používá ke **klasifikaci objektů**. Slouží k **třídění jednotek do skupin** (shluků) tak, aby si jednotky náležící do stejné skupiny byly podobnější než objekty z ostatních skupin.

- **Shlukování** – proces **seskupování dat** do skupin na základě podobnosti.
- **Shluk** – množina maximálně si **podobných** v rámci shluku a maximálně **odlišných** mezi shluky.
- **Podobnost** mezi objekty se stanoví na základě **vzdálenosti** (některé z metrik zmíněných výše – Manhattan, Euklidovská, Minkowského, Čevyševova (Maximova)).

Metody shlukování můžeme klasifikovat do dvou základních kategorií **hierarchické** a **nehierarchické metody**.

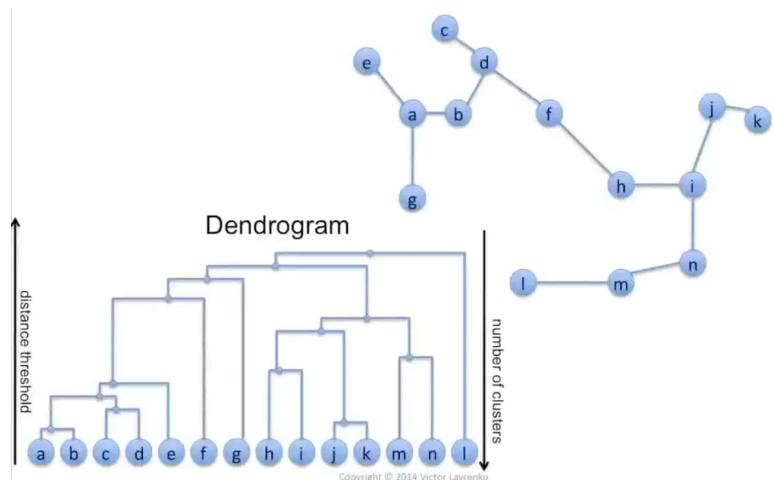
20.1 Hierarchické shlukování

Rozlišujeme následující přístupy:

1. **divizní** (vycházíme z celku, jednoho shluku, a ten dělíme),
2. **agglomerativní** (vycházíme z jednotlivých objektů, shluků o jednom členu, a ty spojujeme).

Výhody/nevýhody:

- + **Není třeba předem specifikovat počet shluků.**
- + Uživatel dokáže často dobře hierarchické struktury interpretovat (odpovídají intuici).
- V každém kroku řeší **pouze lokálně nejlepší řešení**, nebere odhad na další postup.
- **Problém s rozsáhlými daty**, neboť dolní odhad složitosti je $O(n^2)$.



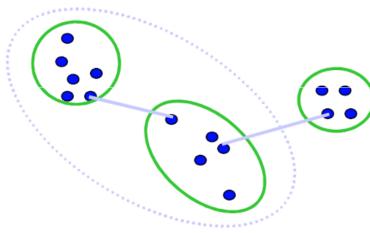
20.1.1 Dendrogram



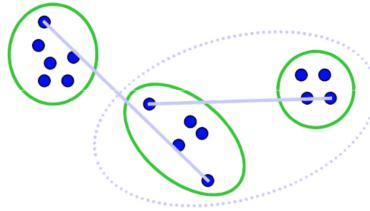
20.1.2 Metody agglomerativního shlukování (měření vzdálenosti mezi shluky)

Nevýhodou těchto metod je, že mohou vzniknout nejednoznačnosti už na začátku shlukování, které se projeví až později ve velkých shlucích. Předchozí kroky není možné změnit.

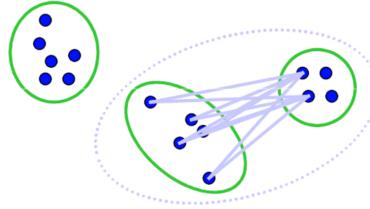
- **Single linkage** (nearest neighbor) – vzdálenost shluků je určena vzdáleností **dvou nejbližších** prvků z různých shluků. Použití této metody vede k tomu, že jsou objekty taženy k sobě, výsledkem jsou dlouhé řetězy menších shluků.



- **Complete linkage** (furthest neighbor) – vzdálenost shluků je určena naopak vzdáleností **dvou nejvzdálenějších** prvků z různých shluků. Funguje dobře, obvykle tvoří poměrně kompaktní shluky.



- **Average linkage** (průměrná vazba) – vzdálenost shluků je určena jako **průměr vzdáleností všech párů objektů** z různých shluků. Může být ve **vážené** i **nevážené** podobě. Nejčastěji používaná míra pro vzdálenost.



- **Centroidní metoda** – pro spočítání nepodobnosti objektů se v této metodě využívá **euklidovská metrika**, v které se změří vzdálenosti **těžíšť shluků** nebo objektů. Následně dojde ke sloučení shluků, které mají **nejmenší vzdálenost mezi těžíšti**. Může být nevážená (mediánová metoda) nebo vážená (váží se podle velikosti shluku).
- **Wardova metoda** – metoda **založena na ztrátě informací**, která vzniká při shlukování. Kritériem pro shlukování je celkový součet druhých mocnin **odchylek** každého objektu od těžíště shluku, do kterého náleží. Hodí pro práci s objekty, které mají stejný rozměr proměnných.

20.1.3 Metody divizního shlukování

Divizní metody berou množinu objektů, kterou mají zpracovat, jako **jeden shluk**, ten dále dělí na menší shluky a tím vytváří hierarchický systém. Každý shluk je rozdělen na dva nové, tak aby byl rozklad optimální vůči nějakému kritériu, a na konci tohoto postupu budou všechny shluky jednoprvkové.

Tento postup je kvůli **exponenciální časové složitosti** (nalezení optimálního rozkladu množiny n objektů vyžaduje prozkoumat až $2^{n-1} - 1$ možností) prakticky proveditelný jen pro malý počet objektů.

MacNaughton–Smithova metoda

Tato metoda se snaží **snížit časovou náročnost** divizních algoritmů až na **kvadratickou**, ale za cenu toho, že výsledné rozdělení **nemusí být optimální**. Je tedy aplikovatelná i na rozsáhlejší množiny objektů při nevelkých náročích na čas počítače.

V porovnání s aglomerativními přístupy je ale **stále pomalejší**. Pomocí středních vzdáleností se vybere objekt uvnitř shluku, který vytvoří nový shluk a na základě rozdílu středních vzdáleností objektů z původního a objektů z nového shluku se objekt přiřadí do nového shluku, zůstane v původním. Výhodou oproti aglomerativnímu přístupu jsou **jednoznačnější výsledky** pro větší shluky.

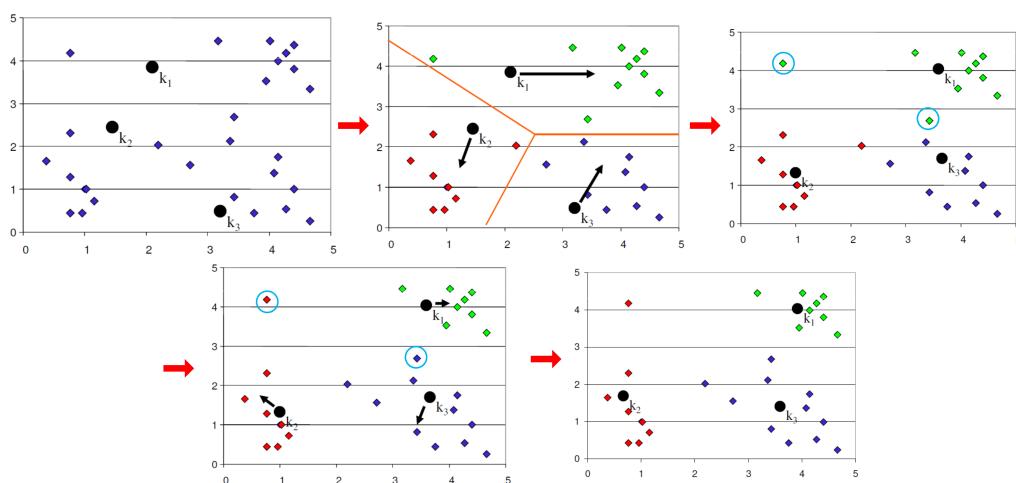
20.2 Nehierarchické shlukování

- Takový postup, při němž se každý objekt vloží do jednoho z k disjunktních shluků.
- Předpokládá se, že uživatel předem stanoví k , tj. požadovaný **cílový počet shluků**.

20.2.1 K-means

K-means **minimalizuje průměrnou vzdálenost mezi prvky** téhož shluku.

1. Stanov **požadovaný počet k** shluků.
2. Vyber náhodně výchozích k jader.
3. Přiřaď každému z N objektů číslo shluku, které odpovídá číslu **nejbližšího jádra**.
4. **Předefinuj pozice jader** všech k shluků tak, že bude použit **průměr hodnot** prvků v daném shluku.
5. Opakuj kroky 3. a 4. až do situace, kdy se příslušnost do shluků stabilizuje (po iteraci není žádný objekt zařazen do jiného shluku než před ní).



Výhody

- + **Jednoduchý** (lehká implementace i ladění).
- + Intuitivní objektivní funkce, která optimalizuje podobnost uvnitř shluků.
- + Poměrně **efektivní**: složitost $O(TKMN)$, kde m je počet objektů, K je počet shluků, N počet atributů a T počet iterací. Obvykle bývají hodnoty T a $K \ll m$.

Nevýhody

- Použitelné, jen tam, kde **umíme spočítat průměr**. Co kategorická data?
- Velmi **záleží na inicializaci** – nebezpečí uvíznutí v lokálním minimu.
- **Požaduje se znalost počtu shluků**.
- Nevhodné pro zašuměná data s výjimkami (outliers).
- Nevhodné pro situace, kdy shluky nemají konvexní tvar .

20.2.2 Odhad počtu shluků

Sledujeme **jak rychle klesá** hodnota objektivní funkce (výpočet vzdálenosti) pro zvyšující se počet jader k . Obecně hledáme „prudký pohyb“ v poklesu, který značí použití dané hodnoty k (jakmile přestane k prudce klesat, dosáhli jsme požadovaného počtu jader).

21 Náhodná veličina. Základní typy náhodných veličin. Funkce určující rozdělení náhodných veličin.

- **Náhodný pokus** – děj, jehož výsledek není předem jednoznačně určen podmínkami, za nichž probíhá.
- **Náhodný jev** – tvrzení o výsledku náhodného pokusu, přičemž o pravdivosti tohoto tvrzení lze po ukončení pokusu rozhodnout. Náhodná veličina „NV“ – číselné vyjádření výsledku náhodného pokusu.

21.1 Náhodná veličina

- Funkce, která **každému elementárnímu jevu** $\omega \in \Omega$ **přiřadí reálné číslo** (převádí elementární jevy (abstraktní objekty) na čísla).
- Obvykle značíme velkými písmeny.
- **Hodnota** $X(\omega)$ NV X **závisí** na tom, který **elementární jev** ω **nastal** \rightarrow víme-li, který elementární jev ω nastal, známe hodnotu $X(\omega)$ NV X .

Příklady

X ... číselný výsledek hodu kostkou

Náhodný pokus: Hod kostkou.

Náhodný jev: Padne sudé číslo. ($X \in \{2; 4; 6\}$)

X ... rychlosť pripojenia k internetu (Mb/s)

Náhodný pokus: Měření rychlosť pripojenia k internetu (download).

Náhodný jev: Rychlosť pripojenia k internetu je vyšší než 20 Mb/s. ($X > 20$)

X ... počet dívek mezi 1 000 náhodně vybranými dětmi

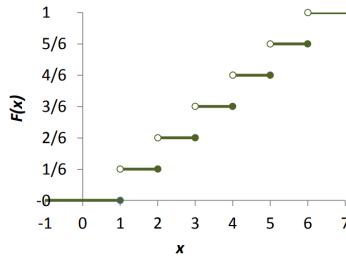
Náhodný pokus: Náhodný výběr 1 000 dětí a zjištění počtu dívek mezi nimi.

Náhodný jev: Mezi 1 000 náhodně vybranými dětmi bude více než 500 dívek. ($X > 500$)

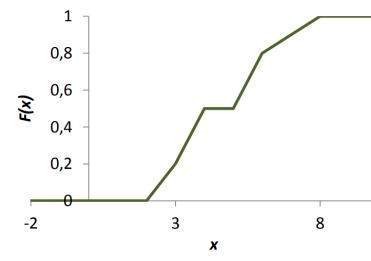
21.2 Distribuční funkce $F(x)$

- Distribuční funkce $F(t)$ udává pravděpodobnost, že náhodná veličina X bude **menší než** dané reálné číslo t .

$$F(t) = P(X < t)$$
- Distribuční funkce jednoznačně určuje rozdělení NV, tj. známe-li distribuční funkci, umíme určit pravděpodobnost $P(X \in M)$ pro libovolnou $M \subset \mathbb{R}$.



Ukázka distribuční funkce diskrétní náhodné veličiny



Ukázka distribuční funkce spojité náhodné veličiny

21.3 Základní typy náhodných veličin

- **Diskrétní NV** (nabývá spočetně mnoha hodnot)
- **Spojité NV** (jakákoliv hodnota na daném intervalu)

21.4 Diskrétní náhodná veličina („DNV“)

- Může nabývat spočetně mnoha hodnot (počet dní hospitalizace, počet dní nemocenské, počet zákazníků v lékárně během jednoho dne..).
- DNV X s distribuční funkcí $F_x(t)$ je charakterizována **Pravděpodobnostní funkcí** $P(X = x_i)$, tj. funkcí pro níž platí:

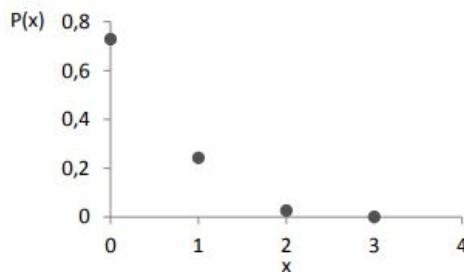
$$F_x(t) = \sum_{x_i < t} P(X = x_i) = \sum_{x_i < t} P(x_i).$$

21.4.1 Pravděpodobnostní funkce

- $P(x_i) \geq 0$
- $\sum_i P(X = x_i) = 1$
- Lze zadat předpisem, tabulkou, grafem.

$$\forall x \in \{0; 1; 2; 3\}: P(X = x) = \binom{3}{x} \cdot 0,1^x \cdot 0,9^{3-x}$$

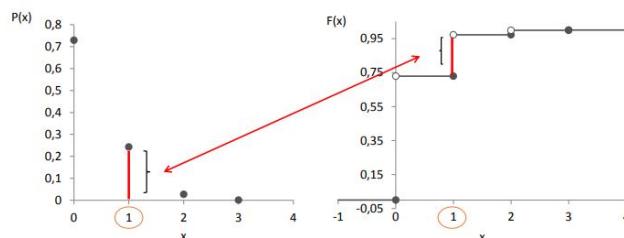
x	0	1	2	3
P(x)	0,729	0,243	0,027	0,001



⇒ *Příklad* – Při ověřování kvality výroby jsou náhodně vybrány dva výrobky a je testována jejich kvalita. Počet vadných výrobků mezi vybranými modelujeme náhodnou veličinou X. Z dlouhodobého pozorování jsou známy údaje uvedené v následující tabulce.

Počet vadných výrobků	Pravděpodobnost
0	0,25
1	0,50
2	0,25

Určete pravděpodobnostní a distribuční funkci počtu vadných výrobků v testovaném vzorku.

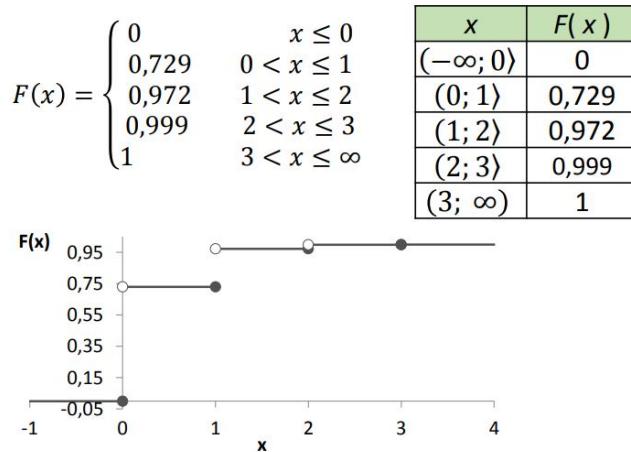


- Body nespojitosti** distribuční funkce jsou body, v nichž je pravděpodobnostní funkce nenulová.
- $P(X = a) = \lim_{x \rightarrow a^+} F(x) - F(a)$, tj. velikost „skoku“ distribuční funkce v bodech nespojitosti je rovna příslušným hodnotám pravděpodobnostní funkce.

21.4.2 Distribuční funkce

- $F(t) = \sum_{x_i < t} P(x_i)$

- Lze zadat předpisem, tabulkou, grafem.

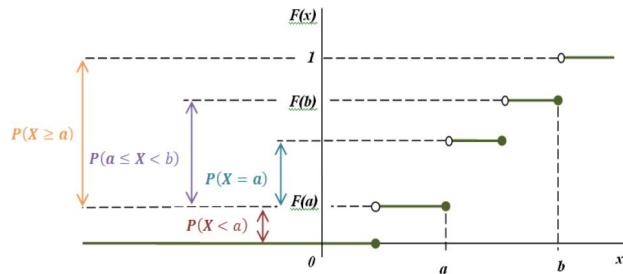


- Distribuční funkce $F(t)$ **udává pravděpodobnost**, že náhodná veličina X bude **menší než** dané reálné číslo t .

$$F(t) = P(X < t)$$

Vlastnosti distribuční funkce

- $0 \leq F(t) \leq 1$,
- je neklesající,
- je zleva spojitá,
- má nejvýše spočetně mnoho bodů nespojitosti,
- $F(t) \rightarrow 0$ pro $t \rightarrow -\infty$ („začíná“ v 0),
- $F(t) \rightarrow 1$ pro $t \rightarrow \infty$ („končí“ v 1).



$$P(X = a) = \lim_{x \rightarrow a^+} F(x) - F(a)$$

21.5 Spojitá náhodná veličina („SNV“)

- Náhodná veličina X má spojité rozdělení pravděpodobnosti (zkráceně „je spojitá“) právě když má spojitu distribuční funkci.
- Mohou **nabývat všech hodnot na nějakém intervalu** (mají spojitu distribuční funkci) (doba do remise onemocnění, výška, váha, BMO, IQ, vitální kapacita plic, chyba měření...).
- SNV X s distribuční funkcí $F_x(t)$ je charakterizována **hustotou pravděpodobnosti** $f(x)$, tj. funkcí pro níž platí:

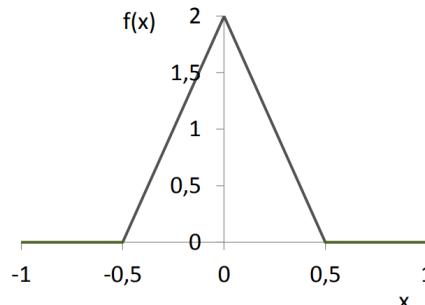
$$F_x(t) = \int_{-\infty}^t f(x) dx.$$

21.5.1 Hustota pravděpodobnosti $f(x)$

$$F(x) = \int_{-\infty}^t f(x) dx \Rightarrow f(x) = \frac{dF}{dx}$$

Vlastnosti hustoty pravděpodobnosti

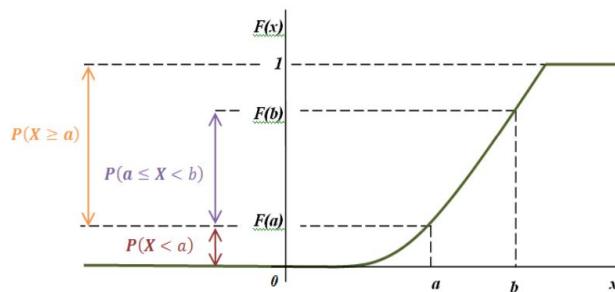
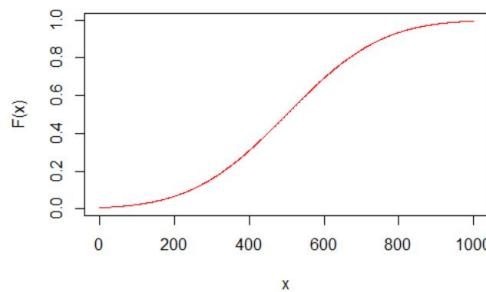
- $f(x)$ je reálná nezáporná funkce,
- $\int_{-\infty}^{\infty} f(x) dx = 1$ (plocha pod křivkou hustoty je 1),
- $\lim_{x \rightarrow -\infty} f(x) = 0$ („začíná v 0“),
- $\lim_{x \rightarrow \infty} f(x) = 0$ („končí v 0“).



Obrázek 21.1: $f(x)$ může nabývat hodnot vyšších než 1

21.5.2 Distribuční funkce

$$F_x(t) = \int_{-\infty}^t f(x) dx \quad P(X = a) = 0$$



Obrázek 21.2: Vztah mezi pravděpodobností a distribuční funkcí

21.6 Číselné charakteristiky NV

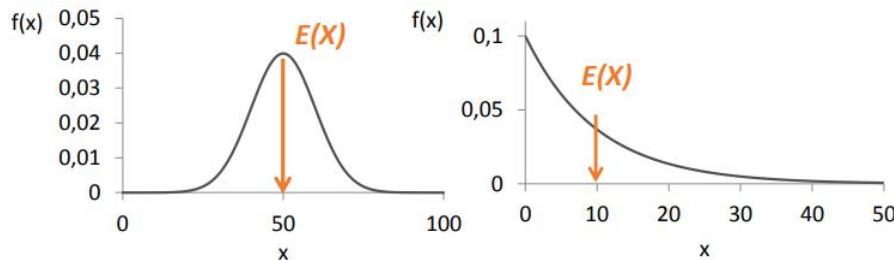
- Distribuční funkce (pravděpodobnostní funkce, hustota pravděpodobnosti) popisují rozdělení NV jednoznačně, do všech podrobností.
- Někdy nás zajímá pouze některý aspekt NV, který se dá popsat jedním číslem:
 - očekávaná hodnota NV,
 - variabilita možných hodnot,
 - hodnota, pod níž leží pouze malé množství hodnot NV,

- šíkmost rozdělení,
- koncentrace hodnot NV kolem očekávané hodnoty (špičatost rozdělení).
- **Obecný moment r-tého řádu** (značí se μ_r nebo $E(X^r)$, pro $r=1,2,\dots$)
pro diskrétní NV: $\mu_r = \sum_{(i)} x_i^r \cdot P(x_i)$
pro spojitou NV: $\mu_r = \int_{-\infty}^{\infty} x^r \cdot f(x) dx$
- **Střední hodnota** (expected value, mean, značí se jako $\mathbf{E}(X)$ nebo μ)
pro diskrétní NV: $E(X) = \mu = \sum_{(i)} x_i \cdot P(x_i)$
pro spojitou NV: $E(X) = \mu = \int_{-\infty}^{\infty} x \cdot f(x) dx$
- **Centrální moment r-tého řádu μ'_r** (značíme $\mu'_r = E[(X - E(X))^r]$)
pro diskrétní NV: $\mu'_r = \sum_{(i)} (x_i - E(X))^r \cdot P(x_i)$
pro spojitou NV: $\mu'_r = \int_{-\infty}^{\infty} (x - E(X))^r \cdot f(x) dx$
- **Rozptyl** (dispersion, variance; značí se μ'_2 nebo $D(X)$ nebo σ^2)
pro diskrétní NV: $D(X) = \mu'_2 = \sum_{(i)} (x_i - E(X))^2 \cdot P(x_i)$
pro spojitou NV: $D(X) = \mu'_2 = \int_{-\infty}^{\infty} (x - E(X))^2 \cdot f(x) dx$

21.6.1 Význam střední hodnoty

Střední hodnotu $E(X)$ náhodné veličiny X lze chápat jako:

- průměrnou (očekávanou) hodnotu NV X , kolem níž hodnoty NV kolísají,
- míru polohy, populační průměr,
- vážený průměr všech možných hodnot ($E(X) = \sum_{(i)} x_i \cdot P(x_i)$),
- „těžiště“ možných hodnot.



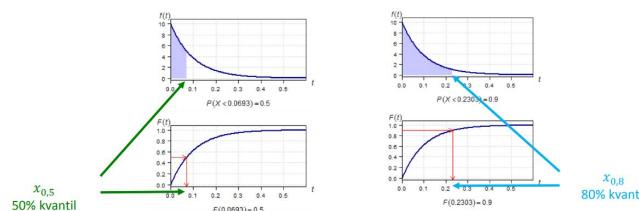
21.6.2 Kvantity

p -kvantil x_p (také $100p\%$ -ní kvantil je číslo, pro které platí):

$$P(X < x_p) = p.$$

$$\Rightarrow F(x_p) = p \Rightarrow x_p = F^{-1}(p)$$

(tj. kvantilová funkce $F^{-1}(p)$ je funkcií inverzní k distribuční funkci $F(x_p)$)



Kvantity obvykle určujeme pouze pro SNV. Význačné kvantity:

- **Kvartily**

Dolní kvartil $x_{0,25}$

Medián $x_{0,5}$

Horní kvartil $x_{0,75}$

- **Decily** – $x_{0,1}; x_{0,2}; \dots; x_{0,9}$
- **Percentily** – $x_{0,01}; x_{0,02}; \dots; x_{0,99}$
- **Minimum** x_{min} a **Maximum** x_{max}

21.6.3 Modus

Modus \hat{x} – typická hodnota náhodné veličiny

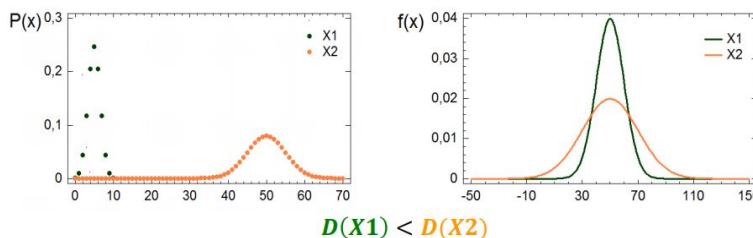
pro diskrétní NV: $\forall x_i : P(X = \hat{x}) \geq P(X = x_i)$ (tzn. modus je taková hodnota DNV, v níž $P(x_i)$ nabývá svého maxima)

pro spojité NV: $\forall x_i : f(\hat{x}) \geq f(x)$ (tzn. modus je taková hodnota SNV, v níž $f(x)$ nabývá svého maxima)

- pro diskrétní NV – $x_i : P(X = \hat{x}) \geq P(X = x_i)$ (tzn. modus je taková hodnota DNV, v níž $P(x_i)$ nabývá svého maxima).
- pro spojité NV – $x_i : f(\hat{x}) \geq f(x)$ (tzn. modus je taková hodnota SNV, v níž $f(x)$ nabývá svého maxima).
- Modus není témoto podmínkami určen jednoznačně, tzn. náhodná veličina může mít několik modů (např. výsledek hodu kostkou). Má-li NV právě jeden modus, mluvíme o **unimodálním rozdělení NV**.
- Má-li NV unimodální symetrické rozdělení, pak $E(X) = x_{0,5} = \hat{x}$.

21.6.4 Význam rozptylu

- Míra variability dat kolem střední hodnoty.
- Střední kvadratická odchylka od střední hodnoty ($D(X) = E(X - E(X))^2$).
- Malý rozptyl \approx hodnoty NV se s vysokou pravděpodobností objevují blízko $E(X)$.
- Velký rozptyl \approx hodnoty NV se často objevují ve velké vzdálenosti od $E(X)$.
- Jednotka rozptylu je kvadrátem jednotky náhodné veličiny.



21.6.5 Směrodatná odchylka σ

$$\sigma(Y) = \sqrt{D(X)}$$

Jedná se o odmocninu rozptylu náhodné veličiny. Směrodatná odchylka **neumožňuje** srovnávat variabilitu náhodných veličin **měřených v různých jednotkách!**

21.6.6 Variační koeficient

Variační koeficient γ je definován pouze pro **nezáporné** náhodné veličiny.

$$\gamma = \frac{\sigma}{\mu'}, \text{ resp. } \frac{\sigma}{\mu'} \cdot 100[\%]$$

- Variační koeficient – **směrodatná odchylka v procentech** střední hodnoty.
- Čím nižší var. koeficient, tím **homogennější** soubor.
- $V_X > 50\%$ značí silně rozptýlený soubor.

22 Vybraná rozdělení diskrétní a spojité náhodné veličiny - binomické, hypergeometrické, negativně binomické, Poissonovo, exponenciální, Weibullovo, normální rozdělení.

- **Náhodná veličina** – číselné vyjádření výsledku náhodného pokusu.
- Základní typy NV – **diskrétní NV, spojité NV**.
- **Rozdělení pravděpodobnosti** – Předpis, který jednoznačně určuje všechny pravděpodobnosti typu $P(X \in M)$, kde $M \subset \mathbb{R}$
(tj. $P(X = a), P(X < a), P(X > a), P(a < X < b), \dots$, kde $a, b \in \mathbb{R}$).
 - Rozdělení pravděpodobnosti **DNV** – distribuční funkci $F(x) = P(X < x)$, resp. **pravděpodobnostní funkcí** $P(x_i)$.
 - Rozdělení pravděpodobnosti **SNV** – distribuční funkci $F(x) = P(X < x)$, resp. **hustotou pravděpodobnosti** $f(x)$.
- **Číselné charakteristiky pro popis NV** – střední hodnota $\mathbf{E}(\mathbf{X})$, rozptyl $\mathbf{D}(\mathbf{X})$, směrodatná odchylka $\sigma(\mathbf{X})$, p-quantily \mathbf{x}_p

22.1 Diskrétní náhodná veličina

Příklady

- počet šroubů typu M10 mezi 10 vybranými (víme-li, že do dodávky 100 šroubů bylo omylem zařazeno 20 šroubů typu M50)
- počet pacientů (z 10 očkovaných) u nichž byla použita prošlá očkovací látka (víme-li, že v balení bylo 20 dávek očkovací látky, přičemž 5 z nich bylo prošlých)
 - ▷ **Obecně:** – počet úspěchů v n (závislých) pokusech
- počet správně přenesených bitů předtím než dojde ke 4. chybě (víme-li, že pravděpodobnost chybného přenosu bitu je 0,12)
- počet dobrovolníků, které budeme muset testovat dříve než najdeme 5 dárců s krevní skupinou AB (předpokládejme, že dobrovolníci neznají svou krevní skupinu, pravděpodobnost výskytu krevní skupiny (populační frekvence) krevní skupiny AB je 0,05)
 - ▷ **Obecně:** – počet (nezávislých) pokusů do k . úspěchu
- počet škrábanců na $1m^2$ lakovaného povrchu (víme-li, že průměrně lze očekávat 3 škrábance na $10m^2$)
- počet červených krvinek v $10ml$ krve ženy (víme-li, že u průměrně lze pozorovat $4,8 \cdot 10^{12}$ červených krvinek v $1l$ krve (u žen))
 - ▷ **Obecně:** – počet událostí v časovém intervalu, na ploše, v objemu

22.1.1 Bernoulliho pokusy

- **Posloupnost nezávislých pokusů majících pouze dva možné výsledky** (takových pokusů, kdy úspěch v libovolné skupině pokusů neovlivňuje pravděpodobnost úspěchů v pokusu, který do této skupiny nepatří).
- **v nichž jev A (úspěch) nastává s pravděpodobností π** a neúspěch s pravděpodobností $1 - \pi$.

Příklad: Předpokládejme, že pravděpodobnost narození dívky je 0,49. Jaká je pravděpodobnost toho, že mezi čtyřmi dětmi v rodině je právě jedna dívka?

- X... počet dívek mezi 4 dětmi
- D ... narodí se dívka, $P(D) = \pi$
- \bar{D} ... narodí se kluk, $P(\bar{D}) = 1 - \pi$

$$(X = 1) \dots \{D\bar{D}\bar{D}\bar{D}, \bar{D}D\bar{D}\bar{D}, \bar{D}\bar{D}D\bar{D}, \bar{D}\bar{D}\bar{D}D\}$$

$$P(D\bar{D}\bar{D}\bar{D}) = P(\bar{D}D\bar{D}\bar{D}) = P(\bar{D}\bar{D}D\bar{D}) = P(\bar{D}\bar{D}\bar{D}D) = \pi \cdot (1 - \pi)^3$$

$$P(X = 1) = P(D\bar{D}\bar{D}\bar{D} \cup \bar{D}D\bar{D}\bar{D} \cup \bar{D}\bar{D}D\bar{D} \cup \bar{D}\bar{D}\bar{D}D) = \binom{4}{1} \cdot \pi \cdot (1 - \pi)^3$$

$$P(X = 1) = 0,260$$

22.1.2 Binomické rozdělení

- X .. počet úspěchů v n Bernoulliho pokusech

$$X \sim Bi(n; \pi)$$

n – počet pokusů

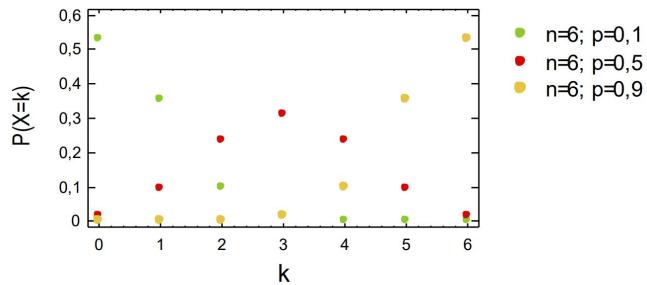
π – pravděpodobnost úspěchu

Počet příznivých realizací posloupnosti Bernoulliho pokusů.

Pravděpodobnostní funkce: $P(X = k) = \binom{n}{k} \pi^k (1 - \pi)^{n-k}$

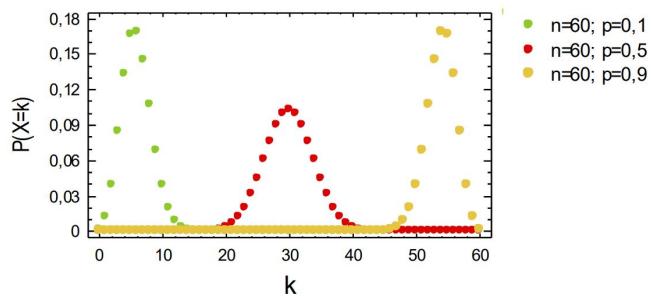
Pravděpodobnost výskytu jedné příznivé realizace posloupnosti Bernoulliho pokusů.

- **Střední hodnota:** $E(X) = n\pi$
- **Rozptyl:** $D(X) = n\pi(1 - \pi)$
- Vlastnosti



- $\pi < 0,5$, malé n \Rightarrow poz. zešikmené r.
- $\pi = 0,5$ \Rightarrow symetrické r.
- $\pi < 0,5$, malé n \Rightarrow neg. zešikmené r.

Obrázek 22.1: $\pi \neq 0,5$, s rostoucím n se rozdělení stává více a více symetrickým



- *Příklad* Mezi 200 vajíčky určenými pro prodej v jisté maloobchodní prodejně je 50 vajíček prasklých. Jaká je pravděpodobnost, že vybereme-li si náhodně 20 vajec, bude 8 z nich prasklých?

- X .. počet prasklých vajec z 20 vybraných (výběr bez vracení)

$$\begin{array}{c}
 \text{celkem vajec} \\
 200 \\
 \swarrow \quad \searrow \\
 \text{prasklých vajec} \quad \text{neprasklých vajec} \\
 50 \quad 150 \\
 \\
 P(X = 8) = \frac{\binom{50}{8} \binom{150}{12}}{\binom{200}{20}}
 \end{array}$$

počet příznivých možností – vybíráme 8 prasklých z 50 prasklých a zároveň 12 ze 150 neprasklých
 počet všech možností – vybíráme 20 vajec z 200

22.1.3 Hypergeometrické rozdělení

- X ... počet prvků se sledovanou vlastností ve výběru n prvků
- V souboru N prvků je M s danou vlastností a zbylých $(N - M)$ prvků tuto vlastnost nemá. Postupně vybereme ze souboru n prvků, z nichž žádný nevracíme zpět.

$$X \sim H(N; M; n)$$

N – rozsah základního souboru

M – počet prvků s danou vlastností

n – rozsah výběru

- **Pravděpodobnostní funkce:** $P(X = k) = \frac{\binom{M}{k} \binom{N-M}{n-k}}{\binom{N}{n}}$
 - Počet příznivých možností, tj. počet možností jak vybrat k prvků s danou vlastností z M a zároveň $(n - k)$ prvků, které uvedenou vlastnost nemají z $(N - M)$ prvků.
 - Počet všech možností, jak vybrat n prvků z N (nezáleží na pořadí).
- **Střední hodnota:** $E(X) = n \cdot \frac{M}{N}$
- **Rozptyl:** $D(X) = n \cdot \frac{M}{N} \left(1 - \frac{M}{N}\right) \left(\frac{N-n}{N-1}\right)$
- **Možnost approximace** – je-li n/N , tzv. **výběrový poměr**, menší než 0,05, lze hypergeometrické rozdělení nahradit binomickým s parametry n a M/N .

$$\left(\frac{n}{N} < 0,05\right) \Rightarrow [H(N; M; n) \approx Bi(n; \frac{M}{N})]$$

22.1.4 Negativně binomické (Pascalovo) rozdělení

- X ... počet Bernoulliho pokusů do k . výskytu události (úspěchu), včetně k . výskytu

$$X \sim NB(k; \pi)$$

k – požadovaný počet úspěchů (výskytu události)

π – pravděpodobnost úspěchu

- **Pravděpodobnostní funkce:**

$$P(X = n) = \binom{n-1}{k-1} \pi^{k-1} (1-\pi)^{((n-1)-(k-1))} \cdot \pi$$

pravděpodobnost k . úspěchu
 pravděpodobnost $k-1$ úspěchů
 v $n-1$ Bernoulliho pokusech

- **Střední hodnota:** $E(X) = \frac{k}{\pi}$
- **Rozptyl:** $D(X) = \frac{k(1-\pi)}{\pi^2}$
- V případě negativně binomické náhodné veličiny není definice ustálená. Některí statistici (popř. statistický software) ji definují jako **počet neúspěchů před k . úspěchem**.

22.1.5 Poissonův proces

Bodový proces

- Sledujeme chod procesu, v němž čas od času dochází k nějaké význačné události



- Registrujeme počet událostí $N(s; t)$ v časovém intervalu $< s; s + t >$
- **Rychlosť výskytu události** λ je parametrem Poissonova procesu
- Poissonův proces lze obdobně jako v časovém intervalu definovat na libovolné uzavřené prostorové oblasti (na ploše, v objemu).

Jako **Poissonův proces** označujeme proces, který je:

- **ordinální** – pravděpodobnost výskytu více než jedné události v limitně krátkém časovém intervalu ($t \rightarrow 0$) je nulová. (tzv. **řídké jevy**),
- **stacionální** – rychlosť výskytu událostí λ je konstantní v průběhu sledovaného intervalu,
- **beznásledný** – pravděpodobnost výskytu událostí není závislá na čase, který uplynul od minulé události.

22.1.6 Poissonovo rozdělení

- **X ... počet výskytu události v uzavřené oblasti (v čase, na ploše, v objemu)**
- náhodný pokus jako Poissonův proces (nezávislé události probíhající v čase t , s rychlosťí výskytu λ ; popř. nezávislé události objevující se na ploše t , resp. v objemu t s hustotou výskytu λ).

$$X \sim Po(\lambda t)$$

- **Pravděpodobnostní funkce:** $P(X = k) = \frac{(\lambda t)^k}{k!} e^{-\lambda t}$
- **Střední hodnota:** $E(X) = \frac{k}{\pi}$
- **Rozptyl:** $D(X) = \frac{k(1-\pi)}{\pi^2}$

22.2 Spojitá náhodná veličina

- Hustota pravděpodobnosti $f(x)$ – funkce $f(x)$ je **hustotou pravděpodobnosti** (na intervalu $a \leq x \leq b$), jestliže splňuje následující podmínky:
 - $f(x) \geq 0; x \in \mathbb{R}$
 - plocha pod křivkou hustoty je rovna 1 ($\int_{-\infty}^{\infty} f(x)dx = 1$).

22.2.1 Exponenciální rozdělení

- **X ... délka časových intervalů mezi událostmi v Poissonově procesu**

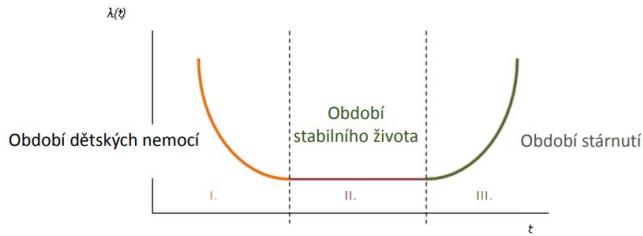
$$X \sim Exp(\lambda)$$

- Předpokládá konstantní intenzitu události $\lambda(t)$ - rozdělení „bez paměti“.
- **Hustota pravděpodobnosti:** $f(t) = \begin{cases} \lambda \cdot e^{-\lambda t} & t > 0 \\ 0, & t \leq 0. \end{cases}$
- **Distribuční funkce:** $F(t) = \begin{cases} 1 - e^{-\lambda t} & t > 0 \\ 0, & t \leq 0. \end{cases}$
- **Střední hodnota:** $E(X) = \frac{1}{\lambda}$
- **Rozptyl:** $D(X) = (E(X))^2 = \frac{1}{\lambda^2}$
- **Riziková funkce** (intenzita poruch) $\lambda(t)$
 - Pro nezápornou náhodnou veličinu X se spojitým rozdělením popsáným distribuční funkcí $F(t)$ definujeme pro $F(t) \neq 1$ rizikovou funkci jako

$$\lambda(t) = \frac{f(t)}{1 - F(t)}.$$

- $\lambda(t)$ – Představuje-li náhodná veličina X dobu do poruchy nějakého zařízení, pak pravděpodobnost, že pokud do času t nedošlo k žádné poruše, tak k ní dojde v následujícím krátkém úseku délky Δt , je přibližně

$$P(t \geq X < t + \Delta t | X > t) = \lambda(t) \cdot \Delta t$$



- *Příklad* Střední doba čekání zákazníka na obsluhu v prodejně je 50 sekund. Doba čekání se řídí exponenciálním rozdělením (pravděpodobnost, že zákazník nebude obslužen klesá s rostoucím časem exponenciálně). Jaká je pravděpodobnost, že náhodný zákazník bude obslužen dříve než za 30 sekund?

X – doba čekání na obsluhu

$$X \sim \text{Exp}(\lambda = \frac{1}{50} \text{ s}^{-1}), E(X) = \frac{1}{\lambda} = 50 \text{ s}$$

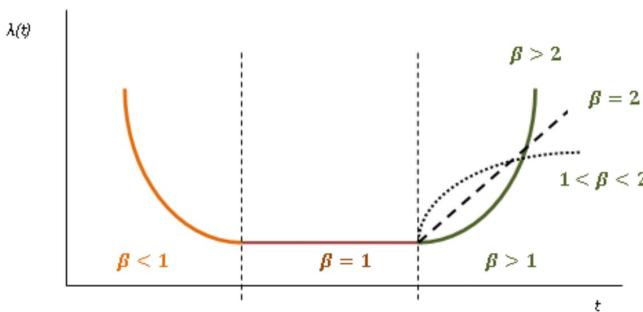
22.2.2 Weibullovo rozdělení

- X ... délka časových intervalů mezi událostmi v Poissonově procesu
- Zobecnění exponenciálního rozdělení
- Tímto rozdělením lze modelovat i dobu do výskytu události u systémů (jedinců), které jsou v období dětských nemocí, resp. období stárnutí.

$$X \sim W(\theta; \beta)$$

θ – parametr měřítka (scale) $\theta = \frac{1}{\lambda}; \theta > 0$
 β – parametr tvaru (shape); $\beta > 0$

- **Hustota pravděpodobnosti:** $f(t) = \begin{cases} \beta \lambda^\beta t^{\beta-1} e^{-(\lambda t)^\beta}, & t > 0 \\ 0, & t \leq 0. \end{cases}$
- **Distribuční funkce:** $F(t) = \begin{cases} 1 - e^{-(\lambda t)^\beta} & t > 0 \\ 0, & t \leq 0. \end{cases}$
- **Riziková funkce:** $\lambda(t) = \begin{cases} \beta \lambda^\beta t^{\beta-1} & t > 0 \\ 0, & t \leq 0. \end{cases}$



Obrázek 22.2: Riziková funkce

22.2.3 Normální rozdělení

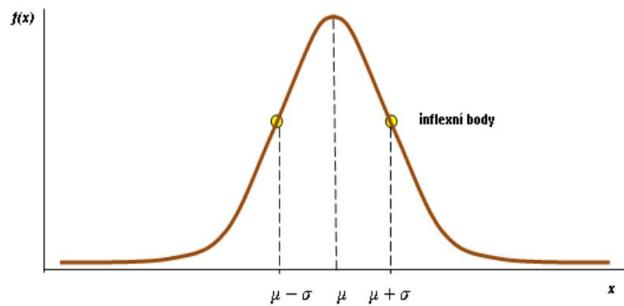
- Bývá vhodné k popisu náhodných veličin, které lze interpretovat jako aditivní výsledek mnoha nepatrných a vzájemně nezávislých faktorů (výška člověka, IQ, délka končetiny).

- Popisuje náhodné veličiny, jejichž hodnoty se symetricky shlukují kolem střední hodnoty a vytvářejí tak charakteristický tvar hustoty pravděpodobnosti známý pod názvem **Gaussova křivka**.

$$X \sim N(\mu; \sigma^2)$$

μ – střední hodnota
 σ^2 – rozptyl

- Hustota pravděpodobnosti:** $f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$



Obrázek 22.3: Riziková funkce

- Distribuční funkce:** $F(x) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{1}{2}(\frac{t-\mu}{\sigma})^2} dt$
 (integrál nelze řešit analyticky)

23 Popisná statistika. Číselné charakteristiky a vizualizace kategoriálních a kvantitativních proměnných.

Popisná statistika zjišťuje a **sumarizuje** informace, zpracovává je ve formě grafů a tabulek a vypočítává jejich **číselné charakteristiky** jako průměr, rozptyl percentily, rozpětí a pod.

23.1 Kvantitativní – Numerická proměnná

1. **Míry polohy** – určující typické rozložení hodnot proměnné (jejich rozmístění na číselné ose).

- Průměr – $\bar{x} = \frac{\sum_{i=1}^n x_i}{n}$
- Modus – střed shorthu
- Kvantity – dolní kvartil, médián, horní kvartil,..

2. **Míry variability** – určující variabilitu (rozptyl) hodnot kolem své typické polohy.

- Variační rozpětí $x_{max} - x_{min}$
- Inverkvartilové rozpětí – $IQR = x_{0,75} - x_{0,25}$
- Výběrový rozpětí
- Výběrová směrodatná odchylka
- Variační koeficient

3. **Míra šikmosti a špičatosti**

- Výběrová šikmost
- Výběrová špičatost

4. **Identifikace odlehlých pozorování**

- Vnitřní hradby dolní mez: $h_D = x_{0,25} - 1,5IQR$, horní mez: $h_H = x_{0,75} + 1,5IQR$
- Vnější hradby dolní mez: $h_D = x_{0,25} - 3IQR$, horní mez: $h_H = x_{0,75} + 3IQR$
- Z-souřadnice
- Mediánová souřadnice

23.1.1 Grafické zobrazení numerické proměnné

- Empirická distribuční funkce
- Krabicový graf (box plot)
- Číslicový histogram (lodyha s listy, stem and leaf)

23.2 Kvalitativní – Kategoriální proměnná

1. **Nominální proměnná** – nemá smysl uspořádaní.

- **Základní statistiky pro popis nominální proměnné:**
 - četnost
 - relativní četnost
 - modus
- **Grafické zobrazení nominální proměnné:**
 - histogram
 - výsečový graf

2. **Ordinální proměnná** – má smysl uspořádání.

- **Základní statistiky pro popis ordinální proměnné:**
 - četnost
 - relativní četnost

- kumulativní četnost
- relativní kumulativní četnost
- modus
- **Grafické zobrazení ordinální proměnné:**
 - histogram
 - výsečový graf
 - Lorenzova křivka
 - Paretův graf

Paretův princip – 80% následků pramení z 20% příčin.

Paretova analýza – postup vedoucí k nalezení „životně důležité menšiny“ (spektra příčin ovlivňujících rozhodujícím způsobem následky).

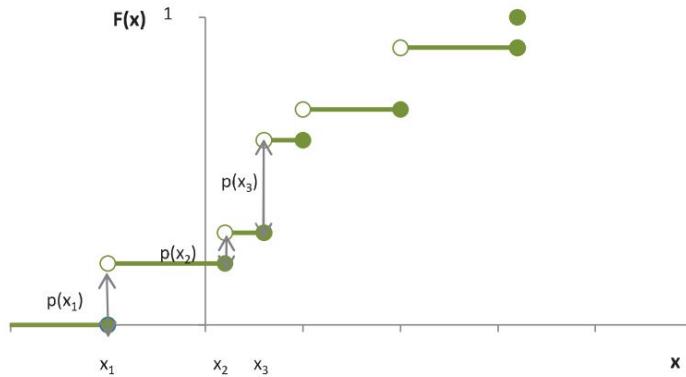
23.3 Míry polohy a variability

Snad nejpoužívanějšími mírami polohy jsou **průměry** proměnných. Průměry představují průměrnou nebo typickou hodnotu výběrového souboru. Zřejmě nejznámějším průměrem pro kvantitativní proměnnou je **aritmetický průměr**.

- **Aritmetický průměr \bar{x} (mean)**
- **Aritmetický vážený průměr**
- **Harmonický průměr** – Pro výpočet průměru v případech, kdy proměnná má charakter části z celku (úlohy o společné práci, ...).
- **Harmonický vážený průměr** – Pokud máme údaje setříděné do tabulky četností.
- **Geometrický průměr** – Pracujeme-li s kladnou proměnnou představující relativní změny (růstové indexy, cenové indexy...). $\bar{x}_G = \sqrt[n]{x_1 \cdot x_2 \cdot \dots \cdot x_n}$
- **Geometrický vážený průměr** – Pracujeme-li s kladnou proměnnou představující relativní změny (růstové indexy, cenové indexy...). $\bar{x}_G = \sqrt[n]{x_1^{n_1} \cdot x_2^{n_2} \cdot \dots \cdot x_n^{n_k}}$ kde $n = \sum_{i=1}^k n_i$.
- **Modus** – Pro **diskrétní proměnnou** definujeme modus jako **hodnotu nejčetnější varianty proměnné** (podobně jako u kvalitativní proměnné), u **spojitě** proměnné považujeme za modus \hat{x} hodnotu kolem níž je největší koncentrace hodnot proměnné. Mnohdy mluvíme o typické hodnotě proměnné. Pro určení této hodnoty využijeme tzv. **short**, což je nejkratší interval, v němž leží alespoň 50% hodnot proměnné (v případě výběru o rozsahu $n = 2k (k \in N)$ (sudý počet hodnot), leží v shorthu k hodnotě - což je 50% ($n/2$) hodnot proměnné, v případě výběru o rozsahu $n = 2k + 1 (k \in N)$ (lichý počet hodnot), leží v shorthu $k + 1$ hodnot - což je o 1 více než je 50% hodnot proměnné). **Modus pak definujeme jako střed shorthu**.
- **Výběrové kvantily** (quantile, resp. percentile) – jsou to statistiky, které charakterizují polohu jednotlivých hodnot v rámci proměnné. Podobně jako modus, jsou i výběrové kvantily **rezistentní** (odolné) vůči odlehlým pozorováním.
- Obecně je výběrový kvantil chápán jako hodnota, která rozděluje výběrový soubor na dvě části – hodnoty, které jsou menší než daný kvantil, druhá část obsahuje hodnoty, které jsou větší nebo rovno danému kvantilu. Pro určení kvantilu je nutné **výběr usporádat** od nejmenší hodnoty k největší. **Kvantily**:
 - **Dolní kvartil $x_{0,25}$** – 25%-ní kvartil (rozděluje datový soubor tak, že 25% hodnot je menších než tento kvartil a zbytek, tj. 75% větších (nebo rovných))
 - **Medián $x_{0,5}$** – 50%-ní kvartil
 - **Horní kvartil $x_{0,75}$** – 75%-ní kvartil
Kvartily dělí výběrový soubor na 4 přibližně stejně velké části.
 - **Decily – $x_{0,1}; x_{0,2}; \dots; x_{0,9}$** – Decily dělí výběrový soubor na 10 přibližně stejně četných částí.
 - **Percentily – $x_{0,01}; x_{0,02}; \dots; x_{0,99}$** – Percentily dělí výběrový soubor na 100 přibližně stejně četných částí.
- **Empirická distribuční funkce $F(x)$ pro kvantitativní proměnnou**

- Empirická distribuční funkce je monotónně rostoucí, zleva spojitu funkcí, která „skáče“ podle relativních četností příslušných jednotlivým hodnotám proměnné.
- Označme si $p(x_i)$ relativní četnost hodnoty x_i seřazeného výběrového souboru $x_1 < x_2 < \dots < x_n$. Pro empirickou distribuční funkci $F(x)$ pak platí:

$$F(x) = \begin{cases} 0 & \text{pro } x \leq x_i \\ \sum_{i=1}^j F(x) & \text{pro } x_j < x \leq x_{j+1}, 1 \leq j \leq n-1 \\ 1, & \text{pro } x_n < x \end{cases}$$



- **Interkvartilové rozpětí IQR** – Tato statistika je mírou variability souboru a je definována jako vzdálenost mezi horním a dolním kvartilem.
- **MAD** (median absolute deviation from the median) – medián absolutních odchylek od mediánu.
 1. Výběrový soubor uspořádáme podle velikosti.
 2. Určíme medián souboru.
 3. Pro každou hodnotu souboru určíme absolutní hodnotu její odchylky od mediánu.
 4. Absolutní odchylky od mediánu uspořádáme podle velikosti.
 5. Určíme medián absolutních odchylek od mediánu, tj. MAD.
- **Výběrový rozptyl s^2** („s kvadrát“, sample variance) –
 - je nejrozšířenější mírou variability výběrového souboru

$$s^2 = \frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n-1}$$

- Výběrový rozptyl je dán podílem součtu kvadrátu odchylek jednotlivých hodnot od průměru a rozsahu souboru sníženého o jedničku.
- Nevýhodou použití výběrového rozptylu jakožto míry variability je to, že jednotka této charakteristiky je **druhou mocninou** jednotky proměnné. Např. je-li proměnnou denní tržba uvedena v Kč, bude výběrový rozptyl této proměnné vyjádřen v $K\text{c}^2$.
- Následující míra variability tuto vlastnost nemá.

- **Výběrová směrodatná odchylka s** – je definována jako kladná odmocnina výběrového rozptylu

$$s = \sqrt{s^2} = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n-1}}.$$

Nevýhodou výběrového rozptylu i výběrové směrodatné odchylky je skutečnost, že neumožňují porovnávat variabilitu proměnných vyjádřených v různých jednotkách. Která proměnná má větší variabilitu – výška nebo hmotnost dospělého člověka? Na tuto otázku nám dá odpověď tzv. variační koeficient.

- **Variační koeficient V_x** – vyjadřuje relativní míru variability proměnné x . Podle níže uvedeného vztahu jej lze stanovit pouze pro proměnné, které nabývají výhradně kladných hodnot. Variační koeficient je bezrozměrný. Uvádíme-li jej v [%], hodnotu získanou z definičního vzorce vynásobíme 100%.

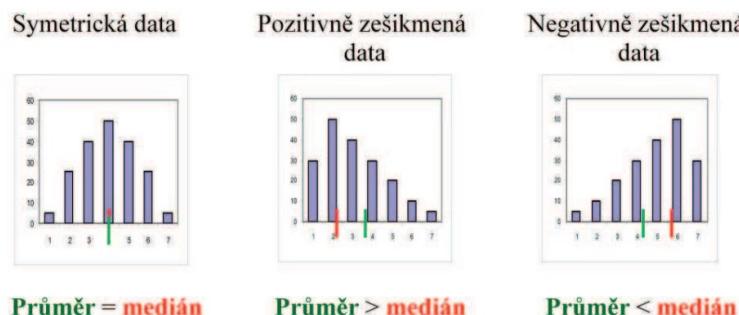
$$V_x = \frac{V}{\bar{x}}, \text{ popr. } V_x = \frac{V}{\bar{x}} \cdot 100[\%].$$

23.4 Identifikace odlehlých pozorování

- Vnitřní hradby** – Za odlehlé pozorování lze považovat takovou hodnotu x_i , která je od dolního, resp. horního kvartilu vzdálená více než 1,5 násobek interkvartilového rozpětí. Tedy: $[(x_i < x_{0,25} - 1,5 \cdot IQR) \vee (x_i > x_{0,75} + 1,5 \cdot IQR)] \Rightarrow x_i$ je odlehlým pozorováním.
- z-souřadnice (z-skóre)** Za odlehlé pozorování lze považovat takovou hodnotu x_i , jejíž absolutní hodnota z-souřadnice je větší než 3, tj. hodnota, která je od průměru vzdálenější než 3s. Tedy: $z - skore_i = \frac{x_i - \bar{x}}{s}$ $|z - skore_i| > 3 \Rightarrow |\frac{x_i - \bar{x}}{s}| > 3s \Rightarrow x_i$ je odlehlým pozorováním.
- $x_{0,5}$ -souřadnice ($x_{0,5}$ -skóre)** – Za odlehlé pozorování lze považovat takovou hodnotu x_i , jejíž absolutní hodnota mediánové souřadnice je větší než 3, tj. hodnota, která je od mediánu vzdálenější než $3 \cdot 1,483 \cdot MAD$. Tedy: $x_{0,5} - skore_i = \frac{x_i - x_{0,5}}{1,483MAD}$ $|x_{0,5} - skore_i| > 3 \Rightarrow |\frac{x_i - x_{0,5}}{1,483MAD}| > 3 \Rightarrow |x_i - x_{0,5}| > 3 \cdot 1,483MAD \Rightarrow x_i$ je odlehlým pozorováním.

23.5 Míra šikmosti a špičatosti

- Výběrová šikmost a** (skewness) – vyjadřuje asymetrii rozložení hodnot proměnné kolem jejího průměru.
 - $a = 0$ – hodnoty proměnné jsou kolem jejího průměru rozloženy symetricky
 - $a > 0$ – u proměnné převažují hodnoty menší než průměr
 - $a < 0$ – u proměnné převažují hodnoty větší než průměr

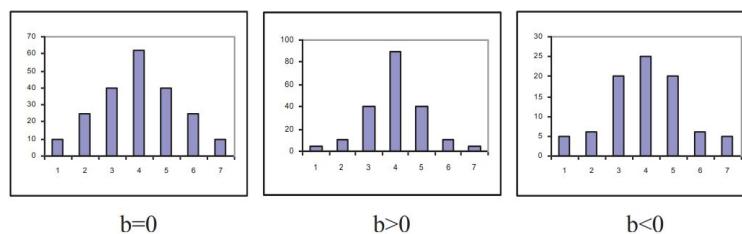


- Souvislost mezi šikmostí a charakteristikami polohy**

- Symetrické rozdělení: $\bar{x} = x_{0,5}$
- Pozitivně zešikmené rozdělení: $\bar{x} > x_{0,5}$
- Negativně zešikmené rozdělení: $\bar{x} < x_{0,5}$

- Výběrová špičatost b** (kurtosis) – vyjadřuje koncentraci hodnot proměnné kolem jejího průměru.

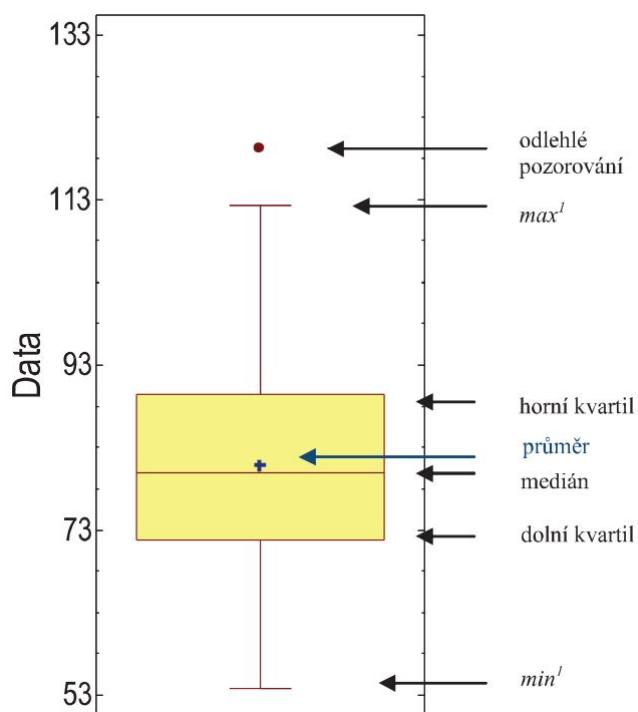
- $b = 0$ – špičatost odpovídá normálnímu rozdělení (bude definováno později)
- $b > 0$ – špičaté rozdělení proměnné
- $b < 0$ – ploché rozdělení proměnné



23.6 Grafické znázornění kvalitativní proměnné

- Krabitový graf (Box plot)**

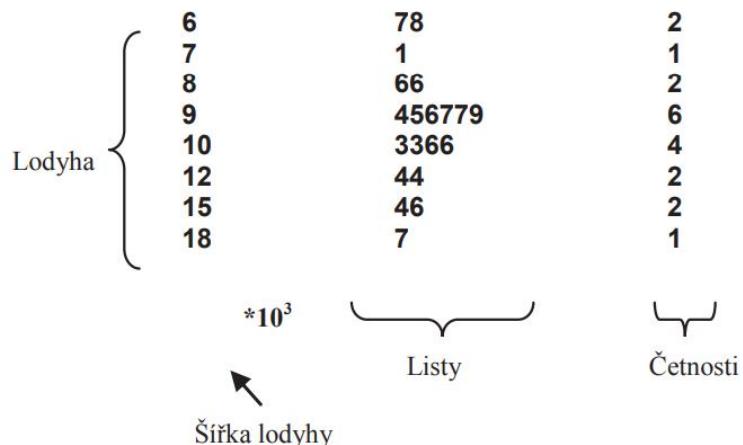
- Odlehlá pozorování jsou znázorněna jako izolované body, konec horního (popř. konec dolního) vousu představují maximum (popř. minimum) proměnné po vyloučení odlehlých pozorování, „víko“ krabice udává horní quartil, „dno“ dolní quartil, vodorovná úsečka uvnitř krabice označuje medián.
- Z polohy mediánu vzhledem ke „krabici“ lze dobře usuzovat na symetrii vnitřních 50% dat a my tak získáváme dobrý přehled o středu a rozptylenosti proměnné.



- **Číslicový histogram** (Lodyha s listy, angl. Stem and leaf plot)

- Jak jsme si ukázali, výhodou krabicového grafu je jeho jednoduchost, někdy nám však chybí informace o konkrétních hodnotách proměnné.
- Chtěli bychom proto nějak přehledně zapsat číselné hodnoty výběru a k tomu nám slouží právě číslicový histogram.
- Navíc nám tento graf dává dobrou představu o šíkosti proměnné.
- Příklad: *Představme si proměnnou představující průměrné měsíční platy zaměstnanců ve státní správě.*

Průměrný měsíční plat [Kč]
10 654, 9 765, 8 675, 12 435, 9 675, 10 343, 18 786, 15 420, 8 675, 7 132, 6 732, 6 878, 15 657, 9 754, 9 543, 9 435, 10 647, 12 453, 9 987, 10 342.



* Pro naší informaci nejsou tak důležité koruny ani desetikoruny rozdílu. V tomto případě se nám jedná přinejmenším o stokoruny.

- * Co kdybychom tedy informaci o „nedůležitých“ rádech zanedbali a znázornili setříděná data pouze na základě vyšších rádů? My jsme se rozhodli, že důležitý rád jsou pro nás stokoruny.
- * Hodnoty stojící o rád výš (v našem případě tisíce) zapíšeme setříděné pod sebe, tak, že tvoří jakýsi stonek (**lodyhu**), přičemž pod graf uvedeme tzv. **šířku lodyhy**, která udává koeficient, jímž se hodnoty uvedené v grafu násobí.
- * Druhý sloupec grafu, **listy**, budou tvořit číslice, reprezentující zvolený „důležitý“ rád, zapisované do příslušných rádků (opět seřazené podle velikosti).
- * Třetí sloupec udává absolutní četnosti příslušné daným rádkům.

23.7 Statistické charakteristiky kvalitativních (kategoriálních) proměnných

23.7.1 Nominální proměnná

Nominální proměnná nabývá v rámci souboru **různých, avšak rovnocenných kategorií**. Počet těchto kategorií nebývá příliš vysoký, a proto první statistickou charakteristikou, kterou k popisu proměnné použijeme je četnost.

- **Četnost n_i** (absolutní četnost, „frequency“) – je definována jako počet výskytu dané varianty kvalitativní proměnné. V případě, že kvalitativní proměnná ve statistickém souboru o rozsahu n hodnot nabývá k různých variant, jejichž četnosti označíme n_1, n_2, \dots, n_k , musí zřejmě platit $n_1 + n_2 + \dots + n_k = \sum_{i=1}^k n_i = n$. Chceme-li vyjádřit, jakou část souboru tvoří proměnné s některou variantou, použijeme pro popis proměnné relativní četnost.
- **Relativní četnost p_i** („relative frequency“) je definována jako $p_i = \frac{n_i}{n}$, popř. $p_i = \frac{n_i}{n} \cdot 100[\%]$. Pro relativní četnosti musí platit $p_1 + p_2 + \dots + p_k = \sum_{i=1}^k p_i = p$.
Při zpracování kvalitativní proměnné je vhodné četnosti i relativní četnosti uspořádat do tzv. **tabulky rozdělení četnosti** („frequency table“)

TABULKA ROZDĚLENÍ ČETNOSTI		
Hodnoty x_i	Absolutní četnosti	Relativní četnosti
	n_i	p_i
x_1	n_1	p_1
x_2	n_2	p_2
x_k	n_k	p_k
Celkem	$\sum_{i=1}^k n_i = n$	$\sum_{i=1}^k p_i = 1$

- **Modus** – s definujeme jako název varianty proměnné vykazující nejvyšší četnost.
- **Grafické znázornění nominální proměnné**
 - **Histogram** – je klasickým grafem, v němž na jednu osu vynášíme varianty proměnné a na druhou osu jejich četnosti. Jednotlivé hodnoty četností jsou pak zobrazeny jako výšky sloupců (obdélníků, popř. hranolů, kuželů...)
 - **Výsečový graf** – prezentuje relativní četnosti jednotlivých variant proměnné, přičemž jednotlivé relativní četnosti jsou úměrně reprezentovány plochami příslušných kruhových výsečí. (Změnou kruhu na elipsu dojde k trojrozměrnému efektu.)

23.7.2 Ordinální proměnná

Ordinální proměnná, stejně jako proměnná nominální, nabývá v rámci souboru různých slovních variant, avšak tyto varianty mají **přirozené uspořádání**, tj. můžeme určit, která je „menší“ a která „větší“.

- **Kumulativní četnost m_i** i („cumulative frequency“) – definujeme jako počet hodnot proměnné, které nabývají varianty nižší nebo rovné i-té variantě.

Uvažte např. proměnnou „známka ze statistiky“, která nabývá variant: „výborně“, „velmi dobré“, „prospěl“, „neprospěl“, pak např. kumulativní četnost pro variantu „prospěl“ bude rovna počtu studentů, kteří ze statistiky získali známku „prospěl“ nebo lepší.

Jsou-li jednotlivé varianty uspořádány podle své „velikosti“ ($x_1 < x_2 < \dots < x_k$), platí $m_i = \sum_{j=1}^i n_j$, Je tedy zřejmé, že kumulativní četnost k -té („nejvyšší“) varianty je rovna rozsahu proměnné $mk = n$.

- **Kumulativní relativní četnost F_i** („cumulative relative frequency“) – vyjadřuje jakou část souboru tvoří hodnoty nabývající i -té a nižší varianty. $F_i = \sum_{j=1}^i p_j$, což není nic jiného než relativní vyjádření kumulativní četnosti: $F_i = \frac{m_i}{n}$.

Obdobně jako pro nominální proměnné, můžeme i pro proměnné ordinální prezentovat statistické charakteristiky pomocí tabulky rozdělení četnosti. Ta obsahuje ve srovnání s tabulkou rozdělení četností pro nominální proměnnou navíc hodnoty kumulativních a kumulativních relativních četností.

TABULKA ROZDĚLENÍ ČETNOSTÍ				
Hodnoty x_i	Absolutní četnost	Relativní četnost	Kumulativní četnost	Kumulativní relativní četnost
	n_i	p_i	m_i	F_i
x_1	n_1	p_1	$m_1 = n_1$	$F_1 = p_1$
x_2	n_2	p_2	$m_2 = n_1 + n_2 = m_1 + n_2$	$F_2 = p_1 + p_2 = F_1 + p_2$
x_k	n_k	p_k	$m_k = m_{k-1} + n_k = n$	$F_k = F_{k-1} + p_k = 1$
Celkem	$\sum_{i=1}^k n_i = n$	$\sum_{i=1}^k p_i = 1$	-----	-----

- **Grafické znázornění ordinální proměnné**

- **Lorenzova křivka** ((polygon kumulativních četností, Galtonova ogiva, S křivka) – S křivka) je spojnicovým grafem, který získáme tak, že na vodorovnou osu vynášíme jednotlivé varianty proměnné v pořadí od „nejmenší“ do „největší“ a na svislou osu příslušné hodnoty kumulativních četností. Znázorněné body spojíme úsečkami.

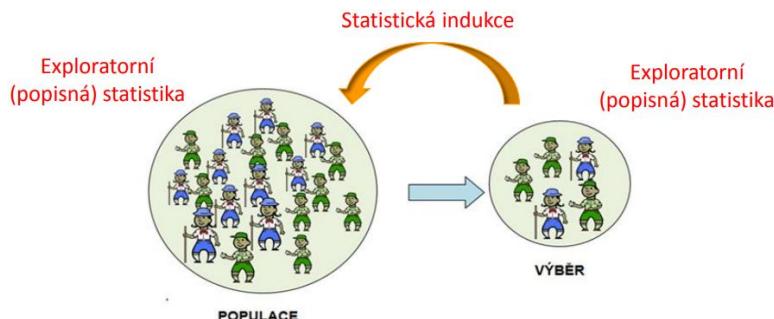


- **Paretova analýza** – lze formulovat tak, že 80% následků pramení z 20% příčin (20% lidí vlastní 80% celkového bohatství). V praxi pak bývá snahou nalézt toto malé spektrum příčin (životně důležitá menšina), které tak významně ovlivňuje výsledek.

24 Metody statistické indukce. Intervalové odhady. Princip testování hypotéz.

24.1 Statistická indukce

Statistická indukce je metoda, která dovoluje stanovit vlastnost celku (**základního souboru**) na základě pozorování jeho částí (**náhodného výběru**).



24.1.1 Základní soubor (populace)

- Je množina všech teoreticky možných objektů (např. jedinců) v uvažované situaci = statistický soubor, který je vymezen cílem výzkumu a pro který vyvzujeme závěry výzkumného šetření.
- Charakterizuje se **parametrem**, což je např. výška, váha, IQ, atp.
- Má konečný nebo nekonečný (hypotetický) **rozsah**, který je dán N (např.: N = 150 lidí, opic, rostlin,...).

24.1.2 Výběrový soubor (výběr)

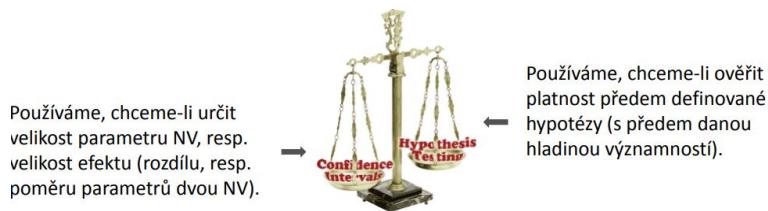
- Je část populace vybrané na základě předem stanovených kritérií resp. pravidel (podmnožina základního souboru).
- **O náhodném výběru** uvažujeme, když splňuje dvě základní vlastnosti:
 - **pravděpodobnost** zařazení do vzorku je pro všechny statistické jednotky populace **nenulová**,
 - statistické jednotky jsou do vzorku vybrané **nezávisle** jedna od druhé.
- **O reprezentativním výběru** uvažujeme, když výběrový soubor dobře odráží strukturu celého zkoumaného souboru.

24.1.3 Principy statistického usuzování

1. Statistické usuzování znamená zobecňování z výběrových statistik na parametry rozdělení.
2. Abychom mohli provést statistické usuzování, musíme mít nějakou teorii, jež popisuje náhodné chování sledovaných proměnných.
3. Existují dva typy výběrových chyb: **náhodné výběrové chyby** a **systematické chyby**. Získáním náhodného výběru zmenšujeme systematickou chybu a získáváme podklad pro odhad náhodné výběrové chyby.
4. Výběrová rozdělení statistik jsou teoretická **pravděpodobnostní rozdělení**, která popisují vztah mezi výběrovou statistikou a populací.
5. Směrodatná odchylka výběrového rozdělení statistiky (odhad parametru) se nazývá směrodatná chyba. Odhaduje náhodnou výběrovou chybu vypočítané statistiky (odhadu parametru).
6. Jak roste velikost výběru, výběrová chyba a směrodatná chyba se zmenšují.
7. Směrodatná chyba se používá k získání intervalového odhadu parametrů i k testování hypotéz o parametrech rozdělení.

24.2 Základní metody statistické indukce

- **Intervalové odhady** (confidence intervals) – umožňují odhadnout **nejistotu** v odhadu parametru náhodné veličiny.
- **Testování hypotéz** (hypothesis testing) – umožňuje posoudit, zda experimentálně získaná data nepopírají předpoklad, který jsem **před** provedením testování učinili.

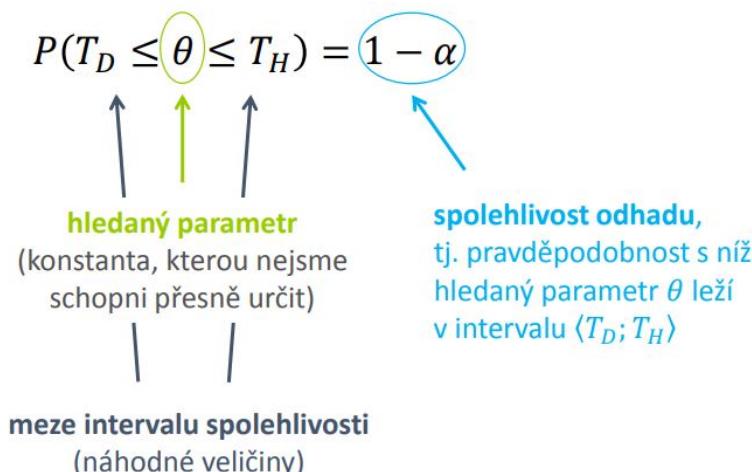


24.2.1 Intervalové odhady

- V praktických aplikacích často určujeme **odhad příslušného parametru** pomocí intervalového odhadu.
- Tento odhad je reprezentován intervalom t_D, t_H , v němž hledaný parametr leží s předem určenou pravděpodobností (spolehlivostí), kterou označujeme (1α).
- neboli parametr populace approximujeme intervalom, v němž s velkou pravděpodobností příslušný populační parametr leží.

24.2.2 Interval spolehlivosti (konfidenční interval)

Interval spolehlivosti (konfidenční interval) pro parametr θ se spolehlivostí 1α , kde $\alpha \in \langle 0; 1 \rangle$, je **taková dvojice statistik** (T_D, T_H) , že $P(T_D \leq \theta \leq T_H) = 1\alpha$.



- **Intervalový odhad** t_D, t_H je jednou z realizací intervalu spolehlivosti.
- Požadavky na interval spolehlivosti:
 - Co **největší spolehlivost** odhadu.
 - Co **nejmenší šírka** intervalu spolehlivosti. (s rostoucí spolehlivostí se zvětšuje šírka intervalového odhadu a tím **klesá významnost** takto získané informace. S rostoucím rozsahem výběru se šírka intervalového odhadu snižuje.)

Typy intervalů spolehlivosti

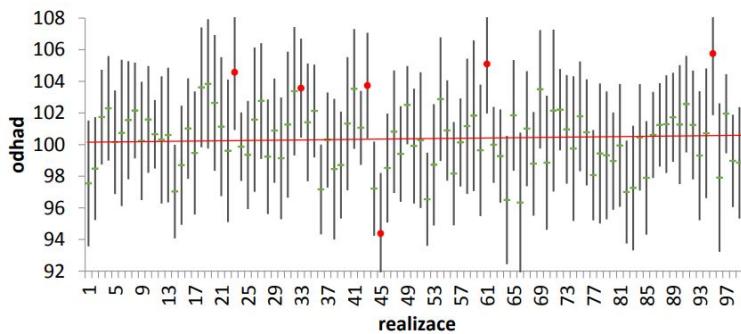
- **oboustranné**

$$P(\theta < T_D) = P(\theta > T_H) = \frac{\alpha}{2}$$

Tyto dvě podmínky zaručují, že $P(T_D \leq \theta \leq T_H) = 1 - \alpha$

- **jednostranné** (odhadujeme-li například délku života nějakého zařízení, je pro nás důležitá pouze dolní mez)

- **levostranné** $P(\theta \geq T_D^*) = 1 - \alpha$
- **pravostranné** $P(\theta \leq T_H^*) = 1 - \alpha$



Co to znamená, že spolehlivost odhadu je $1 - \alpha$? – Simulace 100 intervalových odhadů (obrázek výše) střední hodnoty (spolehlivost 0,95) získaných na základě opakovaných výběrů o rozsahu 30 z populace se střední hodnotou 100. 6 intervalů ze 100 **neobsahuje skutečnou střední hodnotu**.

24.3 Jak najít intervalový odhad parametru θ ?

Obecně:

1. Zvolíme vhodnou výběrovou charakteristiku $T(\mathbf{X})$, jejíž rozdělení známe.
- 2.

$$P(x_{\frac{\alpha}{2}} \leq T(\mathbf{X}) \leq x_{1-\frac{\alpha}{2}}) = 1 - \alpha,$$

$$P(T(\mathbf{X}) \leq x_{1-\alpha}) = 1 - \alpha,$$

$$P(T(\mathbf{X}) \geq x_\alpha) = 1 - \alpha.$$

24.4 Testování hypotéz

Statistická hypotéza – předpoklad (tvrzení) o rozdělení náhodné veličiny.

- Zdrojem statistických hypotéz jsou například předchozí zkušenosti, teorie, kterou je třeba doložit, požadavky na kvalitu produktu, dohadu založené na náhodném pozorování.
- **Příklady** statistických hypotéz:
 - Střední životnost žárovek Ed je nižší než výrobcem udávaných 5 let.
 - Mortalita je u laparoskopických operací nižší než u operací konvenčních.
 - Průměrné výsledky srovnávacích testů závisí na typu absolvované střední školy.
 - Pořízený datový soubor je výběrem z populace mající normální rozdělení.
- **Parametrická statistická hypotéza** – tvrzení ohledně efektu:
 - Hypotézy o **parametru jedné populace** (o střední hodnotě, rozptylu, mediánu, parametru binomického rozdělení,...).
 - Hypotézy o **parametrech dvou populací** (srovnávací testy).
 - Hypotézy o **parametrech více než dvou populací** (ANOVA, Kruskalův–Wallisův test,...).
- **Neparametrická statistická hypotéza** – tvrzení o **jiné vlastnosti** (rozdělení náhodné veličiny) než o jejím parametru (např. hypotézy o typu rozdělení NV, hypotézy o závislosti NV,...)

Příklad, ověření, zda statistická hypotéza je pravdivá: Domníváme se, že střední hodnota obsahu cholesterolu v krvu je u české populace 4,7 mmol/l.

$$H_0 : \mu = 4.7$$

$$H_A : \mu \neq 4.7$$

Jak tento předpoklad ověřit?

- Zjistíme údaje o obsahu cholesterolu v krvi u 100 náhodně vybraných Čechů.
- Průměrný obsah cholesterolu v krvi probandů (tj. jedinců, kteří jsou předmětem zkoumání) byl 5,4 mmol/l.

Jsou tyto výsledky v souladu s naší hypotézou?

- I kdyby byla testovaná hypotéza pravdivá, nelze očekávat, že průměrná hodnota pozorovaná ve výběru bude přesně 4,7 mmol/l.
- **Nulovou hypotézu zamítneme, pokud získané usporádání výběru bude za předpokladu platnosti nulové hypotézy velmi nepravděpodobné.**

Rozhodovací proces, v němž proti sobě stojí nulová a alternativní hypotéza:

- **Nulová hypotéza H_0** – tvrzení, že efekt je nulový, resp. že neexistuje závislost, že data mají určitý typ rozdělení, ...
- **Alternativní hypotéza H_A (H_1)** – tvrzení, popírající hypotézu nulovou (obvykle to, co chceme dokázat).

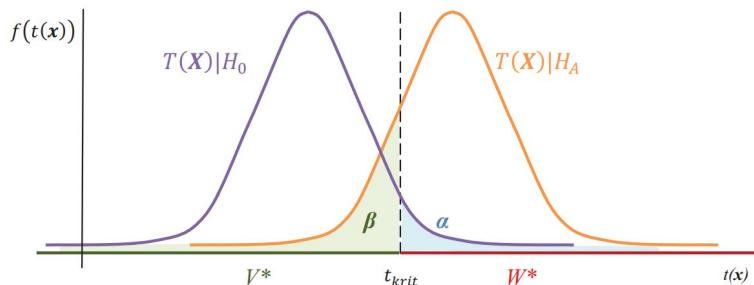
24.4.1 Klasický přístup při testování hypotéz

1. Formulujeme **nulovou a alternativní hypotézu**.
2. Zvolíme tzv. **testovací statistiku**, tj. výběrovou charakteristiku, jejíž rozdělení závisí na testovaném parametru θ . (Rozdělení testované statistiky za předpokladu platnosti nulové analýzy nazýváme **nulové rozdělení**.)
3. Ověříme **předpoklady testu!**
4. Určíme **kritický obor W^*** , tj. množinu, v níž se, za předpokladu platnosti H_0 , hodnoty testované statistiky vyskytují s velmi malou pravděpodobností.
 - Doplňkem k W^* je tzv. **obor přijetí V^*** .
 - Hranici mezi kritickým oborem a oborem přijetí označujeme jako **kritická hodnota testu t_{krit}** .
5. Na základě konkrétní realizace výběru určímě **pozorovanou hodnotu X_{OBS}** testované statistiky.
6. Na základě vztahu mezi X_{OBS} a t_{krit} rozhodneme o výsledku testu („Zamítáme H_0 .“ nebo „Nezamítáme H_0 .“)

24.4.2 Chyba I. a II. druhu

		Výsledek testu	
		Nezamítáme H_0	Zamítáme H_0
Skutečnost	Platí H_0	Správné rozhodnutí $1 - \alpha$ (spolehlivost testu)	Chyba I. druhu α (hladina významnosti)
	Platí H_A	Chyba II. druhu β	Správné rozhodnutí $1 - \beta$ (síla testu)

Jestliže nulová hypotéza je ve skutečnosti platná a my ji přesto zamítneme, dopouštíme se chyby, označované jako **chyba I. druhu**. Pravděpodobnost, že k takovému pochybení dojde, nazýváme **hladina významnosti** a označujeme ji α . Platí-li nulová hypotéza a my jsme ji nezamítli, rozhodli jsme správně. Pravděpodobnost tohoto rozhodnutí označujeme $1 - \alpha$ a nazýváme ji **spolehlivost testu**. Správným rozhodnutím je rovněž **zamítnutí nulové hypotézy v případě, že je platná hypotéza alternativní**. Tohoto rozhodnutí se dopouštíme s pravděpodobností $1 - \beta$, což bývá označováno jako síla testu. **Chybou II. druhu** je nezamítnutí nulové hypotézy v případě, že je platná hypotéza alternativní. Pravděpodobnost této chyby označujeme β .



Obrázek 24.1: Demonstrační pravděpodobností chyb I. a II. druhu

Při testování hypotéz se samozřejmě snažíme postupovat tak, abychom minimalizovali obě chyby, tj. dosáhnout vysoké síly testu (nízkého β) při co nejnižší hladině významnosti α . To však není možné, neboť snížením β se zvýší hladina významnosti α a naopak. Proto je třeba najít kompromis mezi požadavky na α a β .

24.4.3 Parametrická statistická hypotéza

Jednovýběrové testy

- Test o **střední hodnotě** (z-test, t-test).
- Test o **rozptylu**.
- Test o **parametru binomického rozdělení**.
- Test o **mediánu** (Wilcoxonův test, Mediánový test).

Dvouvýběrové testy

- Test o shodě **dvou středních hodnot** (t-test, Aspinové–Welchův test).
- Test o shodě **rozptylů** (F-test).
- Test o shodě **parametrů dvou binomických rozdělení** (test homogeneity dvou binomických rozdělení).
- Test o shodě **mediánů** (Mannův–Whitneyův test).
- **Párové testy** (párový t-test, párový znaménkový test).

Vícevýběrové testy

- Testy **shody rozptylů** (Bartletův test, Hartleyův test, Cochranův test, Leveneův test).
- **Analýza rozptylu** (tzv. ANOVA, tj. shody středních hodnot) – post hoc analýza pro analýzu rozptylu.
- Kruskal–Wallisův test (test **shody mediánů**) – post hoc analýza Kruskal–Wallisův test.

ANOVA

- Test umožňující **srovnání průměrů více než dvou výběrových souborů**.
- Můžeme například zkoumat, zda:
 - typ absolvované střední školy ovlivňuje počet bodů dosažených studenty u přijímací zkoušky z matematiky,
 - použitá medikace ovlivňuje krevní tlak pacientů,
 - typ použitého hnojiva ovlivňuje výnosy určité plodiny,
 - pracovní výkon dělníka závisí na umístění stroje, apod.

Část III

Softwarové inženýrství

25 Softwarový proces. Jeho definice, modely a úrovně vyspělosti.

Softwarové inženýrství je inženýrská disciplína zabývající se praktickými problémy vývoje rozsáhlých softwarových systémů.

25.1 Softwarový proces

Softwarový proces je po částech uspořádaná množina kroků směřujících k vytvoření nebo úpravě softwarového díla. Cílem je spojit požadavky a vyjadřování uživatelů s postupy, algoritmy a technikami používaných při programování.

- Krokem může být **aktivita** nebo opět **podproces** (hierarchická dekompozice procesu).
- Aktivity a podprocesy mohou **probíhat v čase souběžně**, tudíž je vyžadována jejich koordinace.
- Je **nutné zajistit opakovatelnost použití procesu ve vztahu k jednotlivým softwarovým projektům**, tedy zajistit jeho **znovupoužitelnost**. Cílem je dosáhnout stabilních výsledků vysoké úrovně kvality.
- Řada činností je zajišťována lidmi vybavenými určitými schopnostmi a znalostmi a majícím k dispozici technické prostředky nutné pro realizaci těchto činností.
- **Softwarový produkt** je realizován v kontextu organizace s danými ekonomickými možnostmi a organizační strukturou.

25.2 Modely softwarového procesu

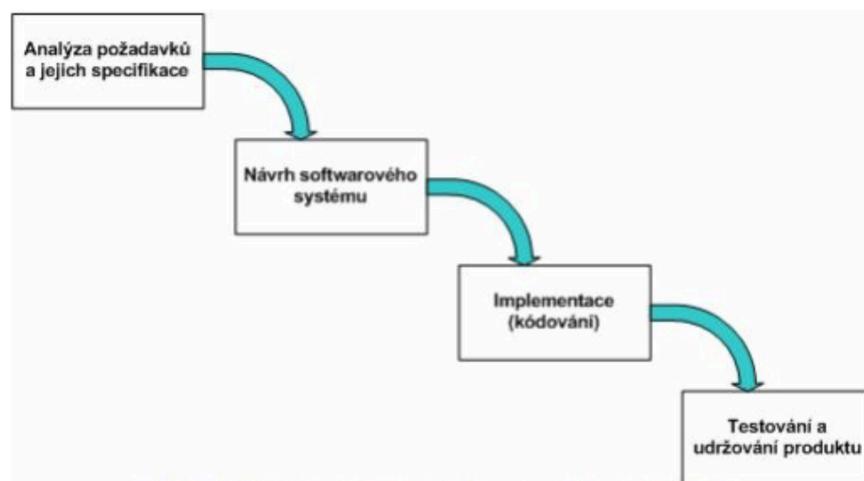
Vzhledem k tomu, že vývoj softwaru je relativně nová problematika dodnes není jasně definováno jak by měl správný softwarový proces vypadat. Byla však vyvinuta řada různých přístupu k vývoji softwaru. Lze říct, že základem všech je vodopádový model, který lze nalézt ve většině používaných přístupů.

25.2.1 Vodopádový model

Vychází z **rozdělení životního cyklu softwarového díla** na **čtyři** základní fáze:

1. **analýza požadavků** a jejich **specifikace**,
2. **návrh** softwarového systému,
3. **implementace**,
4. **testování** a udržování vytvořeného produktu.

Princip vodopádu spočívá v tom, že **následující množina činností spjatá s danou fází** nemůže započít dříve, než skončí předchozí. Jinými slovy, výsledky předchozí fáze „vtékají“ jako vstupy do fáze následující.



Obr. 2.1: Vodopádový model softwarového procesu

Model je možno v **různých modifikacích a rozšířeních** nalézt ve většině současných přístupů. Tyto modifikace vznikly především kvůli odstranění některých jeho **nedostatků**, mezi které patří:

- **Prodleva** mezi zadáním projektu a vytvořením spustitelného systému je příliš dlouhá.
- **Výsledek závisí** na **úplném a korektním zadání požadavků** kladených na výsledný produkt.
- **Nelze odhalit výslednou kvalitu produktu** danou splněním všech požadavků, dokud není výsledný softwarový systém hotov.

25.2.2 Modifikace Vodopádového modelu

- **Inkrementální – postupné vytváření verzí** softwaru zahrnující postupně širší spektrum funkcí definovaných v průběhu jeho vytváření. V podstatě se jedná o **více menších vodopádů provedených za sebou** tak, aby každý z nich odpovídal nové sadě doplněných požadavků.
- **Spirálový** – zahrnuje do svého **životního cyklu další fáze** jako je vytvoření a hodnocení **prototypu** ověřující funkcionality cílového systému, přičemž **každý cyklus nabízí další požadavky** specifikované zadavatelem.
- **V-mode** – Podobně jako vodopádový, v každém kroku ale probíhá testování (Obrázek na wiki, jinak je to zbytečné: [https://en.wikipedia.org/wiki/V-Model_\(software_development\)](https://en.wikipedia.org/wiki/V-Model_(software_development)))

25.3 RUP (Rational Unified Process)

Rational Unified Process (RUP) je **metodika vývoje softwaru** vytvořená společností Rational Software Corporation. Je použitelná pro **jakýkoliv rozsah projektu**, ale díky vysoké rozsáhlosti RUP je vhodné přizpůsobit metodiku specifickým potřebám. RUP je vhodnější spíš pro **rozsáhlejší projekty a větší vývojové týmy**, neboť klade důraz na **analýzu a design, plánování, řízení zdrojů a dokumentaci**. Hlavní znaky:

- softwarový produkt je vyvíjen **iteračním způsobem**,
- jsou **spravovány požadavky** na něj kladené,
- využívá se **již existujících softwarových komponent**,
- na konci každé **iterace** je spustitelný kód,
- model softwarového systému je **vizualizován (UML)**,
- průběžně je **ověřována kvalita** produktu,
- **změny** systému **jsou řízeny** (každá změna je přijatelná a změny jsou sledovatelné).

Zkrácená definice

– Každý cyklus vede k vytvoření takové verze systému, kterou lze předat uživatelům a implementuje jimi specifikované požadavky. RUP cyklus má 4 **fáze**: Zahájení, Rozpracování, Tvorba, Předání. **Iterace** je úplná vývojová smyčka vedoucí k vytvoření spustitelné verze systému reprezentující podmnožinu vyvíjeného cílového produktu a která je postupně rozšiřována každou iterací až do výsledné podoby.

25.3.1 Vývoj software iteračním způsobem

V současné době, kdy se předmětem vývoje staly softwarové systémy vysoké úrovně, je **nemožné nejprve specifikovat celé zadání**, následně navrhnout jeho řešení, vytvořit softwarový produkt implementující toto zadání, vše otestovat a předat zadavateli k užívání. Jediné možné řešení takového problému je přístup postavený na **iteracích**, umožňující **postupně upřesňovat cílový produkt** cestou jeho **inkrementálního rozšiřování** z původní hrubé formy do výsledné podoby. Softwarový systém je tak **vyvíjen ve verzích**, které lze průběžně ověřovat se zadavatelem a případně jej pozměnit pro následující iteraci. **Každá iterace končí vytvořením spustitelného kódu**.

25.3.2 Správa požadavků

Kvalita výsledného produktu je dána mírou uspokojení požadavků zadavatele. Právě otázka korektní specifikace všech požadavků bývá problémem všech softwarových systémů. Velmi často výsledek i mnohaletého úsilí týmu softwarových inženýrů propadne díky neodstatečné specifikaci zadání. Proces RUP popisuje jak požadavky na softwarový systém doslova vykládat od zadavatelů, jak je organizovat a následně i dokumentovat. Monitorování změn v požadavcích se stává nedílnou součástí vývoje stejně jako správně dokumentované požadavky sloužící ke komunikaci mezi zadavateli a týmem řešitelů.

25.3.3 Vývoj pomocí komponent

Proces RUP poskytuje systematický přístup k definování architektury využívající nové či již existující komponenty. Tyto komponenty jsou vzájemně propojovány v rámci korektně definované architektury bud pro případ od případu (ad-hoc), nebo prostřednictvím komponentní architektury využívající Internet, infrastrukturu CORBA (Common Object Request Broker Architecture), nebo COM (Component Object Model). Již dnes existuje celá řada znovupoužitelných komponent a je zřejmé, že softwarový průmysl věnuje velké úsilí dalšímu vývoji v této oblasti. Vývoj software se tak přesouvá do oblasti skládání produktu z prefabrikovaných komponent.

25.3.4 Vizualizace modelování SW systému

Softwarový proces RUP popisuje jak vizualizovat model softwarového systému s cílem uchopit strukturu a chování výsledné architektury produktu a jeho komponent. Smyslem této vizualizace je skrýt detaily a vytvořit kód užitím jazyka postaveného na grafickém vyjádření stavebních bloků výsledného produktu. Základem pro úspěšné použití principů vizualizace je za průmyslový standard považován jazyk **UML** primárně určený pro účely modelování softwarových systémů.

25.3.5 Ověřování kvality softwarového produktu

Princip ověřování kvality vytvářeného produktu je součástí softwarového procesu, je obsažen ve všech jeho aktivitách, **týká se všech účastníků vývoje** softwarového řešení. Využívají se objektivní měření a kriteria (metriky) kvantifikující kvalitu výsledného produktu. Zajištění kvality tak není považováno za něco co stojí mimo hlavní linii vývoje produktu a není to záležitost zvláštní aktivity realizované speciální skupinou.

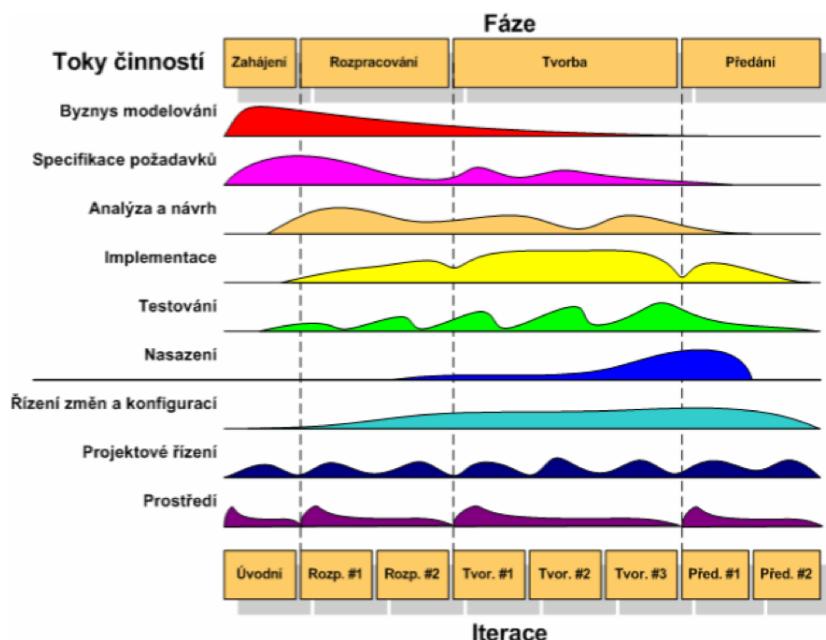
25.3.6 Řízení změn

Řízení změn umožňuje zaručit, že každá změna je přijatelná a všechny změny systému jsou sledovatelné. Důvod, proč je kladen na tuto problematiku takový důraz spočívá v tom, že prostředí ve kterém je softwarový systém vyvíjen podléhá častým a mnohdy i radikálním změnám, které je nutno integrovat do vlastního řešení.

Proces RUP popisuje jak **řídit, sledovat a monitorovat změny** a umožňuje tak úspěšný iterativní vývoj. Nedílnou součástí této problematiky je vytvoření bezpečného pracovního prostředí poskytující maximální možnou ochranu před změnami vznikajících v jiném pracovním prostředí.

25.3.7 Schématické vyjádření RUP

Vlastní softwarový proces má svou statickou strukturu ve smyslu toho, z jakých se skládá **toků činností** a **aktivit**. Na druhou stranu má také svou dynamickou stránku popisující jak je softwarový produkt vyvíjen v čase (níže).



V čem se tento přístup liší o dříve zmíněného vodopádového modelu? Základní rozdíl spočívá v tom, že **toky činností probíhají souběžně**, i když z obrázku jasně vyplývá, že objem prací se liší podle fáze rozpracování softwarového systému. Zřejmě, těžiště činností spjatých s byznys modelováním a specifikací požadavků bude

v úvodních fázích, zatímco problematika rozmístění softwarových balíků na počítačích propojených sítí (počítačové infrastruktury) bude záležitostí fází závěrečných. Celý životní cyklus je pak rozložen do čtyř základních fází (zahájení, rozpracování, tvorba a předání), přičemž pro každou z nich je typická realizace několika iterací umožňující postupné detailnější rozpracování produktu.

25.3.8 Cykly, fáze, iterace

Každý cyklus vede k vytvoření takové verze systému, kterou lze předat uživatelům a implementuje jimi specifikované požadavky. Jak již bylo uvedeno v předchozí kapitole, každý takový vývojový cyklus lze rozdělit do čtyř po sobě jdoucích fází:

1. **Zahájení**, kde je původní myšlenka rozpracována do **vize koncového produktu** a je definován rámec toho, jak celý systém bude vyvíjen a implementován.
2. **Rozpracování** je fáze věnovaná **podrobné specifikaci požadavků a rozpracování architektury** výsledného produktu.
3. **Tvorba** je zaměřena na **kompletní vyhotovení požadovaného díla**. Výsledné programové vybavení je vytvořeno kolem navržené kostry (architektury) softwarového systému.
4. **Předání** je závěrečnou fází, kdy **vytvořený produkt je předán do užívání**. Tato fáze zahrnuje i další aktivity jako je beta **testování, zaškolení** apod.

Každá fáze může být dále rozložena do několika iterací.

25.3.9 Iterace

Iterace je **úplná vývojová smyčka vedoucí k vytvoření spustitelné verze kódu** reprezentující **podmnožinu** vyvíjeného cílového produktu, a která je **postupně rozšiřována každou iterací** až do výsledné podoby.



25.3.10 Statická struktura procesu

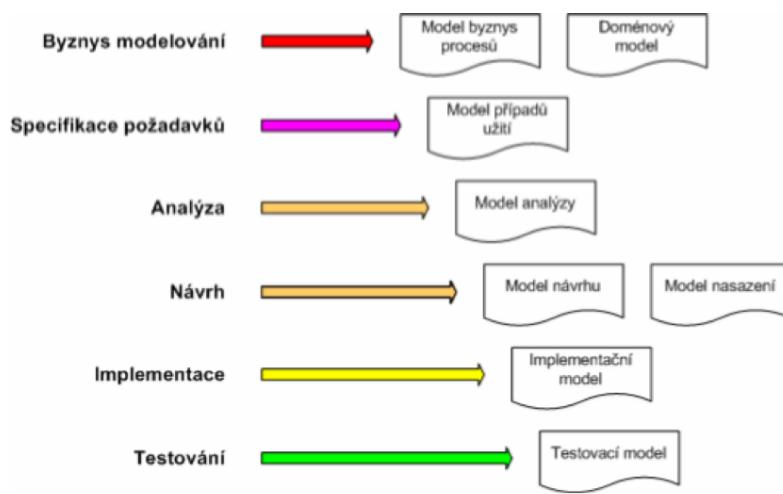
Definuje **KDO** (role), **CO** (artefaky), **JAK** (aktivity a jejich toky) a **KDY** to má vytvořit. Smyslem každého procesu a tedy i softwarového je specifikovat kdo v něm vystupuje, co má vytvořit, jak to má vytvořit a kdy to má vytvořit. Z tohoto pohledu hovoříme o následujících elementech tvořících strukturu každého softwarového procesu:

- **Role (pracovníci)** definují chování, kompetence a zodpovědnosti jednotlivých osob (analytik, programátor, projektový manažer apod.) nebo skupiny osob spolupracujících v týmech. Jednotlivé osoby (lidské zdroje) jsou mapovány na požadované role dle toho, jak jsou požadované kompetence slučitelné se schopnostmi těchto osob.
- **Artefakty** reprezentují entity, které jsou v procesu vytvářeny, modifikovány nebo zužitkovány (modely, dokumentace, zdrojové kódy apod.). Základním artefaktem, který je nosným pro vývoj softwarového systému je **model - zjednodušení reality umožňující lépe pochopit vyvíjený systém**.
- **Aktivity (činnosti)** prováděné pracovníky s cílem vytvořit nebo upravit artefakty (kompilace zdrojových kódů, vytvoření návrhu apod.).
- **Toky činností (workflow)** reprezentují posloupnosti aktivit vytvářející požadované produkty (byznes modelování, specifikace požadavků apod.).

25.3.11 Základní a podpůrné toky činností

Vývoj softwarového systému je dán celou řadou aktivit a činností, které jsou uspořádány do toků charakteristických svým účelem. Z tohoto pohledu můžeme hovořit o tzv. základních tocích, jejichž výsledkem je část softwarového produktu (artefakt) a o tocích podpůrných, které nevytváří hodnotu, ale jsou nutné pro realizaci základních toků. Základní toky vytvářející vlastní softwarový produkt jsou následující:

- **Byznys modelování** popisuje strukturu a dynamiku podniku či organizace.
- **Specifikace požadavků** definuje prostřednictvím specifikace případů použití softwarového systému jeho funkcionalitu.
- **Analýza a návrh** zaměřené na specifikaci architektury softwarového produktu.
- **Implementace** reprezentuje vlastní tvorbu softwaru, testování komponent a jejich integraci.
- **Testování** zaměřené na činnosti spjaté s ověřením správnosti řešení softwaru v celé jeho složitosti.
- **Rozmístění** se zabývá problematikou **konfigurace** výsledného produktu na cílové počítačové infrastrukturu.



Obr. 2.5: Toky činností a modely jimi vytvářené

Podpůrné toky, samozřejmě ty nejdůležitější a spjaté s vlastním vývojem, jsou:

- **Řízení změn a konfigurací** se zabývá problematikou správy jednotlivých verzí vytvářených artefaktů odrážejících vývoj změn požadavků kladených na softwarový systém.
- **Projektové řízení** zahrnující problematiku koordinace pracovníků, zajištění a dodržení rozpočtu, aktivity plánování a kontroly dosažených výsledků. Nedílnou součástí je tzv. **řízení rizik**, tedy identifikace problematických situací a jejich řešení.
- **Prostředí** a jeho správa je tok činností poskytuje vývojové organizaci metodiku, nástroje a infrastrukturu podporující vývojový tým.

25.4 Úrovně vyspělosti

Úroveň definice a využití softwarového procesu je hodnocena dle stupnice **SEI (Software Engineering Institute) 1 - 5** vyjadřující vyspělost firmy či organizace z daného hlediska. Tento model hodnocení vyspělosti a schopností dodavatele softwarového produktu se nazývá **CMM (Capability Maturity Model)** a jeho jednotlivé úrovně lze stručně charakterizovat asi takto:

1. **Počáteční (Initial)** – firma **nemá definován softwarový proces** a každý projekt je řešen **případ od případu** (ad hoc).
2. **Opakovatelná (Repeatable)** – firma identifikovala v jednotlivých projektech **opakovatelné postupy** a tyto je schopna reprodukovat v každém novém projektu.
3. **Definovaná (Defined)** – softwarový proces je **definován (a dokumentován)** na základě integrace dříve identifikovaných opakovatelných kroků.

4. **Řízená (Managed)** – na základě definovaného softwarového procesu je firma schopna jeho **řízení a monitorování**.
5. **Optimalizovaná (Optimized)** – zpětnovazební informace získaná **dlouhodobým procesem monitorování** softwarového procesu je využita ve prospěch jeho optimalizace.

26 Vymezení fáze „sběr a analýza požadavků“. Diagramy UML využité v dané fázi.

26.1 Specifikace požadavků

Specifikace a analýza požadavků je první fáze vývoje softwaru. **Cílem je definovat požadavky na software a popsat jeho funkcionalitu.** Výsledkem této fáze by měly být dokumenty, které se stanou součástí smlouvy mezi zadavatelem a vývojovým týmem. **Kvalita** výsledného produktu je pak dána **mírou uspokojení požadavků zadavatele.**

V rámci specifikace požadavků je třeba analyzovat procesy u zadavatele, které bude software řešit, nebo s ním nějak jinak souvisí. K popisu těchto procesů dobře poslouží **Diagram případu užití**, **Sekvenční diagram** a **Diagram aktivit**.

26.1.1 Cíle požadavků

- chceme vytvořit a udržovat dohody se zákazníky a dalšími zainteresovanými stranami o tom, co by **systém měl dělat a proč**,
- aby vývojáři lépe pochopili **požadavky na systém**,
- definování **hranic systému**,
- vytvořit základ pro plánování **technického obsahu** interakcí,
- poskytnout základ pro **odhad nákladů a času na vytvoření systému**,
- definovat **uživatelské rozhraní** pro systém, se zaměřením na potřeby uživatelů.

26.1.2 Aktivity spojené s analýzou požadavků

- **Sběr požadavků** – komunikace se zákazníky a uživateli za účelem získání jejich požadavků na systém.
- **Analýza požadavků** – identifikování nejasných požadavků, nekompletních, nebo protichůdných a následné řešení těchto nesrovnalostí.
- **Zaznamenání požadavků** – dokumentování požadavků v různých formách, jako běžný textový dokument, případy užití (use case), nebo specifikace procesů.

Požadavky je také nutno: **organizovat, dokumentovat, prioritizovat, filtrovat a sledovat**.

26.1.3 Typy požadavků – FURPS

- **Funkční požadavky (chování)** – se používají k vyjádření chování systému zadáním jak vstupních tak i výstupních podmínek.
- **Doplňující požadavky (nefunkční)**
 - **Použitelnost (Usability)** – se zabývá lidskými faktory, jako je vzhled, snadné používání, atd.
 - **Spolehlivost (Reliability)** – četnost a závažnost selhání, obnovitelnost a přesnost.
 - **Výkon (Performance)** – se zabývá množstvím transakcí, jako je rychlosť, doba odezvy, atd.
 - **Podporovatelnost (Supportability)** – řeší, jak těžké je udržet systém a další vlastnosti potřebné k udržení systému po jeho vydání.

26.2 Diagram případů užití

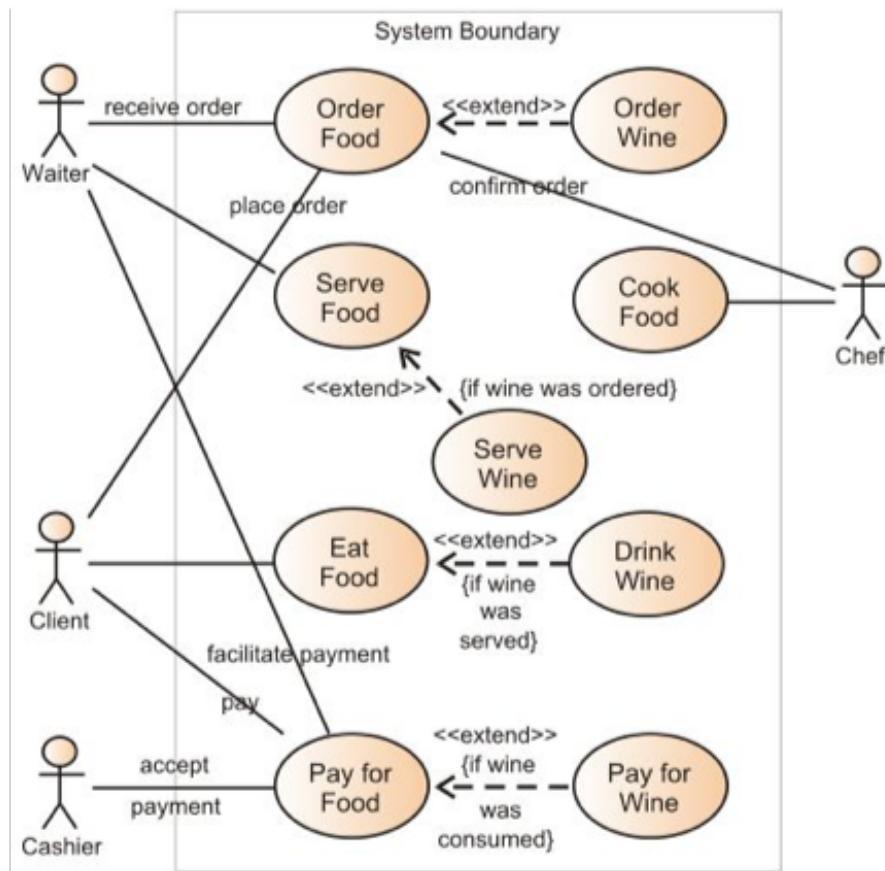
Případ užití (use case) – je technika pro zdokumentování případného požadavku na nový systém, nebo změny na stávající systém. Každý use case poskytuje jeden nebo více scénářů, které zaznamenávají, jak by systém měl spolupracovat s koncovým uživatelem nebo jiným systémem.

Diagram případu užití popisuje **vztahy** mezi **aktéry** a jednotlivými **případy užití**. Součástí diagramu jsou:

- **Aktéři** – definují **uživatele či jiné systémy**, kteří budou vstupovat do interakce s vyvíjeným softwarovým systémem.
- **Relace (vztahy)** – **vazby** a **vztahy** mezi aktéry a případy užití. V diagramu jsou znázorněny **šipkami případně čárami**.

- **Případy užití** – specifikující vzory chování softwarovým systémem. Každý případ užití lze chápat jako posloupnost vzájemně navazujících transakcí vykonalých v dialogu mezi aktérem a vlastním softwarovým systémem.
 - **Scénář** – unikátní sekvence akcí skrz use-case (jedna cesta/instance provedení).

Účelem diagramu případů užití je definovat co existuje vně vyvíjeného systému (aktéři) a co má být systémem prováděno (případy užití). Vstupem pro sestavení diagramu případů užití je **byznys model**, konkrétně modely podnikových procesů. **Výsledkem** analýzy těchto procesů je **seznam požadovaných funkcí softwarového systému**, které podporí nebo dokonce nahradí některé z uvedených aktivity jejich softwarovou implementací.



Pro složitější a obsáhlější diagramy případů užití se zavadí **tři (+1)** typy vztahů:

- **Relace** - bez typu (jen čára) značí přístup k UC, nejčastěji mezi aktéri a UC
 - **Include** – případ užití musí **obsahovat** jiný (UC s include se vždy provede před samotným UC).
 - **Extend** – případ užití může **rozšiřovat** jiný (UC s extend se může nebo nemusí provést).
 - **Generalizace** – případ užití může být **speciálním případem** jiného (dědičnost).

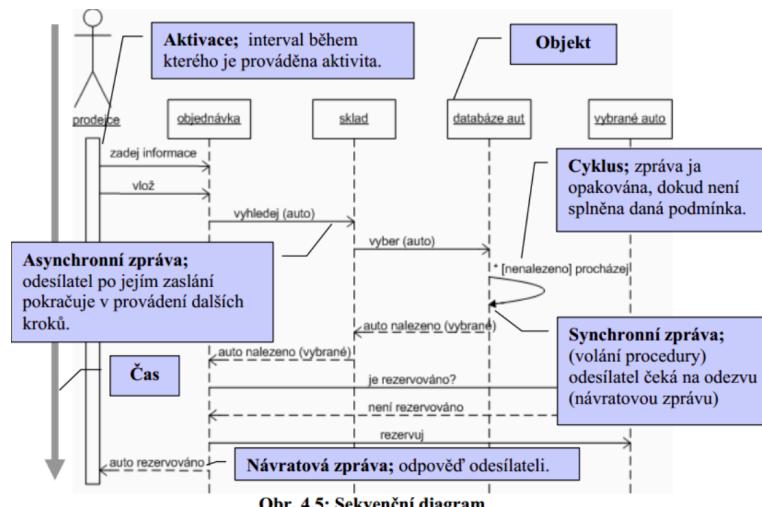
26.2.1 Doporučená forma vytváření use-case

- má vyjadřovat **co systém dělá** (ne jak) a co od něj očekávají aktéři,
 - měly by být používány jen pojmy problémové domény – žádné neznámé termíny,
 - případy užití by měly být co **nejjednodušší**, ať jim rozumí i zadavatelé – nepoužívat příliš `<include>` a `<extend>`,
 - **nepoužívat** příliš funkční dekompozici (**specializaci**),
 - primární aktéři umístění vlevo a pojmenování krátkým podstatným jménem,
 - každý aktér by měl být **propojen s minimálně jedním use-case**,
 - základní use case vlevo a další kreslit směrem doprava, rozšířující use-case směrem dolů.

26.3 Sekvenční diagram

Sekvenční diagram popisuje funkce systému z pohledu **objektů** a **jejich interakcí**. Komunikace mezi objekty je **znázorněná v čase**, takže z diagramu můžeme vyčíst i životní cyklus objektu.

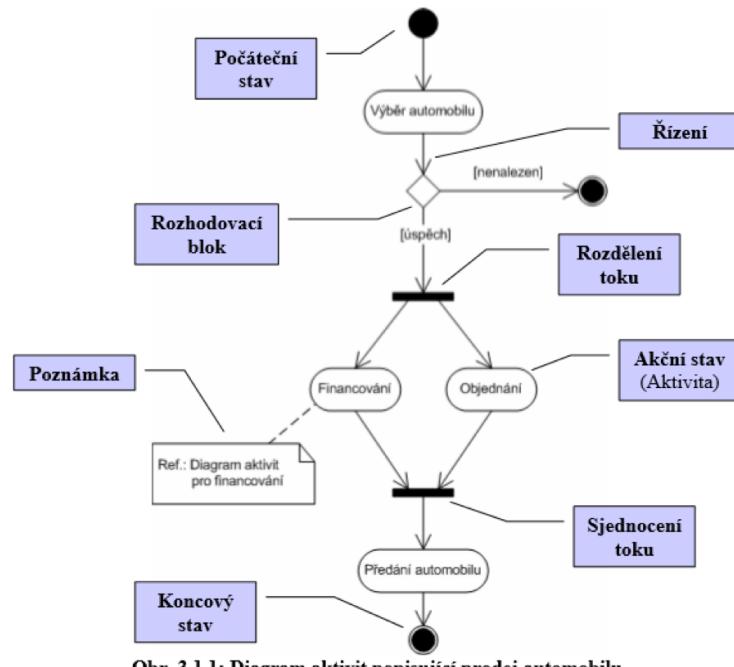
Tento diagram popisuje jaké **zprávy (požadavky)** jsou mezi objekty zasílány z pohledu času. Diagram je tvořen **objekty uspořádanými do sloupců** a šipky mezi nimi odpovídají vzájemně si zasílaným zprávám. Zprávy mohou být synchronní nebo asynchronní. V případě **synchronních zpráv odesílatel čeká na odpověď** adresáta, v případě **asynchronní zprávy odesílatel nečeká** na odpověď a pokračuje ve vykonávání své činnosti. Souvislé provádění nějaké činnosti se v sekvenčním diagramu vyjadřuje svisle orientovaným obdélníkem. Odezvu adresáta lze opět modelovat, v tomto případě tzv. **návratovou zprávou (přerušovaná čára)**. Tok času probíhá ve směru od shora dolů.



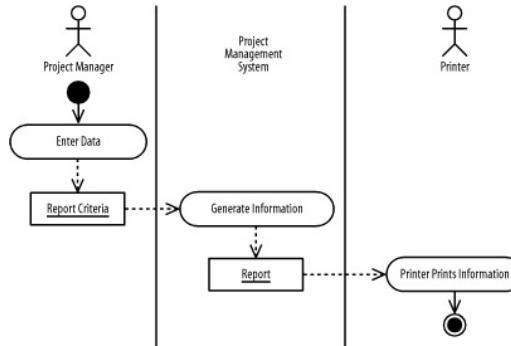
Obr. 4.5: Sekvenční diagram

26.4 Diagram aktivit

Diagram aktivit popisuje jednotlivé procesy pomocí aktivit reprezentující jeho (akční) **stavy** a **přechody mezi nimi**. Pokud aktivity „přetékají“ v jednotlivých stavech mezi uživateli, mohou být tyto stavy naznačeny pomocí tzv **swimlines**.



Obr. 3.1.1: Diagram aktivit popisující prodej automobilu



27 Vymezení fáze „Návrh“. Diagramy UML využité v dané fázi. Návrhové vzory – členění, popis a příklady.

Model návrhu dále **upřesňuje model analýzy** ve světle **skutečného implementačního prostředí**. Model návrhu tak představuje **druhou fázi** vývoje softwaru. Jde o abstrakci zdrojového kódu, která bude sloužit jako hlavní dokument programátorům v další implementační fázi.

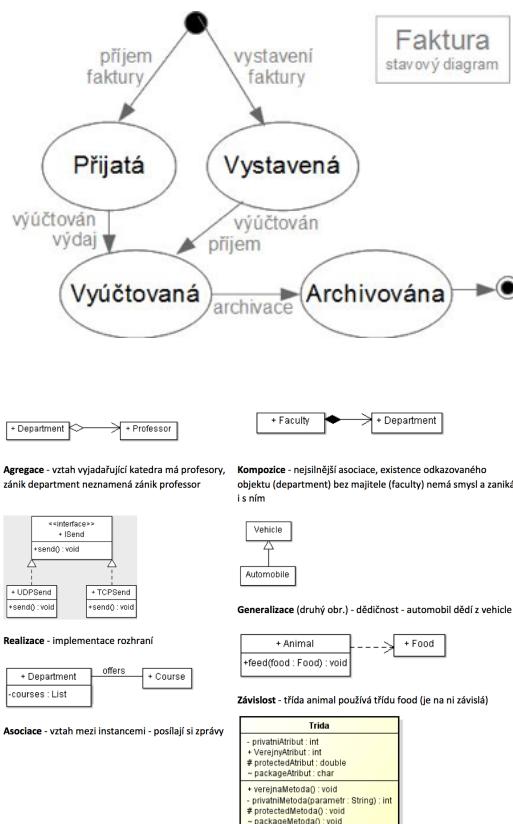
27.1 Návrh a jeho cíle

Pojem implementační prostředí v podstatě vyjadřuje **možnost namapovat navržené softwarové komponenty** obsažené v modelu analýzy na architekturu systému určeného k provozu vyvýšené aplikace s **maximálním možným využitím služeb již existujících softwarových komponent**. Postup včlenění implementačního prostředí do vyvýšené aplikace je dán následující posloupností činností:

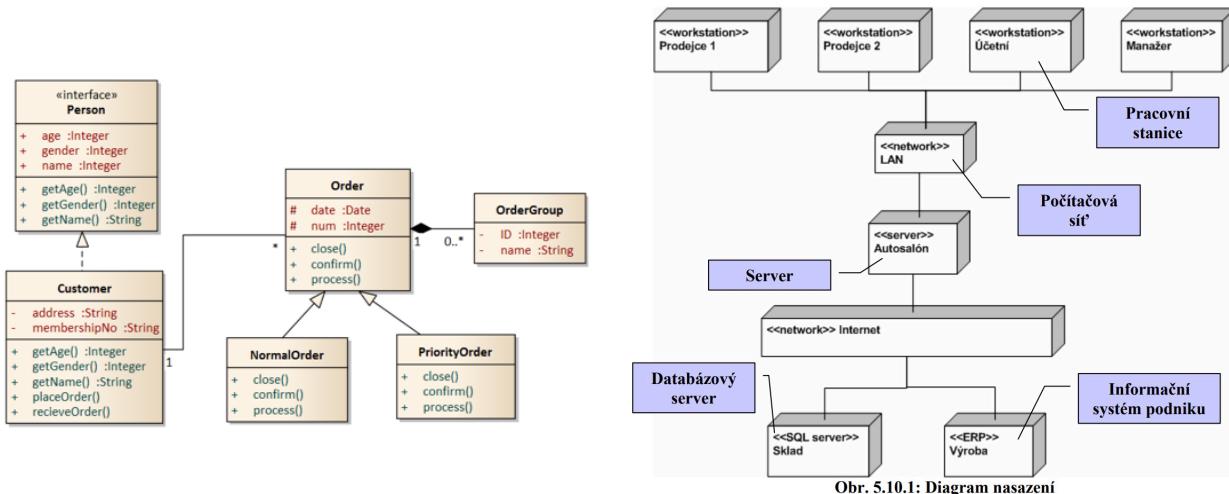
1. Definice **systémové architektury**.
2. Identifikace **návrhových vzorů** a možnosti znovupoužití tzv. rámcových řešení.
3. Definice softwarových komponent a jejich **znovupoužití**.

V této fázi se uplatňují následující diagramy:

- **Diagram tříd** – specifikuje **množinu tříd, rozhraní a jejich vzájemné vztahy**. Tyto diagramy slouží k vyjádření **statického** pohledu na systém. Vazby mezi objekty jsou vysvětleny na obrázku níže.
- **Sekvenční diagram** – popisuje **interakce mezi objekty** z hlediska jejich **časového uspořádání**.
- **Diagram spolupráce** – je obdobně jako sekvenční diagram zaměřen na **interakce**, ale z pohledu strukturální organizace objektů. Jinými slovy není primárním aspektem časová posloupnost posílaných zpráv, ale **topologie rozmístění objektů**.
- **Stavový diagram** – dokumentuje **životní cyklus objektu** dané třídy a stavů, ve kterých se může nacházet.
- **Diagram nasazení** – popisuje **konfiguraci** (topologii) **technických prostředků** umožňující běh vlastního softwarového systému (rozmístění HW a SW).



Obrázek 27.1: Vysvětlení relací v diagramu tříd



Obr. 5.10.1: Diagram nasazení

27.2 Návrhové vzory – členění

Návrhové vzory jsou metodiky (šablony) pro řešení různých problémů, se kterými se vývojář může setkat. Objektově orientované návrhové vzory typicky ukazují **vztahy** a **interakce mezi třídami a objekty**, aniž by určovaly implementaci konkrétní třídy. Dělí se do tří základních skupin:

- **Creational Patterns (vytvářející)** – určené k řešení problému vytváření instancí tříd cestou delegace této funkce na speciálně k tomuto účelu navržené třídy. Většinou se jedná o dynamická rozhodnutí učiněná za běhu programu.
- **Structural Patterns (strukturální)** – představují skupinu návrhových vzorů zaměřujících se na možnosti uspořádání jednotlivých objektů a jejich tříd nebo komponent v systému. Snahou je zpřehlednit systém a využít možností strukturalizace kódů.
- **Behavioral Patterns (chování)** – zajímají se o chování systému. Mohou být založeny na třídách nebo objektech. U tříd využívají při návrhu řešení především principu dědičnosti. V druhém přístupu je řešena **spolupráce mezi objekty a skupinami objektů**, která zajišťuje dosažení požadovaného výsledku.

27.2.1 Creational patterns (Vzory týkající se tvorby objektů)

- **Abstract Factory (Abstraktní továrna)** – Definuje rozhraní pro vytváření rodin objektů, které jsou na sobě závislé nebo spolu nějak souvisí bez určení konkrétní třídy. Klient je odstíněn od vytváření konkrétních instancí objektů.
- **Factory Method (Tovární metoda)** – Definuje rozhraní pro vytváření objektu, které nechává potomky rozhodnout o tom, jaký objekt bude fakticky vytvořen. *Tovární metoda nechává třídy přenést vytváření na potomky.
- **Builder (Stavitel)** – Odděluje tvorbu komplexu objektů od jejich reprezentace tak, aby stejný proces tvorby mohl být použit i pro jiné reprezentace.
- **Singleton (Jedináček)** – Zajišťuje, že daná třída má pouze jednu instanci (využívá se privátních konstruktorů a samotná třída ve statické proměnné uchovává danou instanci).
- **Prototype (Prototyp, Klon)** – Specifikuje druh objektů, které se mají vytvořit použitím prototypového objektu. Nové objekty se vytváří kopírováním tohoto prototypového objektu.

27.2.2 Structural Patterns (Vzory týkající se struktury programu)

- **Adapter (Adaptér)** – Potřebujete-li, aby spolu pracovaly dvě třídy, které nemají kompatibilní rozhraní. Adaptér převádí rozhraní jedné třídy na rozhraní druhé třídy.
- **Bridge (Most)** – Oddělí abstrakci od implementace, tak aby se tyto dvě mohly libovolně lišit.
- **Composite (Kompozit, Strom, Složenina)** – Komponuje objekty do stromové struktury a umožňuje klientovi pracovat s jednotlivými i se složenými objekty stejným způsobem.
- **Decorator (Dekorátor)** – Dekorátor se vytváří za účelem změny instancí tříd bez nutnosti vytvoření nových odvozených tříd, jelikož pouze dynamicky připojuje další funkčnosti k objektu. Nový objekt si zahovává původní rozhraní.

- **Facade (Fasáda)** – Nabízí jednotné rozhraní k sadě rozhraní v podsystému. Definuje rozhraní vyšší úrovně, které zjednodušuje použití podsystému.
- **Flyweight (Muší váha)** – Je vhodná pro použití v případě, že máte příliš mnoho malých objektů, které jsou si velmi podobné.
- **Proxy** – Nabízí náhradu nebo zástupný objekt za nějaký jiný pro kontrolu přístupu k danému objektu.

27.2.3 Behavioral Patterns (Vzory týkající se chování)

- **Observer (Pozorovatel)** – V případě, kdy je na jednom objektu závislých mnoho dalších objektů, poskytne vám tento vzor způsob, jak všem dát vědět, když se něco změní. Observer je možné použít v situaci, kdy je definována závislost jednoho objektu na druhém. Závislost ve smyslu propagace změny nezávislého objektu závislým objektům (pozorovatelům). Nezávislý objekt musí informovat závislé objekty o **událostech**, které je mohou ovlivnit.
- **Command (Příkaz)** – Zapouzdřete požadavek jako objekt a tím umožněte parametrizovat klienty s různými požadavky, frontami nebo požadavky na log a podporujte operace, které jdou vzít zpět.
- **Interpreter (Interpret)** – Vytváří jazyk, což znamená definování gramatických pravidel a určení způsobu, jak vzniklý jazyk interpretovat.
- **State (Stav)** – Umožňuje objektu měnit své chování, pokud se změní jeho vnitřní stav. Objekt se tváří, jako kdyby se stal instancí jiné třídy.
- **Strategy (Strategie)** – Zapouzdřuje nějaký druh algoritmů nebo objektů, které se mají měnit, tak aby byly pro klienta zaměnitelné.
- **Chain of responsibility (Zřetězení zodpovědnosti)** – Řeší jak zaslat požadavek bez přesného vymezení objektu, který jej zpracuje.
- **Visitor (Návštěvník)** – Reprezentuje operaci, která by měla být provedena na elementech objektové struktury. Visitor vám umožní definovat nové operace beze změny tříd elementů na kterých pracuje.
- **Iterator (Iterátor)** – Řeší problém, jak se pohybovat mezi prvky, které jsou sekvenčně uspořádány, bez znalosti implementace jednotlivých prvků posloupnosti.
- **Mediator (Prostředník)** – Umožňuje zajistit komunikaci mezi dvěma komponentami programu, aniž by byly v přímé interakci a tím můžete přesně znát poskytované metody.
- **Memento (Memento)** – Bez porušování zapouzdření zachytíte a uložíte do externího objektu interní stav objektu tak, aby ten objekt mohl být do tohoto stavu kdykoliv později vrácen.
- **Template method (Šablonová metoda)** – Definuje kostru toho, jak nějaký algoritmus funguje, s tím, že některé kroky nechává na potomcích. Umožňuje tak potomkům upravit určité kroky algoritmu bez toho, aby mohli měnit strukturu algoritmu.

28 Objektově orientované paradigma. Pojmy třída, objekt, rozhraní. Základní vlastnosti objektu a vztah ke třídě. Základní vztahy mezi třídami a rozhraními. Třídní vs. instanční vlastnosti.

28.1 OOP

Objektově-orientované programování (OOP) je metodika vývoje softwaru, která jej odlišuje od procedurálního programování. Paradigma OOP popisuje způsob vývoje a zápisu programu a způsob uvažování o problému.

OOP definuje program jako **soubor spolupracujících komponent (objektů)** s přesně stanoveným **chováním a stavem**. Metody OOP napodobují vzhled a chování objektu z reálného světa s možností velké abstrakce.

28.1.1 Základní paradigma OOP

- Při řešení úlohy vytváříme **model popisované reality** – popisujeme entity a interakci mezi entitami.
- **Abstrahujeme** od nepodstatných detailů – při popisu/modelování entity vynecháváme jejich nepodstatné vlastnosti.
- Postup řešení je v řadě případů efektivnější než při procedurálním přístupu (ne vždy), kdy se úlohy řeší jako posloupnost příkazů.

28.1.2 Cíle OOP

- Je vedeno snahou o **znovupoužitelnost** komponent.
- Rozkládá složitou úlohu na dílčí součásti, které jdou pokud možno řešit nezávisle.
- Přiblížení struktury řešení v počítači reálnému světu (komunikující objekty).
- **Skrytí detailů implementace** řešení před uživatelem.

28.2 Třída

Třída představuje **základní konstrukční prvek OOP**. Třída slouží jako šablona pro vytváření **instancí tříd – objektů**. Seskupuje objekty stejného typu a podchycuje jejich podstatu na obecné úrovni. (Samotná třída tedy nepředstavuje vlastní informace, jedná se pouze o předlohu; data obsahují až objekty.) Třída definuje data a metody objektů.

28.3 Objekt, jeho vlastnosti a vztahy

Jednotlivé prvky modelované reality (jak data, tak související funkčnost) jsou v programu seskupeny do entit, nazývaných objekty. Objekt je **identifikovatelná samostatná entita** (dána jedinečností a chováním). Objekty si pamatují svůj stav a navenek poskytují operace přístupné jako metody pro volání. **Objekt** je také **instancí třídy**.

Ve vztahu k definovaným případům užití je nutné definovat takové interakce mezi objekty, které povedou ke splnění jejich funkcionality, účelu ke kterému byly navrženy. Jazyk UML poskytuje pro účely zaznamenání těchto vzájemných interakcí tzv. **sekvenční diagram**.

28.4 Rozhraní

Rozhraní entity je **souhrn informací, kterým entita specifikuje, co o ní okolí ví a jakým způsobem je možné s ní komunikovat**. Je to množina metod, která může být implementována třídou. Interface **pouze popisuje metody**, jejichž implementace však neobsahuje. Je **pojmenování skupiny externě viditelných operací**. Každá třída však může implementovat libovolný počet rozhraní, do jisté míry tedy rozhraní vícenásobnou dědičnost nahrazují. Implementace rozhraní není na hierarchii tříd nijak vázána a nevzniká z ní vztah dědičnosti.

28.5 Abstraktní třída

Je to takový hybrid mezi rozhraním a klasickou třídou. Od klasické třídy má schopnost implementovat vlastnosti (proměnné) a metody, které se na všech odvozených třídách budou vykonávat stejně. Od rozhraní zase získala možnost obsahovat prázdné **abstraktní metody**, které si každá odvozená podtřída **musí naimplementovat sama**. S těmito výhodami má abstraktní třída i pár omezení, a to že jedna podtřída **nemůže zdědit více abstraktních tříd** a od rozhraní přebírá omezení, že **nemůže vytvořit samostatnou instanci** (operátorem new).

28.6 Základní vztahy

Vztahy (relace) mezi třídami **specifikují cestu, jak mohou objekty mezi sebou komunikovat**. Relace složení částí do jednoho celku, má v podstatě dvě možné podoby. Jedná se o tzv. **agregaci**, pro kterou platí, že části mohou být obsaženy i v jiných celcích, jinými slovy řečeno, jsou sdíleny. Nebo se jedná o výhradní vlastnictví částí celkem, pak hovoříme o složení typu **kompozice**. Druhá z uvedených typu složení má jednu důležitou vlastnost z hlediska životního cyklu celku a jeho částí. Existence obou je totiž totožná. **Zánik celku (kompozitu) vede i k zániku jeho částí** na rozdíl od agregace, kde části mohou přežívat dále jako součástí jiných celků.

- **Asociace** popisuje **skupinu spojení (mezi objekty)** mající společnou strukturu a sémantiku. Vztah mezi asociací a spojením je analogický vztahu mezi třídou a objektem. Jinými slovy řečeno, jedná se tedy o dvousměrné propojení mezi třídami popisující množinu potenciálních spojení mezi instancemi asociovaných tříd stejně jako třída popisuje množinu svých potenciálních objektů.
- **Složení** popisuje **vztah mezi celkem a jeho částmi**, kde některé objekty definují komponenty jejichž složením vzniká celek reprezentovaný jiným objektem.
- **Závislost** reprezentuje slabší formu **vztahu mezi klientem a poskytovatelem služby**.
- **Zobecnění (generalizace)** je taxonomický **vztah mezi obecnějším elementem a jeho více specifikovaným elementem**, který je plně konzistentní s prvním z uvedených pouze k jeho specifikaci **přidává další konkretizující informaci**.

28.7 Rysy OOP

- **Skládání** – Objekt může obsahovat jiné objekty.
- **Delegování** – Objekt může využívat služeb jiných objektů tak, že je požádá o provedení operace.
- **Dědičnost** – objekty jsou organizovány stromovým způsobem, kdy objekty nějakého druhu mohou dědit z jiného druhu objektů, čímž přebírají jejich schopnosti, ke kterým pouze přidávají svoje vlastní rozšíření. Tato myšlenka se obvykle implementuje pomocí rozdělení objektů do tříd, přičemž každý objekt je instancí nějaké třídy. Každá třída pak může dědit od jiné třídy (v některých programovacích jazycích i z několika jiných tříd).
- **Polymorfismus** – odkazovaný **objekt se chová podle toho, jaké třídy je instancí**. Pozná se tak, že několik objektů poskytuje stejné rozhraní, pracuje se s nimi navenek stejným způsobem, ale jejich konkrétní chování se liší podle implementace. U **polymorfismu podmíněného dědičnosti** to znamená, že na místo, kde je očekávána instance nějaké třídy, můžeme dosadit i instanci libovolné její podtřídy, neboť rozhraní třídy je podmnožinou rozhraní podtřídy. U **polymorfismu nepodmíněného dědičnosti** je dostačující, jestliže se rozhraní (nebo jejich požadované části) u různých tříd shodují, pak jsou vzájemně polymorfní.
- **Zapouzdření** – zaručuje, že objekt **nemůže přímo přistupovat k „vnitřnostem“** jiných objektů, což by mohlo vést k nekonzistenci. Každý objekt navenek poskytuje rozhraní, pomocí kterého (a nijak jinak) se s objektem pracuje.

29 Mapování UML diagramů na zdrojový kód.

Důsledná a přesná specifikace objektů a jejich tříd v etapě návrhu umožňuje **automatické generování zdrojových kódů**. K mapování UML diagramů na kód dochází v první části implementační fáze vývoje.

Z **diagramu tříd** lze vygenerovat jednotlivá rozhraní, třídy s proměnnými a metodami (bez implementace) – <http://www.milosnemec.cz/clanek.php?id=199>. Úkolem samotné implementace je pak dopsat těla metod, jejichž chování může být popsáno v **diagramu aktivit**.

Analýza a návrh (UML)	Zdrojový kód (Java)
Třída	Struktura typu class
Role, Typ a Rozhraní	Struktura typu interface
Operace	Metoda
Atribut třídy	Statická proměnná označená static
Atribut	Instanční proměnná
Asociace	Instanční proměnná
Závislost	Lokální proměnná, argument nebo návratová hodnota zprávy
Interakce mezi objekty	Volání metod
Případ užití	Sekvence volání metod
Balíček, Subsystém	Kód nacházející se v adresáři specifikovaném pomocí package

Cílem implementace je **doplnit** navrženou architekturu (kostru) aplikace o **programový kód** a vytvořit tak kompletní systém.

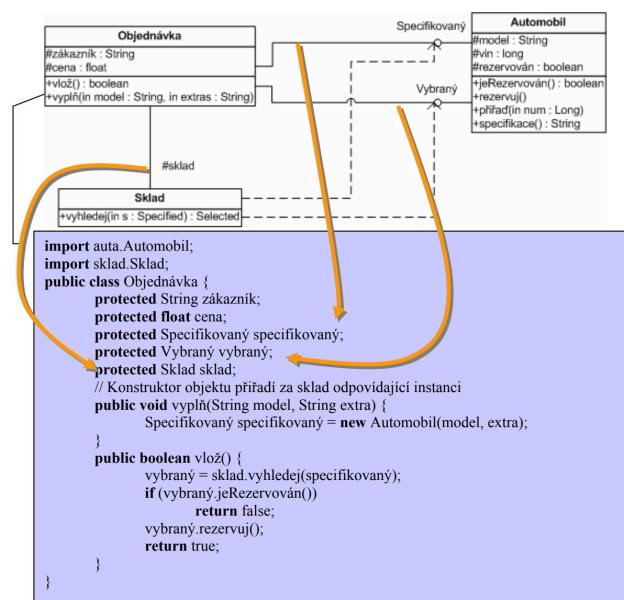
Implementační model specifikuje, jak jsou jednotlivé elementy (objekty a třídy) implementovány ve smyslu softwarových komponent.

Softwarová komponenta je definována jako **fyzicky existující** a **zaměnitelná část** systému, která vyhovuje požadované množině rozhraní a poskytuje jejich realizaci. Typy softwarových komponent dělíme na:

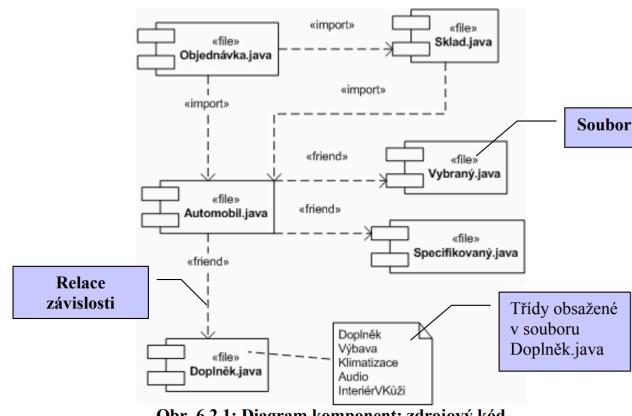
- **Zdrojové kódy** – části systému zapsané v programovacím jazyce.
- **Binární a spustitelné kódy** – přeložené do strojové kódu procesoru.
- **Ostatní části** – databázové tabulky, dokumenty apod.

Jestliže jsme ve fázi analýzy a návrhu pracovali pouze s abstrakcemi dokumentovanými v podobě jednotlivých diagramů, pak v **průběhu implementace dochází k jejich fyzické realizaci**. Implementační model se tedy také zaměřuje na specifikaci toho, jak budou tyto komponenty fyzicky organizovány podle implementačního prostředí, pro splnění těchto cílů lze využít následující diagramy:

- **Diagram komponent** – ilustruje organizaci a závislosti mezi softwarovými komponentami.
- **Diagram nasazení** – upřesňuje nejen konfiguraci technických prostředků, ale především rozmístění implementovaných softwarových komponent na těchto prostředcích.



Obr. 6.2.2: Zdrojový kód: Objednávka.java



Obr. 6.2.1: Diagram komponent: zdrojový kód

30 Správa paměti(v jazycích C/C++, Java , C#, Python), virtuální stroj, podpora paralelního zpracování a vlákna.

30.1 Správa paměti

Správa paměti je v soubor metod, které operační systém používá při **přidělování operační paměti jednotlivým spuštěným procesům**.

Výhody Automatické správy paměti

- Programátor se může věnovat řešení skutečného problému.
- Rozhraní modulů jsou přehlednější – není třeba řešit problém zodpovědnosti za uvolnění paměti pro objekty vytvořené různými moduly.
- Nastává **menší množství chyb** spojených s přístupem do paměti.
- Správa paměti je často pro nezkušené uživatele mnohem **efektivnější**.

Nevýhody Automatické správy paměti

- Paměť může být zachována jen proto, že je dostupná, i když není dále využita.
- Automatická správa paměti není k dispozici ve starších, ale často používaných jazycích.
- Může navíc využívat další zdroje PC a mít tak dopad na výkon (GC).

30.2 Úrovně správy paměti

- **Technické vybavení** – registry, cache
- **Operační systém** – virtuální paměť, segmentace, stránkování
- **Aplikace** – přidělování paměti, regenerace paměti:
 - **Manuální** – delete, dispose, free()
 - **Automatická** – garbage collector

30.3 Správa paměti v jednotlivých jazycích

Pro zjištění toho, které úseky paměti se již nepoužívají, je k dispozici mnoho algoritmů. Většinou spoléhá automatická regenerace paměti na informace o tom, na které bloky paměti **neukazuje žádná programová proměnná**. V zásadě existují dvě skupiny metod:

- metody založené na **sledování odkazů**,
- metody založené na **čítačích odkazů**.

30.3.1 C

Jazyk C umožňuje spravovat paměť staticky (automaticky) a dynamicky. **Staticky** alokované proměnné jsou umístěny do hlavní paměti (velice často s výkoným kódem programu). **Automaticky** alokované proměnné se alokují na zásobníků a vznikají/zanikají podle potřeb programu. Pro tyto proměnné platí, že jejich velikost musí být známá v čase komplikace.

Dynamická správa paměti je plně v **rukou programátora** a využívá se **4 funkcí** (`malloc`, `realloc`, `calloc`, `free`). Typicky je paměť alokována na haldě a přistupuje se k ní pomocí ukazatelů.

30.3.2 C++

C++ poskytuje mnoho způsobů jak na správu paměti, od **automatické** po **manuální** pomocí operátorů `new` a `delete`. **Automatická správa paměti** alokovaných objektů je z velké části postavena na návrhovém vzoru **RAII** (Resource Aquisition Is Initialization) – správa zdrojů daného objektu je vázána na jeho životnost a jeho destruktor je povinen jejich uvolnění. Jednoduše stačí získat zdroj (paměť, soubor, grafický handle, cokoli) a uložit ho do proměnné, která ho bude vlastnit a při jejím zániku (voláním destruktoru) ho také uvolní. A tím se o to můžete přestat starat, protože zbytek zařídí C++ kompilátor, který bude uvolňovat zdroje automaticky v rámci rušení lokálních proměnných na konci oboru platnosti (scope). Mezi **výhody** RAII patří:

- RAII uklízí nejenom paměť, ale všechny zdroje (soubory, mutex, databáze, transakce, síťové sockety).
- Úklid je proveden okamžitě bez prodlení, přesně v čase kdy zdroj přestává být potřeba.
- Proměnné a zdroje likviduje po skončení platnosti stejný thread, který je vytvořil.
- Není třeba zastavovat program kvůli GC jako v případě Javy, LISPu, atd.

Má však i **nevýhody**:

- Je nepatrň pomalejší, protože dealokuje okamžitě jeden podruhé, zatímco klasické GC dealokují naráz větší množství paměti.
- Je třeba podpořit RAII věnovat pozornost při implementaci tříd (implementace destruktoru).

Smart pointery představují další způsob jak na automatickou správu paměti dynamicky alokovaných objektů. SP jsou definované jako třída, která overloaduje operátory `->`, `*`, `->*`, což ji umožňuje „chovat se jako pointer“ a fungovat ve většině kódu v kombinaci s reálnými a smart pointery. Existuje mnoho typů, které definují typ vlastnictví a podmínky, za nichž je objekt dealokován (`shared_ptr`, `weak_ptr`, `unique_ptr`). Princip dealokace je stejný jako u RAII.

30.3.3 C#

Správa paměti je v jazyce C# **plně automatizovaná**, paměťový prostor se přiděluje operátorem `new` a jeho uvolnění zajistí systém **řízení běhu programu**. V .NET se používá varianta GC v podobě **Mark & Sweep**.

V případě, že pracujeme s **neřízenými zdroji** (systémové zdroje – soubory, připojení k DB, síť), které je třeba **explicitně uvolnit**, implementuje se rozhraní **IDisposable**. To předepisuje jedinou metodu `Dispose()`, kterou je třeba zavolat po skončení práce s objektem (stará se o uvolnění zdrojů, samotný objekt poté existuje dál až do uvolnění GC).

Dále lze v .NET definovat u tříd i tzv. safe-guard v podobě **finalizer** (metoda, která se volá při uklízení objektu pomocí GC). Syntaxe je podobná C++ (`~ClassName()`). Jelikož však zhoršuje efektivitu, definuje se pouze u tříd s rozhraním **IDisposable**. Jeho cílem je uvolnit zdroje, **pokud uživatel objektu nezavolá/zapomene zavolat** metodu `Dispose()`.

30.3.4 Java

V jazyce Java je správa paměti rovněž **plně automatizovaná** a o její uvolnění se stará GC (varianta **Mark & Sweep**). Java uchovává všechny reference na Stacku a na heapu vytváří objekty. Programátor má možnost vytvářet **strong** (klasická), **weak** (pokud na objekt ukazují pouze weak reference, tak bude uvolněn), **soft** (zajištěno uvolnění při nedostatku paměti před vyhozením chyby `OutOfMemory`) reference, které definují kdy bude objekt dealokován.

Speciálním případem jsou Stringy (jsou **immutable**), kdy Java spravuje tzv. **String pool**. To znamená, že si ukládá a snaží se stringy znovupoužívat jak je to možné. To neplatí pro stringy, které jsou vypočítány, pouze pro konstanty. Můžeme však přinutit JVM k vložení těchto stringů do poolu pomocí metody `.intern()`. V Javě existují **3 typy GC**:

- **Serial GC** – kolektor, který běží pouze v jednom jádře (malé aplikace, malé využití dat).
- **Parallel GC** – oproti serial využívá více threadů.
- **Mostly concurrent GC** – při běhu GC vždy dojde k pozastavení aplikace, tento GC se tomu snaží předejít tak, že jede **souběžně** s aplikací. Nefunguje však souběžně 100% času, kdy je někdy nutné program pozastavit, tyto pauzy se však snaží být co nejkratší pro zajištění co nejlepšího výkonu.

Podobně jako v C#, Java poskytuje rozhraní **AutoCloseable**, které poskytuje operaci `close()`. Nejedná se o přímého zástupce za **IDisposable**, ale toto rozhraní umožňuje v Java využívat tzv. **try-with-resources** (ekvivalent v C# `using`), kdy dochází k automatickému zavolání metody `close()` na konci `try-catch` bloku v části `finally`.

30.4 Garbage collector (GC)

Je obvykle část běhového prostředí (programovacího) jazyka, nebo přídavná knihovna (podporovaná komplátorem, hardware, operačním systémem, nebo jakoukoli kombinací těchto tří). Má za úkol **automaticky určit**, která část paměti programu je už **nepoužívaná**, a připravit ji pro další znovupoužití.

30.4.1 Mark & Sweep

Algoritmus nejdříve nastaví všem objektům, které jsou v paměti, **speciální příznak** navštíven na hodnotu ne. Poté projde všechny objekty, ke kterým se lze dostat. Těm, které takto navštívily, nastaví příznak na hodnotu ano. V okamžiku, kdy se už nemůže dostat k žádnému dalšímu objektu, znamená to, že všechny objekty s příznakem navštíven majícím hodnotu ne jsou odpad – a mohou být tedy uvolněny z paměti.

Tato metoda má několik nevýhod. Největší je, že při garbage collectionu je **přerušen běh programu**. To znamená, že programy **pravidelně zamrzou**, takže je nemožné pracovat s aplikacemi pracujícími v reálném čase.

30.4.2 Reference counting

Ke každému objektu je přiřazen **čítač referencí**. Když je objekt vytvořen, jeho čítač je nastaven na **hodnotu 1**. V okamžiku, kdy si nějaký jiný objekt uloží referenci na tento objekt, hodnota čítače je **zvětšena o 1**. Ve chvíli, kdy je referenční hodnota mimo rozsah platnosti (např. po opuštění funkce, která si referenci uložila), nebo když je referenční hodnota snížena novou hodnotou, čítač je **snížen o 1**. Jestliže je hodnota čítače některého objektu nulová, může být tento objekt uvolněn z paměti.

30.4.3 Generační algoritmus

Staví na **dvou základních principech**:

- Mnoho objektů se stane **odpadem** krátce **po svém vzniku**.
- Jen malé procento **referencí** ve „starších“ objektech **ukazuje na objekty mladší**.

Rozděluje si paměť programu do několika částí, tzv. „generací“. Objekty jsou vytvářeny ve spodní (nejmladší) generaci a po splnění určité podmínky, obvykle stáří, jsou přeřazeny do starší generace. Pro každou generaci může být **úklid** prováděn v rozdílných **časových intervalech** (nejkratší intervaly obvykle budou platit pro nejmladší generaci) a dokonce pro rozdílné generace mohou být použity **rozdílné algoritmy úklidu**. V okamžiku, kdy se prostor pro spodní generaci zaplní, všechny dosažitelné objekty v nejmladší generaci jsou zkopírovány do starší generace. I tak bude množství kopírovaných objektů pouze zlomkem z celkového množství mladších objektů, jelikož většina z nich se již stala odpadem.

30.4.4 Nevýhody GC

- Garbage collector potřebuje ke své práci **procesorový čas**, aby mohl rozhodovat o tom, jestli je objekt v paměti „mrtvý“, nebo „živý“.
- Některé garbage collectory mohou způsobit i dosti znatelné **pauzy**, což je vážný problém pro systémy běžící v reálném čase.
- O stavu objektů musí mít garbage collector uloženou informaci. Tyto informace vyžadují určitou **paměť navíc**.
- Některé jazyky s garbage collectorem neumožňují programátorovi **znovupoužití paměti**, i když ví, že paměť už nebude použita. To vede k **nárůstu alokace paměti**.

30.5 Virtuální stroj

Je v informatice software, který vytváří **virtualizované prostředí** mezi platformou počítače (HW i SW) a operačním systémem, ve kterém koncový uživatel může provozovat software na **abstraktním stroji**.

30.5.1 Hardwarový virtuální stroj

Původní význam pro virtuální stroj, někdy nazývaný též hardwarový virtuální stroj, označuje **několik jednotlivých totožných pracovních prostředí na jediném počítači**, z nichž na každém běží operační systém. Díky tomu může být aplikace psaná pro jeden OS používána na stroji, na kterém běží jiný OS, nebo zajišťuje vykonání sandboxu, který poskytuje větší úroveň izolace mezi procesy, než je dosaženo při vykonávání několika procesů najednou (multitasking) na stejném OS.

Jedním využitím může být také poskytnutí iluze mnoha uživatelům, že používají celý počítač, který je jejich „soukromým“ strojem, izolovaným od ostatních uživatelů, přestože všichni používají jeden fyzický stroj. Další výhodou může být to, že start (bootování) a restart virtuálního počítače může být mnohem rychlejší, než u fyzického stroje, protože mohou být přeskoveny úkoly jako například inicializace hardwaru.

Podobný software je často označován termíny jako virtualizace a virtuální servery. Hostitelský software, který poskytuje tuto schopnost je často označovaný jako **hypervisor** nebo virtuální strojový monitor (virtual machine monitor). **Softwarové virtualizace** mohou být prováděny ve **třech hlavních úrovních**:

- **Emulace** – virtuální stroj simuluje kompletní hardware, dovolující provoz nemodifikovaného OS na úplně jiném procesoru.
- **Paravirtualizace** – virtuální stroj nesimuluje hardware, ale místo toho nabídne **speciální rozhraní API**, které vyžaduje určité modifikace hostovaného operačního systému, aby mohl být tento OS nad virtuálním strojem spouštěn.
- **Nativní virtualizace a „plná virtualizace“** – virtuální stroj jen částečně simuluje dost hardwaru, aby mohl nemodifikovaný OS běžet samostatně, ale hostitelský OS musí být určený pro stejný druh procesoru. Pojem nativní virtualizace se někdy používá ke zdůraznění, že je **využita hardwarová podpora pro virtualizaci** (tzv. virtualizační technologie) (VMware, Parallel).

30.5.2 Aplikační virtuální stroj

Dalším významem termínu virtuální stroj je počítačový software, který **izoluje aplikace používané uživatelem na počítači**. Protože verze virtuálního stroje jsou psány pro **různé počítačové platformy**, jakákoliv aplikace psaná pro virtuální stroj může být provozována na kterékoli z platforem, místo toho, aby se musely vytvářet oddělené verze aplikace pro každý počítač a operační systém. Aplikace běžící na počítači používá **interpret** nebo **Just in time komplikaci**.

Jeden z nejlepších známých příkladů aplikačního virtuálního stroje je **Java Virtual Machine (JVM)** od firmy Sun Microsystems. Java Virtual Machine umí zpracovat mezikód (**Java bytecode**), který je obvykle vytvořen ze zdrojových kódů programovacího jazyka Java. Díky tomu, že je JVM k dispozici na mnoha platformách, je možné aplikaci v Javě vytvořit pouze jednou a spustit na kterékoliv z platforem, pro kterou je vyvinut JVM (např. Windows, Linux).

30.5.3 Virtuální prostředí

Virtuální prostředí (jinak virtuální soukromý server) je jiný druh virtuálního stroje. Ve skutečnosti, to je **virtualizované prostředí** pro běh programů **na úrovni uživatele** (tj. ne jádra operačního systému a ovladače, ale aplikace). Virtuální prostředí jsou vytvořena použitím softwaru zavádějícího virtualizaci na úrovni operačního systému, jako například Virtuozzo, OpenVZ. Příkladem může být **VPS** u poskytovatelů stránek.

30.6 Podpora paralelního zpracování

Paralelně programovaný software využívá možnost rozdělení jednoho velkého výpočetního problému na několik menších problémů, které jsou řešeny „současně“. Prvky sloužící k paralelnímu zpracování výpočtu mohou být různé. Jedná se například o jeden **počítač s více procesory**, **několik počítačů v síti**, **specializovaný hardware** nebo **kombinaci** těchto prvků.

30.7 Thread

Operační systémy používají pro oddělení různých běžících aplikací procesy. **Proces** je tvořen paměťovým prostorem a jedním nebo více vlákny. **Vlákno je samostatně prováděný výpočetní tok** (posloupnost instrukcí), který běží v rámci procesu. Každému vláknu přísluší vlastní priorita a řada systémových struktur.

Operační systémy s **preemptivním multitaskingem** vytvářejí dojem souběžného provádění více vláken ve více procesech. To je zajištěno rozdělením času procesoru mezi jednotlivá vlákna po malých časových intervalech. Pokud časový interval vyprší, je běžící vlákno pozastaveno, uloží se jeho kontext a obnoví se kontext dalšího vlákna ve frontě, jemuž je pak předáno řízení. Vzhledem k tomu, že tyto časové úseky jsou z pohledu

uživatele velmi krátké, je výsledný dojem i na počítači s jediným procesorem takový, jako by pracovalo více vláken současně. V případě, že máme k dispozici více procesorů, jsou mezi ně vlákna přidělována ke zpracování a k současnemu běhu pak skutečně dochází.

Přepnutí mezi vláknem bývá výrazně rychlejší než v procesovém multitaskingu, neboť vlákna **sdílejí stejnou paměť** a uživatelská práva svého mateřského procesu a není je třeba při přepínání měnit. Vlákno také spotřebuje méně paměti a je rychlejší na vytváření.

Vlákna je možné vytvořit i čistě na **aplikační úrovni** bez nativní podpory operačního systému (využitím sdílené paměti a dalších technik). Takto vzniklá vlákna je poté možné spouštět postupně v jednotlivých procesech operačního systému nebo takzvaně m:n, tedy v několika vláknach operačního systému současně spouštět větší počet aplikačních vláken.

Samotným zvyšováním počtu vláken však obvykle odpovídajícího zvýšení výkonu aplikace nedosáhneme. Naopak se doporučuje, abychom používali co nejméně vláken a tím omezili spotřebu systémových prostředků a nárůst režie. Typické problémy jsou následující:

- Pro ukládání kontextových informací se **spotřebovává dodatečná paměť**, a tedy celkový počet procesů a vláken, které mohou v systému současně existovat, je **omezený**.
- Obsluha velkého počtu vláken **spotřebovává významnou část času procesoru**. Existuje-li tedy příliš mnoho vláken, většina z nich příliš významně nepostupuje. Navíc pokud je většina vláken v jednom procesu, dostávají se vlákna jiných procesů na řadu méně často.
- Organizace programu s mnoha vláknem je složitá a může být zdrojem mnoha chyb. Zejména je obtížné zajistit jejich **správnou synchronizaci**.

30.7.1 Shrnutí

- Vlákno je **odlehčený proces**, pomocí něhož se snižuje režie OS při změně kontextu (umožňuje multitasking).
- Vlákno je **samostatně prováděný výpočetní tok**.
- Vlákna běží v **rámci procesu**.
- Vlákna jednoho procesu běží v rámci stejného prostoru paměti. **Sdílí jeho prostředky a paměť**.
- Každé vlákno má vyhrazený prostor pro specifické proměnné (runtime prostředí).
- Pokud běží v jádře OS dochází k **paralelizaci**, simulování threadů v aplikaci paralelizaci pouze simuluje.

30.8 Kdy použít vlákna?

Vlákna je výhodné použít, pokud aplikace splňuje některé následující kritérium:

- Je **složena z nezávislých úloh**.
- Může být **blokována** po dlouho dobu.
- Obsahuje **výpočetně náročnou část**.
- Musí reagovat na asynchronní události.
- Obsahuje úlohy s nižší nebo vyšší prioritou než zbytek aplikace.

30.8.1 Typické aplikace

- **Servery** – obsluhují více klientů najednou. Obsluha typicky znamená **přístup k několika sdíleným zdrojům** a hodně vstupně výstupních operací (I/O).
- **Výpočetní aplikace** – na **víceprocesorovém systému** lze výpočet urychlit rozdělením úlohy na více procesorů.
- **Aplikace reálného času** – lze využít specifických rozvrhovačů. Více vláknová aplikace je výkonnější než složité asynchronní programování, neboť vlákno čeká na příslušnou událost namísto přerušování vykonávání kódu a přepínání kontextu.

30.9 Vlákna v Javě

Každé vlákno v Javě je instancí třídy `java.lang.Thread`. Tato třída zajišťuje spuštění, zastavení a ukončení vlákna. Vlákno musí implementovat metodu `run`, která definuje činnost vlákna. Této metodě je předáno řízení po spuštění vlákna metodou `start`.

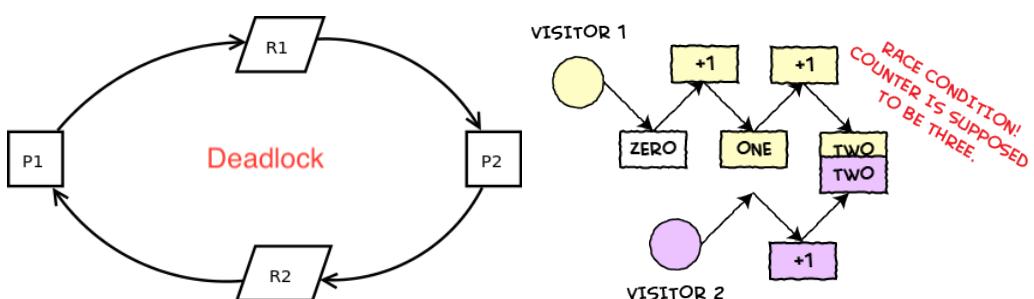
30.9.1 Životní cyklus

Vlákno v průběhu svého života prochází posloupností následujících stavů:

- **New** – bezprostředně po vytvoření ještě nejsou vláknu přiděleny žádné systémové prostředky, vlákno neběží.
- **Runnable** – v tomto stavu se nachází po provedení metody `start()`, scheduler jej však nezvolil, aby byl běžící.
- **Running** – vlákno běží, do tohoto stavu se dostane po té co jej vybere scheduler.
- **Not runnable (Blocked)** – vlákno je pozastaveno voláním jedné s metod `sleep`, `wait` nebo `suspend`, případně čeká na dokončení operace vstupu/výstupu.
- **Terminated** – thread je ukončen nebo ve stavu **Dead** pokud existuje metoda `run`.

30.10 Hlavní problémy vícevláknových aplikací

- **Deadlock (uváznutí)** – je situace, kdy úspěšné dokončení první akce je **podmíněno předchozím dokončením druhé akce**, přičemž druhá akce může být dokončena až po dokončení první akce. Deadlocku můžeme zabránit například tím, že proces musí o **všechny prostředky**, které potřebuje, zažádat **najednou**. Budť je všechny dostane, nebo nedostane ani jeden.
- **Souběh (race conditions)** – **přístup více vláken ke sdíleným proměnným** a alespoň jedno vlákno nevyužívá synchronizačních mechanismů. Vlákno čte hodnotu, zatímco jiné vlákno zapisuje. Zápis a čtení nejsou atomické a data mohou být neplatná. K zabránění souběhu se používají **zámkы (lock)**, jejichž použitím se zajistí, že jen jedno vlákno může v jeden okamžik přistoupit k označenému resource souboru nebo části kódu.
- **Starvation (vyhľadovění)** – stav, kdy jsou vláknu neustále odeplírány prostředky. Bez těchto prostředků program nikdy nedokončí svůj úkol.



30.11 Mechanismy synchronizace

- **Mutex** – zámek kritické sekce.
- **Podmíněná proměnná** (condition variable) synchronizace hodnotou proměnné. Čekání vlákna na probuzení od jiného vlákna.
- **Semafor** – čítač, definuje kolik procesů může max přistupovat k nějakému zdroji.

30.11.1 Mutex

Mutex = mutual exclusion, neboli vzájemné vyloučení je algoritmus používaný v programování jako **synchroni-zační prostředek**. Zabraňuje tomu, aby byly současně vykonávány dva (nebo více) kódy nad stejným sdíleným prostředkem. Základní operace:

- **Lock** – uzamknutí mutexu (přiřazení mutexu vláknu). Pokud nelze mutex získat, vlákno přechází do blokovaného režimu a čeká na uvolnění zámku.
- **Unlock** – uvolnění zámku. Pokud jiná vlákna čekají na uvolnění, je vybráno jedno vlákno, které mutex získá.
- **Rozšířené metody:**
 - **Rekursivní** – vícenásobné zamykání stejným vláknem.
 - **Try** – okamžitý návrat pokud není možné mutex získat.
 - **Timed** – pokus o získání zámku s omezenou dobou čekání.

30.11.2 Semafor

Semafora jsou velmi podobné mutexům. Zatímco mutex má jen dva stavy – zamknut/odemknut, semafor jich může mít více. Semafor je v podstatě **čítač**, který může být **zmenšován** a **zvětšován**. Když je čítač roven **nule** je semafor považován za **zamčený**, v opačném případě je **odemčen**. Semafora se používají, pokud máme omezený počet nějakých zdrojů. Pokud nějaké vlákno chce ke zdroji přistupovat, zmenší hodnotu semaforu. Pokud semafor nebyl roven nule, proces pokračuje dál, v opačném případě čeká, až jiné vlákno přestane zdroj používat a hodnotu semaforu zvětší.

30.12 Rozdělování práce vláknům

Modely řeší způsob **vytváření** a **rozdělování práce** mezi vlákna:

- **Boss/Worker** – hlavní vlákno, řídí rozdělení úlohy jiným vláknům.
- **Peer** – vlákna běží paralelně bez specifického vedoucího.
- **Pipeline** – zpracování dat sekvencí operací. Předpokládá dlouhý vstupní proud dat.

31 Zpracování chyb v moderních programovacích jazyčích, princip datových proudů – pro vstup a výstup. Rozdíl mezi znakově a bytově orientovanými datovými proudy.

Výjimky představují určité situace, ve kterých musí být **výpočet přerušen** a řízení předáno na místo, kde bude provedeno ošetření výjimky a rozhodnutí o dalším pokračování výpočtu. Starší programovací jazyky pro ošetření chyb během výpočtu obvykle žádnou podporu neměly. U všech funkcí, které mohly objevit chybu, bylo třeba stále testovat různé příznaky a speciální návratové hodnoty, kterými se chyba oznamovala, a v případě, že jsme na testování chyby zapomněli, program běžel dál a na chybu v nejlepším případě nereagoval nebo se zhroustil na zcela jiném místě, kde již bylo obtížné zdroj chyby dohledat.

Jazyk Java, podobně jako např. C++, využívá pro ošetření výjimek metodu strukturované obsluhy. To znamená, že programátor může pro konkrétní úsek programu specifikovat, **jakým způsobem se má konkrétní typ výjimky ošetřit**. V případě, že výjimka nastane, vyhledá se vždy nejbližší nadřazený blok, ve kterém je výjimka ošetřena.

31.1 Java

Výjimky jsou v jazyce Java reprezentovány jako objekty. Tyto objekty jsou instancemi tříd odvozených obecně od třídy **Throwable** se dvěma podtřídami **Error** a **Exception**:

- **Error** – tyto výjimky představují vážný problém v činnosti aplikace a programátor by je **neměl zachytávat** (nedostatech zdrojů pro práci virtuálního stroje, přetečení zásobníku, nenalezení potřebné třídy při classloadingu) – **unchecked**.
- **Exception** – ošetření těchto výjimek má smysl. Jde například o výjimky jako pokus o otevření neexistujícího souboru, chybný formát čísla, dělení nulou apod. Do této skupiny by měly patřit také veškeré výjimky **definované uživatelem**. Jedná se o **checked exceptions** – kompilátor kontroluje zda jsou ošetřeny při komplikaci.
- **RuntimeException** – potomek třídy **Exception**, značí obvyklé chyby, které **způsobil sám programátor** (neplatný index pole, volání nad nulovým ukazatelem). Jedná se o tzv. **unchecked exceptions**, tedy není při komplikaci kontrolováno zda jsou zachytávány **catch**, **throws**.

V Javě lze zachytávat výjimky pomocí bloku **try-catch-finally** nebo klíčovým slovíčkem **throws** v **hlavičce metody**, který přenechá její ošetření nadřazenému bloku. Vlastní výjimky lze vracet slovíčkem **throw**.

31.2 C#

- Všechny výjimky v C# jsou jsou **unchecked (nehlídané)** – kompilátor nekontroluje zda se ošetřují.
- Odvozena z třídy **Exception** nebo z některé z jejich následníků.
- Obsahuje informace o: svém **původu, důvodu vzniku**.

Výjimky fungují stejně jako v Javě, v případě neošetřené chyby dojde k **ukončení programu** s odpovídající **běhovou chybou**. Mechanismus výjimek je v jazyce C# založen na klíčových slovech: **try**, **catch**, **finally**. Vlastní vyhození výjimky lze pomocí klíčového slova **throw**.

31.3 try-catch-finally

Zpracování výjimky se vždy vztahuje k bloku programu vymezeného příkazem **try** následovaném **jedním nebo více** bloky **catch** popisujícími způsob ošetření jednotlivých výjimek a volitelným blokem **finally**, který se vykoná vždy na konci bloku (vhodné místo pro uvolnění zdrojů).

V C++ neexistuje blok **finally**. Díky technice RAII blok **finally** není potřeba, proto asi nikdy v C++ obsažen ani nebude.

```
class MojeVýjimka extends Exception {
    MojeVýjimka(String msg) { super(msg); }
}

class Výjimky {
    static void zpracuj(int x) throws MojeVýjimka {
        System.out.println("Volani zpracuj " + x);
```

```

if( x < 0 ) { throw new MojeVyhledka("Parametr nesmi byt zaporny"); }

public static void main(String args[]) {
    try {
        zpracuj(1);
        zpracuj(-1);
    } catch( MojeVyhledka e ) {
        e.printStackTrace();
    } finally {
    }
}
}

```

31.4 Assert

Assert (od verze 1.4) přináší snadný způsob jak vkládat do programu **jednoduché kontroly** (např.: zda je hodnota > 0), které se vyhodnotí a přeloží pouze pokud při překladu zadáme parametr `-ea`. To se hodí zejména při vývoji aplikace a ladění, kdy v produkčním buildu se program přeloží bez těchto kontrol (bez `-ea`) a výpočet se tak nikde nezdržuje.

```

class TestAssert {
    static double prevraca_hodnota(double x) {
        assert x != 0.0 : "Argument nesmi byt nulovy";
        return 1.0 / x;
    }

    public static void main(String args[]) {
        System.out.println(prevraca_hodnota(1));
        System.out.println(prevraca_hodnota(0));
    }
}

```

31.5 Datové proudy

Datové proudy jsou **sekvence dat**. Proud je definován svým **vstupem** a **výstupem**, těmi mohou být například soubor na disku, zařízení (vstupní klávesnice nebo myš, výstupní displej) nebo jiný program.

Proudy také rozlišujeme na **binární** a **znakové**. Jak již názvy napovídají, tak zatímco binární proudy využijeme pro libovolná binární data (tj. data vkládáme je po bajtech), tak znakové proudy jsou určeny pouze pro text (znaky).

Proudy jsou základní cestou jak pracovat s daty, **náhodným** i **sekvenčním** přístupem. Mezi nejjednodušší stream patří výpis na obrazovku `System.out.print();` `console.WriteLine();`.

31.6 Streamy v .NET

- **Textové:** `StreamWriter`, `StreamReader`
- **Binární:** `BinaryWriter`, `BinaryReader`
- **Další:** `MemoryStream`, `GZipStream`, `Bufferedstream`

31.7 Streamy v Javě

V Javě všechny vstupní streamy dědí z abstraktní třídy **InputStream** a všechny výstupní z **OutputStream**.

31.7.1 Binární

Binární proudy umožňují přenést **libovolná data**. Základní operací definovanou v `InputStream` je metoda `read`, pomocí které můžeme z proudu přečíst jeden byte. Analogicky výstupní proud definuje metodu `write`. **Čtení a zápis po jednotlivých bytech je velmi pomalý**, zvláště pokud uvážíme, že na druhé straně proudu může být disk – a každý dotaz může velmi snadno znamenat nutnost nového vystavení hlaviček.

Třídy **BufferedInputStream** a **BufferedOutputStream** (v Javě) za nás tento nedostatek řeší. Tyto proudy obsahují **pole**, které slouží jako **vyrovnávací paměť**. V případě čtení z disku bufferovaný proud načte celý blok dat a uloží jej do pole, ze kterého data dále posílá našemu programu. V okamžiku, kdy se pole vyprázdní, učiní dotaz na další blok. Tímto způsobem dochází k eliminaci velkého množství zbytečných a drahých volání. (V .NET existuje třída **BufferedStream**.)

31.7.2 Textový

Znakové proudy fungují stejným způsobem jako ty binární, pouze **operují s textem**. Poměrně důležitou poznámkou je, že **Java** interně uchovává řetězce ve znakové sadě **Unicode**, C# využívá také Unicode, konkrétně UTF-16. Z toho plyne, že při každém zápisu a čtení ze znakového proudu dochází k překódování daného řetězce (znaků). **BufferedReader/BufferedWriter**.

31.7.3 Datové proudy

Java obsahuje třídy pro **pohodlné čtení a zápis primitivních datových typů** a typu **String**. Nejrozšířenější třídy jsou **DataInputStream** a **DataOutputStream**. **String** je ukládán v kódování UTF-8. Údaje takto uložené např. do souboru nejsou uživatelsky přívětivé a s výjimkou řetězců čitelné. Metody pro čtení jsou **readDouble**, **readInt**, **readUTF**. Obdobně metody pro zápis mají předponu **write**.

31.7.4 Objektové proudy

Většina standardních tříd implementuje rozhraní **Serializable** (serializovatelný), které je nezbytné pro jejich podporu objektovými proudy. Objektové proudy rozšiřují datové proudy, takže objektové proudy **umí pracovat i s primitivními datovými typy**. Nové metody jsou **readObject** a **writeObject**. Pokud metoda **readObject** vrátí jiný než očekávaný objekt, vyhodí výjimku typu **ClassNotFoundException**. Pokud se objekt neskládá jen z primitivních typů ale i z referencí na další objekty, je potřeba zachovat tyto reference. Proto je při zápisu objektu uložit i všechny objekty, na které má daný objekt odkaz. Podobně se bude chovat čtecí proud, který se bude snažit zrekonstruovat celou takovou síť objektů.

32 Jazyk UML – typy diagramů a jejich využití v rámci vývoje.

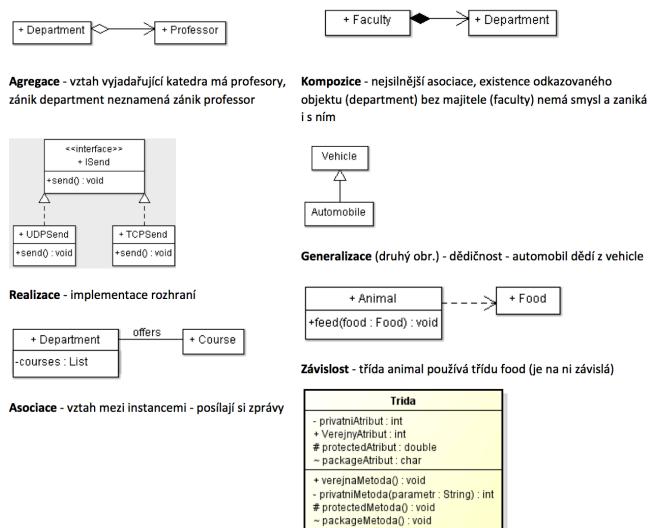
UML je jazyk umožňující **specifikaci** (struktura a model), **vizualizaci** (grafy), **konstrukci** (vygenerování kódu, např. class diagram) a **dokumentaci artefaktů** softwarového systému. V průběhu let se UML stal **standardizovaným jazykem** určeným pro vytvoření výkresové dokumentace (softwarového) systému v **různých fázích vývoje**. K vytváření jednotlivých modelů systému jazyk UML poskytuje celou řadu diagramů umožňujících **postihnout různé aspekty systému**. Jedná se celkem o čtyři základní náhledy a k nim přiřazené diagramy:

1. Funkční náhled

- (a) **Diagram případů užití** – popisuje vztahy mezi aktéry a jednotlivými případy použití. Poskytuje funkční náhled na systém (kdo se systémem pracuje a jak). Uplatňuje se pro realizaci **DFD** (Data flow diagram) ve fázi **specifikace požadavků** (VIS).

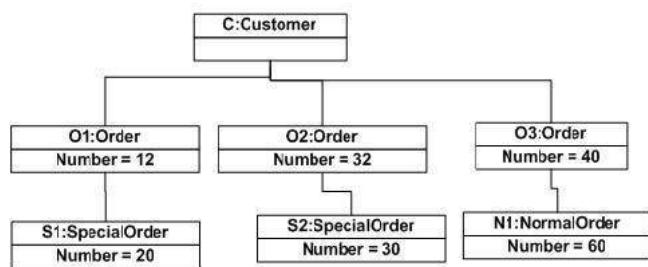
2. Logický náhled

- (a) **Diagram tříd** – specifikuje množinu tříd, rozhraní a jejich vzájemné vztahy. Tyto diagramy slouží k vyjádření statického pohledu na systém.



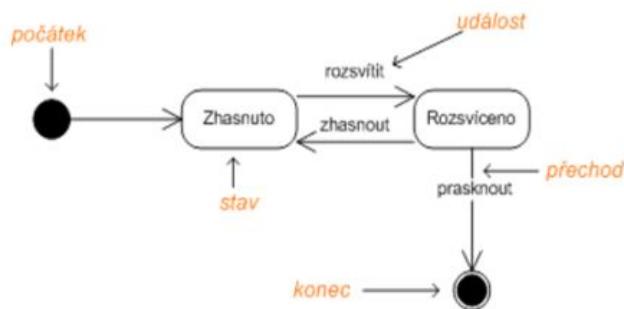
- (b) **Objektový diagram** – zobrazuje **instance tříd** (objekty), někdy nazýván jako instanční diagram. Je **snímkem objektů** a jejich vztahů v systému v **určitém časovém okamžiku**, který chceme z nějakého důvodu zdůraznit. Využívá se v datové analýze pro ERD.

Object diagram of an order management system



3. Dynamický náhled popisující chování

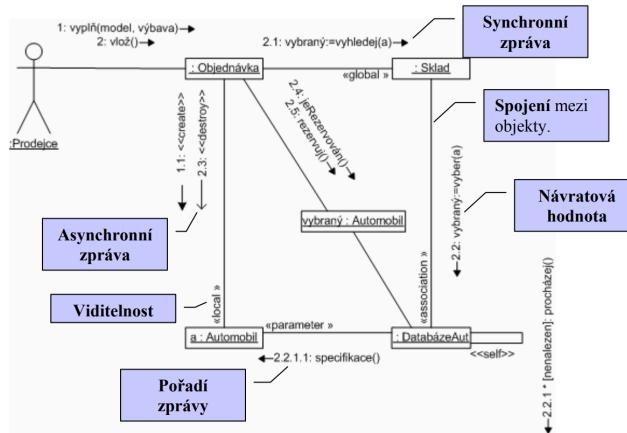
- (a) **Stavový diagram** – dokumentuje **životní cyklus** objektu dané třídy z hlediska jeho **stavů**, **přechodů** mezi těmito stavů a **událostmi**, které tyto přechody uskutečňují.



- (b) **Diagram aktivit** – popisuje podnikový proces pomocí jeho stavů reprezentovaných vykonáváním aktivit a pomocí přechodů mezi těmito stavů způsobených ukončením těchto aktivit. Účelem diagramu aktivit je blíže popsat tok činností daný vnitřním mechanismem jejich provádění.
 - (c) **Sekvenční diagramy** – popisuje interkace mezi objekty z hlediska jejich **časového uspořádání**.
 - (d) **Diagramy spolupráce** – je obdobně jako předchozí sekvenční diagram zaměřen na interkace, ale z pohledu strukturální organizace objektů. Jinými slovy není primárním aspektem časová posloupnost posílaných zpráv, ale **topologie rozmístění objektů**.

Časová posloupnost zaslání zpráv je vyjádřena jejich **pořadovým číslem**. Návratová hodnota je vyjádřena **operátorem přiřazení** `:=`. Opakování zaslání zprávy je dáno symbolem `*` a v hraničních závorkách uvedením podmínky opakování cyklu. Navíc tento diagram zavádí i následující **typy vlastnosti** vzájemně spojených objektů:

- i. **<<local>>** – vyjadřuje situaci, kdy objekt je vytvořen v těle operace a po jejím vykonání je zrušen,
 - ii. **<<global>>** – specifikuje globálně viditelný objekt,
 - iii. **<<parameter>>** – vyjadřuje fakt, že objekt je předán druhému jako argument na něj zaslané zprávy,
 - iv. **<<association>>** – specifikuje trvalou vazbu mezi objekty (někdy se také hovoří o tzv. známostním spojení).



Obr. 5.3.2: Diagram spolupráce

4. Implementační náhled

- (a) **Diagram komponent** – ilustruje organizaci a **závislosti mezi softwarovými komponentami**. Zdrojové komponenty tvoří soubory vytvořené použitým programovacím jazykem. Diagram spustitelných komponent specifikuje všechny komponenty vytvořené námi i ty, které nám dává k dispozici implementační prostředí.

(b) **Diagram nasazení** – upřesňuje nejen ve smyslu konfigurace technických prostředků, ale především z hlediska rozmístění implementovaných softwarových komponent na těchto prostředcích.

32.1 Diagramy a jejich použití v rámci fází vývoje

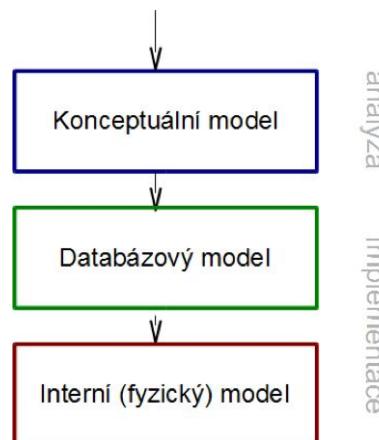
- **Specifikace požadavků:** Diagram případů užití, Sekvenční diagramy, Diagram aktivit.
 - **Návrh:** Diagram tříd, Objektový diagram, Stavový diagram, Sekvenční diagramy.
 - **Implementace:** Diagramy spolupráce, Diagram komponent, Diagram nasazení.

Část IV

Databázové a informační systémy

33 Modelování databázových systémů, konceptuální modelování, datová analýza, funkční analýza; nástroje a modely.

33.1 Modelování databázových systémů



Databázový systém můžeme modelovat **třemi datovými modely**. Ve fázi analýzy se používá **konceptuální model**, který modeluje realitu na logickou úroveň databáze. Konceptuální model je výsledkem datové analýzy a je **nezávislý na konkrétní implementaci**.

V implementační fázi si pak pomáháme **databázovými modely**, kde modelujeme vazby a vztahy (realitu) na konkrétní tabulky (obecně SŘBD). Databázový model můžeme dále dělit na **relační** a **síťový** model. **Fyzickým uložením dat** na paměťové médium se zabývá **interní model**.

33.1.1 Základní pojmy

- **Entita** – objekt reálného světa, konkrétní výskyt instance entitního typu.
 - **Entitní typ** – něco jako třída, je popsán jménem a množinou atributů (množina entit se stejnými atributy).
 - **Atribut** – vlastnost entity (možné hodnoty jsou označeny jako **doména atributu**).
 - **Klíč** – množina atributů, která jednoznačně **určuje entitu**.
 - **Vztah/vazba** – definován názvem a vztahem mezi **dvěma entitními typy**.
 - **Kardinalita vztahu** – dělení vztahů podle počtu entit vstupujících do vztahu – 1:1, 1:N, M:N.
 - **Povinnost v členství** – musí li vztah mezi dvěma entitami existovat, či nemusí.

33.2 Datová analýza a konceptuální model

Datová analýza zkoumá objekty reálného světa, jejich vlastnosti a vztahy. Zabývá se strukturou obsahové části systému (strukturou databaze). Výsledkem datové analýzy je konceptuální model. V rámci datové analýzy zpracováváme zadání (specifikaci požadavků na IS):

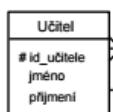
- podtrhneme **podstatná jména** = identifikujeme **objekty**,
 - podtrhneme **slovesa** = identifikujeme **vazby** mezi objekty,
 - najdeme **vlastnosti** a **stavy** nalezených objektu = identifikujeme **atributy**.

Z takto získaných informací sestavíme konceptuální model. **Konceptuální model** je jednoduchý popis entit a jejich vzájemných vztahů. Jedná se o jakýsi prvotní jednoduchý návrh námi vytvářené databáze. Je kladen důraz na zobrazení všech entit, jejich vztahů a je **nezávislý** na SŘBD. Skládá z:

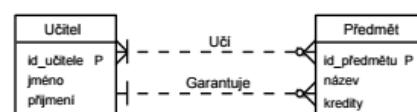
- **ER Diagram**, lineární zápis **entit**, lineární zápis **vztahů**, **datový slovník**, popis dalších IO (**integritních omezení**).

33.2.1 ER (Entity-Relationship) Diagram

Grafické znázornění **konceptuálního modelu** (objektů a vztahů mezi nimi). Může mít několik podob v závislosti na používaném prostředí a detailnosti s jakou jej potřebujeme vypracovat. **Atributy** můžou být v grafu znázorněny **ovály** spojenými s **objekty (obdélníky)**, **vazba 1:N** může být znázorněna „hráběmi“ místo N, či celý diagram se může podobat třídnímu diagramu s atributy vepsanými do objektu.



Crow's foot notace - Oracle



Crow's foot notace - Toad

Slabý entitní typ je označení entity, která **není jednoznačně definována jen svými atributy**, ale i jinou entitou, se kterou je ve vztahu. Tj bez tohoto vztahu by nedávala smysl. Nejčastěji implementováno pomocí složeného primárního klíče ve slabém entitním typu, kde jeden atribut PK je rovněž cizí klíč druhé entity. V příkladu je to položka objednávky, která bez znalostí informací o objednávce nemá žádnou vypovídací schopnost.



Oracle crow's foot notace pro slabý entitní typ

33.2.2 Lineární zápis entit a vztahů

Lineárním zápisem **popisujeme objekty**, jejich vlastnosti a vztahy **z pohledu implementačního**. Lineárním zápisem entit jsou v podstatě definovány **tabulky a jejich atributy** včetně **primárních a cizích klíčů**.

- Příklad lineárního zápisu entity: **Pes (IDPes, jmeno, pohlavi, vek, CRasa, IDUtulek)**.
- Příklad lineárního zápisu vztahů: **NABIZI (Útulek, Pes) 1:N**.

33.2.3 Datový slovník

Podrobný rozpis jednotlivých atributů. Tabulka obsahuje typ atributů, velikost, integritní omezení, atd.

Integritní omezení obsahují další specifikace atributů, které nejsou dány typem a délkou. Nejčastěji se týkají formátu atributu (podmínka v jakém má být formátu) – např: login se skládá z třech čísel a třech písmen, nebo rodné číslo je složeno z data narození, apod.

Další integritní omezení – konceptuální schéma obsahuje také soupis dalších IO, které se týkají entit (tabulek) a vazeb mezi nimi. Může jít například o omezení vícenásobné vazby, vyjádření hierarchie mezi entitama, apod.

Pes	Typ	Délka	Klíč	NOT NULL	IO
IDPes	int	8	primarni	ano	pravidla pro tvar čípového čísla
jmeno	varchar	50			
rokNarozeni	int	4			Validní rok
CRasa	int	2	sekundarni		

Tabulka 33.1: Datový slovník pro tabulkou Pes.

33.3 Funkční analýza

Zatímco datová analýza se zabývá strukturou obsahové části systému (strukturou databaze), **funkční analýza** řeší funkce systému. Funkční analýza tedy vyhodnocuje manipulaci s daty v systému. Skrze **DFD** (Data Flow Diagramy) analyzuje **toky dat, základní funkce systému a aktéry**, kteří se systémem pracují. Výstupem jsou pak **minispecifikace** – podrobné analýzy elementárních funkcí systému.

Cílem je popsat vytvářený systém jako „černou skříňku“, definovat její **vnější chování** a strukturalizovat **okolí systému**, které se systémem komunikuje. **Popsat všechny funkce**, které se budou s daty provádět.

Otázky na požadavky

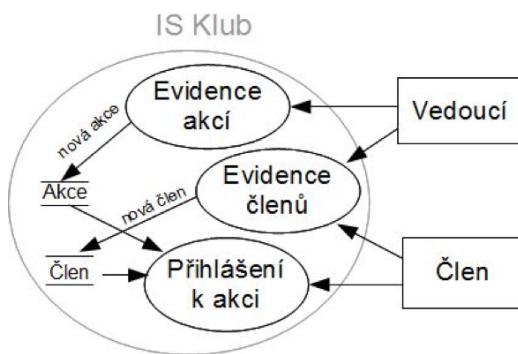
- **PROČ** nový systém.
- **ČEMU** má sloužit.
- **KDO** s ním pracuje - běžně, příležitostně, pravidelně zřídka.
- **VSTUPY** – objekty, atributy
- **VÝSTUPY** – výstupní sestavy, požadované informace
- **FUNKCE** – jaké výpočty, odvozování, výběry, třídění, ...
- **Vazby na OKOLÍ systému** – odkud data a kam.

Nefunkční požadavky

- Požadavky na **výsledný program**.
- **Vnější požadavky**: ostatní nefunkční implementační požadavky, použití **standardů**, **cenová omezení**, **časové požadavky**.

33.3.1 Diagram datových toků (DFD)

DFD je grafický nástroj pro **modelování funkcí a vztahů v systému**. Znázorňuje nejen procesy (funkce) a datové toky, ke kterým v systému dochází, ale definuje také hlavní aktéry a jejich omezení nad systémem. DFD diagram obsahuje tyto prvky: **aktér** (obdélník mimo systém), **proces** (kruh uvnitř systému), **datové toky** (šipky) a **paměť** (viz. obr. Akce a Člen).



DFD diagramy lze zakreslit v různých úrovních. Např. proces Evidence akcí na obrázku lze dále rozkreslit dalším DFD, obsahující procesy vytvoření a editace akce. DFD nejvyšší úrovně se nazývá **kontextový diagram**. Znázorňuje pouze práci aktérů se systémem jako celkem. Systém v kontextovém diagramu vystupuje jako černá skříňka a v diagramu tedy nejsou použity prvky procesu a paměti. **Hlavní znaky DFD**:

- Má několik úrovní podrobnosti.
- Definuje hranici systému.
- Definuje **všechny akce**, které mezi systémem a jeho okolím probíhají.

33.3.2 Minispecifikace = algoritmy elementárních funkcí

- Popisuje logiku každé z funkcí **poslední úrovně DFD**.
- Každému **elementárnímu (nerozložitelnému) procesu** z poslední úrovně DFD odpovídá **jedna minispecifikace**.
- Popisuje postup, jak jsou **vstupní data transformována na výstupní**.
- Popisuje, co **funkce znamená**, ne jak se to spočítá.

- Používá se **přirozený jazyk** s omezeným množstvím jasně definovaných pojmu, aby byla **srozumitelná** jak pro analytika, tak i uživatele a programátorovi.

IF všechny výrobky v objednávce jsou rezervovány,
THEN pošli objednávku k dalšímu zpracování oddělení prodeje.
OTHERWISE,

FOR EVERY nezarezervovaný výrobek v objednávce **DO**:

Zkus najít volný výrobek a rezervuj ho.

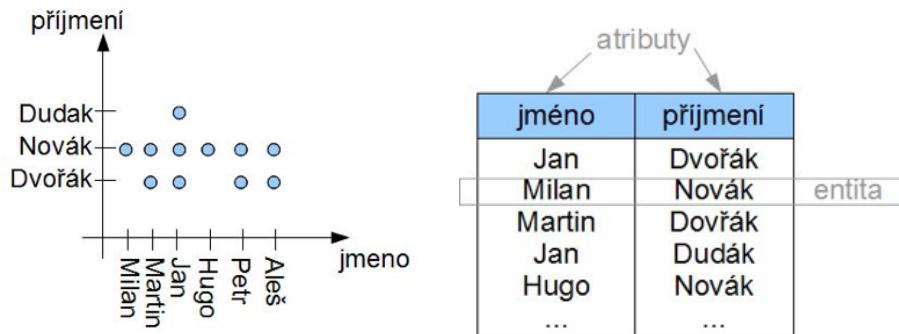
IF výrobek není na skladě,

THEN informuj správce.

34 Relační datový model, SQL; funkční závislosti, dekompozice a normální formy.

34.1 Relační datový model

Relační datový model představuje **způsob uchování dat v tabulkách**. Relační se mu říká proto, jelikož tabulka je definována přes Relaci.



Obrázek 34.1: Tabulka s dvěma atributy jako relace (vlevo), relace zobrazena tabulkou (vpravo).

Relace je tabulka definována jako **podmnožina kartézského součinu domén**. Relace na obrázku je tedy podmnožina kartézského součinu množin $\{\text{Dudak, Novák, Dvořák}\} \times \{\text{Milan, Martin, Jan, ..., Aleš}\}$.

Na rozdíl od matematické relace se ta databázová **mění v čase** (přidáváním a odebíráním prvků relace). Kromě základních **množinových operací** se u databázové relace setkáme s operací **selekcí** – výběr řádků a **projekcí** – výběr sloupců.

- **Doména** je **množina všech hodnot, kterých může daný atribut nabývat** (obor hodnot atributu). V praxi je doména dáná **integritním omezením** (IO). Doména atributu Příjmení z obrázků je množina $\{\text{Dudak, Novák, Dvořák}\}$.
- **Atribut** je vlastnost entity (z pohledu tabulky jde o sloupec).
- **Relační schéma** můžeme chápat jako strukturu tabulky (atributy a domény). Relační schéma R je výraz tvaru $R(A, f)$, kde R je jméno schématu, $A = A_1, A_2, \dots, A_n$ je **konečná množina jmen atributů**, f je zobrazení přiřazující každému jménu atributu A_i neprázdnou množinu (obor hodnot atributu), kterou nazýváme **doménou atributu** D_i , tedy $f(A_i) = D_i$.

Příklad pro tabulku (relaci) Učitel

- **Atributy:** ID, jméno, příjmení, funkce, kancelář.
- **Domény:**
 - D1 – tři písmena z příjmení, tři cifry pořadového čísla,
 - D2 – kalendář jmen,
 - D3 – množina příjmení,
 - D4 – množina funkcí (asistent, vědec, učitel,...),
 - D5 – A101, A102, ... A160.
- **Relační schéma:** Učitel (ID, jméno, příjmení, funkce, kancelář).
- **Relace:** Učitel = $\{\text{(nov001, lukas, novak, vědec, A135), (kom123, jan, komensky, učitel, A111), ...}\}$

34.1.1 Základní úlohy relačního modelu:

1. Návrh „správné“ struktury databáze bez redundancí – funkční závislosti, normální formy.
2. Vyhledávání informací z databáze – (dotazovací) relační jazyky.

34.1.2 Vlastnosti relačního datového modelu

Z definice relace vyplývají tyto jejich tabulkové vlastnosti:

- **Homogenita** (stejnorodost) sloupců (prvky domény).
- Každý údaj (hodnota atributu ve sloupci) je **atomickou položkou**.
- Na **pořadí** řádků a sloupců **nezáleží** (jsou to množiny prvků/atributů).
- Každý řádek tabulky je **jednoznačně identifikovatelný** hodnotami jednoho nebo několika atributů (primárního klíče).

34.1.3 Vazby relačního modelu

Obecně se vazby v relačním modelu realizují pomocí další relace (tabulky). Jedná se o tzv. **vazební tabulku**. Ta obsahuje ty atributy relací (tabulek, které se vazby účastní), které jednoznačně identifikují jejich entity – primární klíče. Obsahuje-li tabulka atribut, který slouží jako primární klíč v jiné tabulce, pak obsahuje cizí klíč. Vazební tabulka tedy obsahuje cizí klíče. Příklad vazby M:N:

Učitel			Učí		Předmět	
idu	jméno	příjmení	idu	cp	cp	název
dvo01	Jan	Dvořák	dvo01	3	1	matematika
kov01	Marie	Kovářová	dvo01	1	2	anglický j.
kov02	Martin	Kovádlna	kov01	2	3	fyzika
chy01	Jana	Chtrá	chy01	1	4	biologie
mal01	Libuše	Malinová	mal01	5	5	český j.

34.2 SQL (Structured Query Language)

SQL (Structured Query Language) je **relační jazyk založen na predikátovém kalkulu**. Na rozdíl od jazyků založených na relační algebře, kde se dotaz zadává algoritmem, tyto jazyky se soustředí na to **co se má hledat**, ne jak.

- Standardizovaný **strukturovaný dotazovací jazyk**, který je používán pro práci s daty v **relačních databázích**. (DQL - Data Query Language).
- Navržen IBM jako **dotazovací jazyk** (původní název Sequel).
- Základem je **n-ticový relační kalkul**.
- Standardy podporuje prakticky každá relační databáze, ale obvykle nejsou implementovány vždy **všechny požadavky normy**.
- Obsahuje i příkazy pro **vytvoření** a **modifikace** tabulek, pro **ukládání**, **modifikaci** a **rušení** dat v databázi a řadu dalších příkazů.
- **Příklad:** CREATE TABLE Drazitel (jmeno CHAR(20), adresa CHAR(30), aukce NUMBER(4), zisk NUMBER(4)); INSERT INTO clovek VALUES('nj001', 'Jan', 'Novotný', '777111222'); SELECT telefon FROM clovek WHERE příjmení = "Novotný";

34.2.1 DML - Data manipulation language

- **Modifikací** dat – INSERT, UPDATE, DELETE = vlož, uprav, smaž.
- **Vyhledávání** v relacích – SELECT, ORDER BY, GROUP BY, JOIN = vyhledej, seřaď, shlukuj, spoj.
- Další, pro podmínky, logické operátory, ... (WHERE, LIKE, BETWEEN, IN, IS NULL, DISTINCT/UNIQUE, JOIN, INNER JOIN, OUTER JOIN, EXISTS, HAVING, COUNT, VIEW, INDEX, ...).

34.2.2 DDL - Data definition language

- **Vytváření** a **modifikace** relačního schématu (tabulek, databází) - CREATE, ALTER (MODIFY, ADD), DROP = vytvoř, uprav, smaž.

34.2.3 DCL - Data control language

- **Správa práv** - příkazy jako GRANT, REVOKE.

34.2.4 TCL - Transaction control language - transakce

- COMMIT - úspěšně provedená transakce, vše se uloží.
- ROLLBACK - zrušení všech změn celé transakce.
- SAVEPOINT - uložení bodu, ke kterému lze provést ROLLBACK. Tj zruší se jen část transakce a může se pokračovat jinou větví celé transakce dále. Lze tak transakce dělit na menší atomické části.

34.3 Relační jazyky

Jazyky pro formulaci požadavků na výběr dat z relační databáze (dotazovací jazyky) se dělí do dvou skupin:

- **Jazyky založené na relační algebře**, kde jsou výběrové požadavky vyjádřeny jako posloupnost speciálních operací prováděných nad daty. Dotaz je tedy **zadán algoritmem**, jak vyhledat požadované informace.
- **Jazyky založené na predikátovém kalkulu**, které požadavky na výběr zadávají jako predikát charakterizující **vybranou relaci**. Je úlohou překladače jazyka nalézt odpovídající algoritmus. Tyto jazyky se dále dělí na
 - **n-ticové** relační kalkuly,
 - **doménové** relační kalkuly.

34.4 Relační algebra

Relační algebra je velmi silný **dotazovací jazyk** vysoké úrovně. Nepracuje s jednotlivými enticemi relací, ale s **celými relacemi**. Operátory relační algebry se aplikují na relace, výsledkem jsou opět relace. Protože relace jsou množiny, přirozenými prostředky pro manipulaci s relacemi budou množinové operace.

I když relační algebra v této podobě **není vždy implementována v jazycích SŘBD**, je její zvládnutí nutnou podmínkou pro správnost manipulací s relacemi. I složitější dotazy jazyka SQL, který je deskriptivním dotazovacím jazykem, mohou být bez zkušeností s relační algebrou problematické.

34.4.1 Základní operace relační algebry

Jsou dány relace **R** a **S**. **Množinové operace**:

- **Sjednocení** relací téhož stupně: $R \cup S = \{x|x \in R \vee x \in S\}$
- **Průnik** relací: $R \cap S = \{x|x \in R \wedge x \in S\}$
- **Rozdíl** relací: $R - S = \{x|x \in R \wedge x \notin S\}$
- **Kartézský součin** relace R stupně m a relace S stupně n: $R \times S = \{rs|r \in R \wedge s \in S\}$, kde $rs = \{r_1, \dots, r_m, s_1, \dots, s_n\}$

Další relační operace:

- **Projekce** (výběr atributů) relace R, jedná se o unární operaci $\Pi_X(R)$, kde X je množina názvů atributů.
- **Selekce** (výběr řádků) z relace R podle podmínky P. Selekcí je unární relační operace $\sigma_{\varphi(X)}(R)$, kde R je relace, $\varphi(X)$ predikátová formule hovořící o jednotlivých prvcích a jejich příslušnosti do relací.
- **Spojení** relací R s atributy A a S s atributy B (join). Značí se $R \bowtie S$, výsledkem je množina všech kombinací prvků relace **R** a **S**. Takto definovaný join se nazývá Přirozené spojení (natural join). Existují i další (outer, inner, left, right ...).

Příklad: $\Pi_{\text{název}} \sigma_{\varphi(\text{pohlaví}=\text{žena})}(\text{Úkol} \bowtie \text{Pracuje} \bowtie \text{Zaměstnanec})$

34.5 N-ticový relační kalkul

- Dr. Codd definoval n-ticový relační kalkul pro RDM jazyk matematické logiky - predikátový počet je využit pro výběr informací z relační databáze.
- Název odvozen z oboru hodnot jeho proměnných - **relace je množina prvků = n-tic**.
- Je **základem pro jazyk typu SQL**.

- Syntaxe je **přizpůsobena** programovacímu jazyku: **matematické vyjádření** $\{x|F(x)\}$ nahradíme zápisem $x \text{ WHERE } F(x)$
 - Kde x je proměnná pro hledané n-tice (struktura relace).
 - $F(x)$ je **podmínka**, kterou má x splňovat (výběr prvků relace).

34.5.1 Definice

Výraz n-ticového relačního kalkulu je výraz tvaru $x \text{ WHERE } F(x)$, kde x je jediná volná proměnná ve formuli F . Základní operace relační algebry se dají vyjádřit pomocí výrazů n-ticového relačního kalkulu, tedy n-ticový relační kalkul je relačně úplný.

Platí:	$R \cup S$	$\Rightarrow x \text{ WHERE } R(x) \text{ OR } S(x)$
	$R \cap S$	$\Rightarrow x \text{ WHERE } R(x) \text{ AND } S(x)$
	$R - S$	$\Rightarrow x \text{ WHERE } R(x) \text{ AND NOT } S(x)$
	$R \times S$	$\Rightarrow x, y \text{ WHERE } R(x) \text{ AND } S(y)$
	$R[a_1, a_2, \dots, a_k]$	$\Rightarrow x.a_1, x.a_2, \dots, x.a_k \text{ WHERE } R(x)$
	$R(P)$	$\Rightarrow x \text{ WHERE } R(x) \text{ AND } P$
	$R[A^*B]S$	$\Rightarrow x, y \text{ WHERE } R(x) \text{ AND } S(y) \text{ AND } x.A^*y.B$

34.6 Funkční závislost

Funkční závislost je v databázi **vztah mezi atributy** takový, že máme-li atribut Y je funkčně závislý na atributu X píšeme $X \rightarrow Y$, pak se **nemůže stát**, aby **dva řádky mající stejnou hodnotu atributu X měly různou hodnotu Y** . Je-li Y , X říkáme, že závislost $X \rightarrow Y$ je **triviální**.

- FZ je definována **mezi dvěma podmnožinami atributů** v rámci jednoho schématu relace. Jde o vztah mezi atributy, nikoliv mezi entitami.
- FZ je definována na **základě všech možných aktuálních relací**, není tedy možné soudit na funkční závislost z vlastností jediné relace. Tak můžeme poznat jen neplatnost funkční závislosti.
- FZ jsou **tvrzení o reálném světě**, o významu atributů nebo **vztahů mezi entitami**, je nutné realitu brát v úvahu při návrhu schématu databáze.

Příklad: Atribut 'datum narození' je funkčně závislý na atributu 'rodné číslo' (nemůže se stát, že u záznamů se stejnými rodnými čísly bude různé datum narození).

Pomocí funkčních závislostí můžeme **automaticky navrhnut schéma databáze** a předejít problémům jako je **redundance, nekonzistence databáze**, zablokování při vkládání záznamů, apod.

34.7 Armstrongovy axiomy

K určení **klíče schématu** a logických implikací množiny závislostí potřebujeme **nalézt uzávěr F^+** , nebo určit, zda daná závislost $X \rightarrow Y$ je prvkem F^+ . K tomu existují pravidla zvaná Armstrongovy axiomy. Jsou **úplná** (dovolují odvodit z dané množiny závislostí F všechny závislosti patřící do F^+) a **bezesporná** (dovolují z F odvodit pouze závislosti patřící do F^+).

- **Reflexivita** – je-li $Y \subset X \subset A$, pak $X \rightarrow Y$
- **Tranzitivita** – pokud je $X \rightarrow Y$ a $Y \rightarrow Z$, pak $X \rightarrow Z$
- **Pseudotranzitivita** – pokud je $X \rightarrow Y$ a $WY \rightarrow Z$, pak $XW \rightarrow Z$
- **Sjednocení** – pokud je $X \rightarrow Y$ a $X \rightarrow Z$, pak $X \rightarrow YZ$
- **Dekompozice** – pokud je $X \rightarrow YZ$, pak $X \rightarrow Y$ a $X \rightarrow Z$
- **Rozšíření** – pokud je $X \rightarrow Y$ a $Z \subset A$, pak $XZ \rightarrow YZ$
- **Zúžení** – pokud je $X \rightarrow Y$ a $Z \subset Y$, pak $X \rightarrow Z$

Závislost, která má na pravé straně pouze jeden atribut, nazýváme **elementární**.

34.7.1 Určení klíče pomocí funkčních závislostí

Ze zadání jsme určili atributy $A = \{\text{učitel}, \text{jméno}, \text{příjmení}, \text{email}, \text{předmět}, \text{název}, \text{kredity}, \text{místo}, \text{čas}\}$ a funkční závislosti F :

- učitel \rightarrow jméno, příjmení, email
- předmět \rightarrow název, kredity
- místo, čas \rightarrow učitel, předmět

Rozšíření:

- učitel, **místo, čas** \rightarrow jméno, příjmení, email, **místo, čas**
- předmět \rightarrow název, kredity
- místo, čas \rightarrow učitel, předmět

Dekompozice 1:

- učitel, **místo, čas** \rightarrow jméno, příjmení, email, místo, čas, **učitel, předmět**
- předmět \rightarrow název, kredity

Dekompozice 2:

- učitel, místo, čas \rightarrow jméno, příjmení, email, místo, čas, učitel, předmět, **název, kredity**

Atributy **učitel, místo, čas** je klíč schématu velké relace. V dalším kroku je třeba provést dekompozici a tuto velkou relaci rozbit na menší relace.

34.8 Dekompozice

Dekompozice relačního schématu je **rozklad relačního schématu na menší relač. sch.** (rozloží velkou tabulkou na menší) aniž by došlo k narušení redundance databáze. Mezi základní vlastnosti dekompozice patří - **zachování informace a zachování funkčních závislostí**.

- **Algoritmus dekompozice (metoda shora dolů)** – na počátku máme celé relační schéma se všemi atributy, snažíme se od tohoto schématu odebírat funkční závislosti a tvořit schémata nová. **Exponenciální složitost, BCNF**.
- **Algoritmus syntézy (zdola nahoru)** – vytvoří pro každou funkční závislost novou relaci. Pak tyto malé relace spojuje do větších celků. **Menší složitost, 3NF**.

Binární dekompozice, kterou budeme dále řešit je rozklad jednoho relačního schématu na dvě. Obecná dekompozice vznikne postupnou aplikací binárních. Dekompozice relačního schématu $R(A,f)$ je množina relačních $RO=\{R_1(A_1, f_2), R_2(A_2, f_2), \dots\}$, kde $A = A_1 \cup A_2 \cup A_3 \cup \dots$

34.9 Normální formy

Normální formy relací (NF) prozrazují jak dobře je databáze navržena (čím vyšší NF tím lepší).

- **0 NF** – Pokud nesplňuje ani 1 NF, je v 0 NF
- **1 NF** – definuje tabulky, které obsahují **pouze atomické atributy**. Žádné složené atributy - např. v jednom atributu je Jméno i Příjmení.
- **2 NF** – je v 1NF + **každý sekundární atribut je úplně závislý na každém klíči schématu**. Neboli neexistuje závislost sekundárních na podklíči (pokud se klíč skládá z více atributů). Např.: když $AB \rightarrow CD$, pak nesmí být $B \rightarrow C$. Atribut adresa není závislý na všech klíčích FZ, ale pouze na F.

2NF

firma	adresa	zboží	cena
F1	A1	Z010	100
F1	A1	Z020	50
F2	A2	Z020	80

- **3 NF** – je 2NF + žádný sekundární atribut **není tranzitivně závislý** na žádném klíči schématu. Nesmí existovat závislosti mezi sekundárními atributy (Model auta \rightarrow značka auta). Když $AB \rightarrow CD$, pak nesmí $C \rightarrow D$. **Příklad porušení 3NF** – atribut počet obyvatel je tranzitivně závislý (přes atr. město) na klíči.

firma	město	obyvatel
F1	M1	100 000
F2	M1	100 000
F3	M2	8 000

- **BCNF** (Boyce-Coddova normální forma) – 3NF + je-li funkční závislost $(X \rightarrow Y) \in F^+$ a $Y \notin X$, pak X obsahuje klíč schématu. **Musí být závislost sekundárních atributů na primárních nikoli naopak.** Když $AB \rightarrow CD$, pak nesmí $C \rightarrow A$.

Často pokud je splněna 3NF je zároveň splněna i BCNF. Pro nesplnění BCNF je nutné: Aby relace měla více kandidátních klíčů, alespoň 2 z nich musí být složené z více atributů a některé složené klíče musí mít společný atribut.

Příklad relace: PSČ, město, ulice. Toto je validní dle 3NF, ale ne BCNF. Kandidátní klíče jsou tedy PSČ-město a město-ulice. Město je v obou, překrývá se, tudíž není BCNF ale jen 3NF.

35 Transakce, zotavení, log, ACID, operace COMMIT a ROLLBACK; problémy souběhu, řízení souběhu: zamykání, úroveň izolacev SQL.

35.1 Transakce

Logická (nedělitelná, atomická) jednotka práce s databází, která musí proběhnout buď celá, nebo (v případě že je přerušena) obnovit původní stav databáze a spustit se znovu. Začíná operací **BEGIN TRANSACTION** a končí provedením operací **COMMIT** nebo **ROLLBACK**.

- Obecně zahrnuje posloupnost operací.
- Jejím úkolem je převést **korektní stav databáze** na jiný korektní stav.
- O řízení se stará **manager transakcí** nebo **monitor transakčního zpracování**.
- Operace transakce jsou nejprve zaznamenávány do **logu**.
- Transakce nemohou být vnořovány.
- Všechny SQL příkazy v transakci jsou atomické.
- Nepoužití transakcí může dojít k nekonzistenci databáze.

35.1.1 COMMIT

- Transakce doběhla úspěšně a změny mohou být **trvale uloženy**, zámky a adresace uvolněny (kromě WITH HOLD).
- Zavádí **potvrzovací bod**.
- Odpovídá úspěšnému ukončení logické jednotky práce a označuje **korektní stav DB**.

35.1.2 ROLLBACK

- Označuje, že databáze může být v **nekorektním stavu** a všechny změny transakce musí být **zrušeny**.

35.1.3 SAVEPOINT

- Rozdělení transakcí na menší části.
- ROLLBACK lze provést pouze částečně, pouze do předem vytvořeného SAVEPOINTu, co bylo před zůstane zachováno. Tím není zrušená celá transakce, ale může klidně pokračovat i nadále dalšími SQL příkazy až do závěrečného COMMITu.
- Po ukončení transakce je savepoint zahozen.

35.1.4 ACID

Každá transakce by měla splňovat následující vlastnosti:

- **Atomičnost (Atomicity)** – transakce musí být atomická: jsou provedeny všechny operace transakce nebo žádná.
- **Korektnost (Correctness)** – transakce převádí korektní stav databáze do jiného korektního stavu databáze, mezi začátkem a koncem transakce nemusí být databáze v korektním stavu.
- **Izolovanost (Isolation)** – transakce jsou navzájem izolovány: změny provedené jednou transakcí jsou pro ostatní transakce viditelné až po provedení COMMIT.
- **Trvalost (Durability)** – jakmile je transakce potvrzena, změny v databázi se stávají trvalými i po případném pádu systému.

35.2 Zotavení

- Nastává po **chybě SŘBD** => Zotavení databáze z nějaké chyby.
- Výsledkem musí být **korektní stav DB**.
- Využívají se **skryté redundantní** informace.
- Jednotkou zotavení je **transakce**.
- Všechny změny jsou zapisovány do logu před zápisem změn do DB => **pravidlo dopředného zápisu do logu**.
- Do logu se zapisuje **sekvenčně**, proto poskytuje **vyšší výkon** než přímý zápis dat.

35.2.1 Chyby zotavení

- **Lokální** - pouze v rámci jedné transakce (chyba v dotazu, přetečení hodnoty atributu). Vyskytuje se **10-100x/min** a čas pro zotavení je shodný, jako čas provedení transakce.
- **Globální** - ovlivňuje více transakcí najednou:
 - **Systémové (soft crash)** (výpadek proudu, pád systému). Může se vyskytovat i několikrát do roka. Čas potřebný k obnově je několik minut.
 - **Chyby média (hard crash)** - chyba disku (zotavení probíhá ze záložní kopie a z logu jsou obnoveny potvrzené transakce po vytvoření zálohy). Vyskytuje se zřídka a obnova může trvat i hodiny.

35.2.2 Průběh zotavení

Základním problémem vzniklým při systémové chybě je ztráta obsahu hlavní paměti, tedy ztráta obsahu vyrovnávací paměti SŘBD. Přesný stav transakce přerušené chybou není znám a transakce musí být **zrušena (UNDO)**. Někdy je transakce úspěšně dokončena, ovšem změny, nejsou přeneseny z vyrovnávací paměti na disk. V tomto případě musí být transakce po restartu systému přepracována (**REDO**). **Typy zotavení**:

Odloženou aktualizací (deferred update, NO-UNDO/REDO)

- Nepochází aktualizace databáze na disk dokud transakce nedosáhne **potvrzovacího bodu**. Všechny změny jsou v paměťovém bufferu.
- Jakmile transakce dosáhne potvrzovacího bodu, tak se **nejprve vše zapíše do REDO logu** a pak do DB (**pravidlo dopředného zápisu do logu**).
- Při selhání transakce není nutné provádět **undo**, změny jsou ztraceny spolu s vyrovnávací pamětí.
- **Redo** se provádí při chybě během zápisu do DB.
- Do logu jsou v případě odložené aktualizace zapsány nové hodnoty (kvůli REDO).
- Minimální I/O operace, používá se pouze pro **krátké a nenáročné transakce** - hrozí přetečení bufferu.

Okamžitou aktualizací (immediate update, UNDO/NO-REDO)

- **Provádí aktualizaci DB** než transakce dosáhne potvrzovacího bodu.
- Operace jsou zapsány do UNDO logu a zároveň je aktualizována DB (pravidlo dopředného zápisu do logu).
- Při chybě je nutné provést **undo**, protože **došlo k aktualizaci DB**.
- Do logu se zapisují **původní hodnoty**, což umožní systému provést UNDO.
- **Velká zátěž disku / nízký výkon**.

Kombinovanou aktualizací (UNDO/REDO)

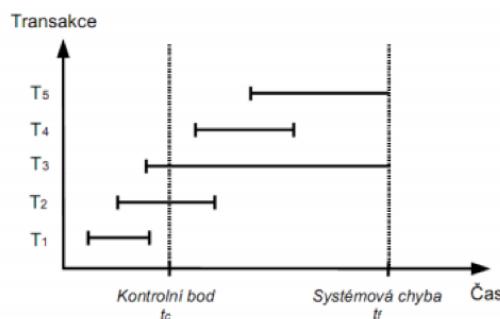
- Používaná v praxi. Využívá obou operací v kombinaci s technikou kontrolních bodů.
- Nezapisuje všechny potvrzené operace na disk, místo toho vytváří **kontrolní body**.
 - Zápis operací hromadně po určitém počtu záznamů.
 - Zapisuje se obsah vyrovnávací paměti na disk a záznam o kontrolním bodu do logu.
- Po restartu systému se provádí: **undo** na všechny transakce, které se **nestihly potvrdit** a **redo** na všechny transakce, které se **potvrdily** po vytvoření kontrolního bodu.

35.3 Kontrolní body

Kontrolní body **jsou vytvářeny např. po určitém počtu záznamů**, které byly zapsány do logu a zahrnují:

- zápis obsahu vyrovnávací paměti na disk,
- zápis záznamu o kontrolním bodu do logu.

V případě následující situace musí být:



- Po restartu systému musí být transakce typu T_3 a T_5 zrušeny (undo).
- Transakce typu T_2 a T_4 musí být přepracovány (redo).
- Jelikož změny provedené transakcí T_1 byly provedeny p kontrolním bodem tc, tuto transakci při zotavení vůbec neuvažujeme.

Postup zotavení systému:

1. Vytvoří se seznamy UNDO a REDO.
2. UNDO se naplní všemi neuloženými transakcemi (vše kromě T_1 na obrázku výše).
3. Procházíme všechny transakce - je v logu COMMIT pro danou transakci -> přesuň transakci do REDO.
4. Všechny transakce z UNDO jsou postupně zrušeny.
5. Všechny transakce z REDO jsou přepracovány a uloženy.
6. Nyní je systém použitelný

35.4 Problémy souběhu

- Pro víceuživatelský DB systém (kolik současně). Pro jednouživatelský přístup (SQLite) se toto vůbec neřeší.
- Souběh umožňuje SŘBD **zpřístupnit databázi mnoha transakcím ve stejném čase**.
- Souběh také přináší mnoho **problémů**, které je nutné řešit i na **aplikáční úrovni**.

35.4.1 Plán provádění transakce a anomálie

Plán provádění transakce = posloupnost operací transakce, při souběžném provedení - **plán souběžný/paralelní**. Vznikají **3 problémy**:

- **Problém ztráty aktualizace** – jedna transakce **přepíše právě prováděnou hodnotu**. Časová posloupnost: `read_A, read_B, write_A, write_B`.

Transakce A	Čas	Transakce B
READ t	t_1	-
-	t_2	READ t
WRITE t	t_3	-
-	t_4	WRITE t

Obrázek 9.1: Aktualizace transakce A je ztracena v čase t_4 .

- **Problém nepotvrzené závislosti**

- **Scénař 1 – tr. A pracuje se špatnými daty** - Transakce B zapíše X, transakce A přečte X, transakce B provede ROLLBACK.
- **Scénař 2 – změna tr. A je ztracena** - Transakce B zapíše X, transakce A zapíše X, transakce B provede ROLLBACK.

Transakce A	Čas	Transakce B
-	t_1	WRITE t
READ t	t_2	-
-	t_3	ROLLBACK

Obrázek 9.2: Transakce A se stala v čase t_2 závislou na nepotvrzené změně transakce B.

- **Problém nekonzistentní analýzy** – A provádí součet na účtech, před dokončením B provede přesun z účtu na účet, přičemž 1 už byl započítán a druhý ne. **Špatný součet zůstatků!** A čte committed data (B provede commit, než si A vyžádá další účet), ale i tak to není správné.

Transakce A	Čas	Transakce B
READ acc_1	t_1	-
$suma = 30$		
READ acc_2	t_2	-
$suma = 50$		
-	t_3	READ acc_3
-	t_4	WRITE $acc_3 = 60$
-	t_5	READ acc_1
-	t_6	WRITE $acc_1 = 20$
-	t_7	COMMIT
READ acc_3	t_8	-
$suma = 110$ ne 100		

Obrázek 9.4: Transakce A provedla nekonzistentní analýzu.

35.5 Konflikty čtení/zápis

A a B chtějí číst/zapisovat stejnou entaci (záznam). Nastávají 4 možnosti konfliktu:

- **RR (READ-READ)** – negativně se neovlivní, není problém.
- **RW (READ-WRITE)** – `read_A, write_B` = A dále počítá s daty → RW zapříčinuje **problém nekonzistentní analýzy**. `read_A, write_B, read_A` → A načte odlišené hodnoty = **neopakovatelné čtení (non repeatable read)**
- **WR (WRITE-READ)** – `write_A, read_B, rollback_A?` → **Problém nepotvrzené závislosti**. Pokud B přečte data → **Špinavé čtení (dirty read)** = čtení non-commited dat.
- **WW (WRITE-WRITE)** – `write_A, write_B, rollback_A?` → **Ztráta aktualizace** (pro A) a **nepotvrzená závislost** pro B. **Špinavý zápis (dirty write)** - přepisování non-commited dat.

35.6 Techniky řízení souběhu

35.6.1 Správa verzí - optimistický přístup

Předpoklad, že se paralelní **transakce ovlivňovat nebudou**. Systém **vytváří** při aktualizaci **kopie dat** a sleduje, která z verzí má být viditelná pro ostatní transakce (podle úrovně izolace).

35.6.2 Zamykání - pesimistický přístup

Předpokládáme, že se paralelní **transakce budou ovlivňovat**. Systém spravuje jednu kopii dat a jednotlivým transakcím přiděluje **zámkы**. Používá se nejčastěji.

- Chce-li transakce A provést čtení/zápis nějakého objektu v DB (nejčastěji n-tice), **požádá o zámek** na tento objekt. Žádná jiná paralelní transakce zámek získat nemůže, dokud jej A **neuvolní**.
- **2 typy zámků (existuje jich i více):**
 - **výlučný** zámek (exclusive lock / write lock) **X**.
 - **sdílený** zámek (shared lock / read lock) **S**.
- A má zámek X a B **nedostane žádný zámek** hned. A má zámek S, B **může hned dostat S, X** nikoliv.
- **Matice kompatibility** - vzájemné vztahy typů zámků, sloupce a řádky: X, S, -, A (okamžitě), N (ne)

	X	S	-
X	N	N	A
S	N	A	A
-	A	A	A

- **Operace aktualizace** - mění obsah DB - UPDATE, INSERT i DELETE.
- **Uzamykací protokol** - většinou žádání zámků implicitně → při **získání** n-tice z DB žádán **zámek S**. Při aktualizaci **zámek X**; žádá-li zámek X a má už S, je mu **S změněn na X**; když nemůže být zámek přidělen okamžitě, transakce přechází do stavu **čekání** (wait state).
- Systém musí zajistit aby v tomto stavu nesetrvala navždy - situace **"livelock"**nebo **"starvation"** → řadit požadavky do **fronty** (FIFO). Zámky uvolněny až po operaci COMMIT nebo ROLLBACK.
- **Explicitní uzamykání** - LOCK TABLE <names> IN [ROW SHARE|ROW EXCLUSIVE|SHARE UPDATE|SHARE|SHARE ROW EXCLUSIVE|EXCLUSIVE] MODE [NOWAIT]

35.7 Uváznutí

- **Deadlock** - dvě nebo více transakcí jsou ve stavu **čekání** na uvolnění zámků držených jinou transakcí.
- **Detekce uváznutí - časové limity** (nastavení max. času pro vykonání transakce), **detekce cyklu v grafu Wait-For** (zaznamenává, které transakce na sebe čekají → u jedné provede ROLLBACK).
- **Prevence uváznutí pomocí časových razítek** - 2 verze uzamykacího protokolu. Každá transakce na začátku dostane časové razítko (unikátní). Pokud A požaduje zámek na entaci, která je zamčená B pak:
 - při **Wait-Die** - pokud je A starší než B, A přejde na čekání; je-li mladší, A je zrušena ROLLBACK a spuštěna znovu.
 - při **Wound-Die** - pokud A je mladší než B, A přejde na čekání; starší → B zrušena ROLLBACK a spuštěna znovu.
- Při opětovném spuštění si transakce nechá své časové razítko. **Nevýhodou** je velký počet operací ROLLBACK. První část jména - situace kdy A je starší než B. **Nemůže nikdy dojít k uváznutí**.

35.8 Sériový a serializovatelný plán

- **Ekvivalentní plán** - 2 plány jsou ekvivalentní, pokud dávají shodné výsledky.
- **Sériový plán** - n-tice uspořádaná dle **pořadí vykonávání** jednotlivých transakcí. (transakce jsou provedeny zasebou).
- **Serializovatelný plán** - **plán vykonávání dvou transakcí** je korektní jen tehdy, pokud je serializovatelný → plán ekvivalentní s výsledkem libovolného sériového plánu.

35.8.1 Dvoufázové uzamykání

Transakce které dodržují protokol dvoufázového uzamykání **jsou vždy serializovatelné**.

1. Transakce musí požádat o zámek, než začne pracovat s nějakou enticí.
2. Po uvolnění jakéhokoli zámku nesmí žádat jiný zámek. Všechny držené zámky musí uvolnit.

35.9 Úroveň izolace transakce

Serializovatelnost garantuje izolaci transakcí ve smyslu podmínky **ACID**. Je-li plán transakcí serializovalný, neprojeví se negativní vlivy souběhu. Za izolovanost transakcí se platí **menším výkonem** souběhu → **nižší propustností**. SŘBD umožňuje nastavit úroveň izolace - ta **sníží míru izolace** transakce a **zvýší propustnost**.

Pod pojmem špinavé čtení spadá i špinavý zápis. **Výskyt fantomů** nastane v případě:

- Zámek probíhá jen nad existujícími enticemi.
- Pokud provedeme `SELECT .. WHERE xxx BETWEEN 1 AND 10` – dostaneme zámek na všechny entice, které jsou v rozsahu.
- Při vložení nového záznamu s xxx mezi 1 a 10 nebo úpravě jiného kde za xxx bude dosazeno číslo mezi 1 a 10, se při opakováném čtení objeví fantom. Protože i přes zámek stále dostáváme jiné výsledky, protože dané nové/upravené entice zámek nemají, ale již spadají do podmínky výše.

Úroveň izolace	Špinavé čtení	Neopakovatelné čtení	Výskyt fantomů
READ UNCOMMITTED	Ano	Ano	Ano
READ COMMITTED	Ne	Ano	Ano
REPEATABLE READ	Ne	Ne	Ano
SERIALIZABLE	Ne	Ne	Ne

36 Procedurální rozšíření SQL, PL/SQL, T-SQL, triggers, funkce, procedury, kurzory, hromadné operace.

36.1 Procedurální rozšíření SQL

Kromě základních příkazu pro vytváření a modifikaci dat obsahuje SQL triggers, funkce, procedury, kurzory. To umožňuje přenést část aplikační logiky přímo do databází, čímž se ušetří hodně I/O operací při přenosu dat mezi systémem a DB. Je **závislé na SŘBD** a její různé implementace se mnohdy velice liší:

- **PL/SQL** pro Oracle
- **Transact-SQL** (T-SQL) pro Sybase a MSSQL
- **PL/pgSQL** pro PostgreSQL
- **SQL PL** pro DB2

36.2 PL/SQL

- Založeno na jazyku ADA.
- Kód uložen a **prováděn v SŘBD**, může být sdílen více aplikacemi.
- **Nezávislý na aplikační platformě** (pouze na SŘBD).

36.2.1 Proměnné, procedury a funkce

- **Proměnné**
 - Proměnné můžeme rozdělit do několika skupin, dle různých kritérií, nejčastějším dělením je podle **datového typu** na **číselné** (NUMBER), **stringové** (CHAR, VARCHAR2), **datumové** (DATE, TIMESTAMP).
 - Definujeme proměnné `part_no NUMBER(4); in_stock BOOLEAN;`
 - Přiřazení do proměnných je pomocí operátoru `:=`.
- **Anonymní procedury**
 - **Nepojmenované** procedury které **nejde volat**, jsou spuštěny a zahrozeny.
 - Mohou být uloženy v souboru nebo spuštěny přímo z konzole.
 - Jsou pomalejší než pojmenované procedury, protože **nemohou být překompilovány**.
- **Pojmenované procedury**
 - Obsahují **hlavičku se jménem a parametry**. Díky tomu se dají volat z jiných procedur či triggerů nebo spuštěny příkazem **EXECUTE**.
 - Jelikož jsou **kompilovány** jen jednou, jsou rychlejší než anonymní.
 - `CREATE [OR REPLACE] PROCEDURE jmeno_procedury [(jmeno_parametru [mod] datovy_typ , ...)] IS|AS definice lokálních proměnných BEGIN tělo procedury END [jmeno_procedury]`
- **Funkce**
 - Na rozdíl od procedury **vrací hodnotu**. Kromě standardních funkcí (TO.CHAR, TO.DATE, SUBSTR, apod.) si můžeme definovat **vlastní funkce**.
 - Lze následně používat v SQL dotazech jako je SELECT apod.
 - `CREATE [OR REPLACE] FUNCTION jmeno_funkce [(jmeno_parametru [mod] datovy_typ , ...)] RETURN navratovy_datovy_typ IS |AS definice lokálních proměnných BEGIN tělo procedury END [jmeno_procedury]`

36.2.2 Dynamické a statické PL/SQL

- **Statické PL/SQL** - klasické procedury, které mají vázané proměnné.
- **Dynamické PL/SQL** - kód SQL příkazu je vytvářen dynamicky za běhu - vytvoření textového řetězce a jeho spuštění příkazem **EXECUTE IMMEDIATE**.

36.2.3 Výjimky, podmínky, cykly

- **Výjimky**
 - Vznikají ručně i ze systému.
 - Zpracování v bloku **EXCEPTION**
 - Pro ruční vyvolání je nutné ji deklarovat (DECLARE vyjimka EXCEPTION;) a vyhodit (**RAISE** vyjimka)
- **Podmínky**
 - IF podminka1 THEN příkazy [ELSIF podminka2 THEN příkazy] [ELSE příkazy] END IF ;
- **Cykly**
 - **do while** - LOOP příkazy cyklu [EXIT; | EXIT WHEN podminka ;] END LOOP;
 - **while do** - WHILE podminka LOOP příkazy cyklu END LOOP;
 - **for** - FOR jmeno_promenne IN [REVERSE] value1 .. value2 LOOP příkazy cyklu END LOOP;

36.2.4 Kurzory, triggersy, hromadné operace

- **Triggersy**
 - Kód spouštěný v reakci na **událost** (DML, DDL, systémové eventy).
 - Lze určit kdy se má spouštět - BEFORE, AFTER, INSTEAD OF.
 - Při jaké události se má spustit - INSERT, UPDATE, DELETE - lze specifikovat i více akcí najednou.
 - V PL/SQL FOR EACH ROW - Tělo triggeru se bude volat pro každý řádek zvlášť, nikoli najednou.
 - Lze používat speciální proměnné obsahující starou a novou hodnotu :OLD, :NEW.
 - Pokud se pokusíme v triggeru číst či modifikovat stejnou tabulkou dostaneme **mutating table error**.
 - CREATE [OR REPLACE] TRIGGER jmeno_triggeru BEFORE | AFTER | INSTEAD OF INSERT [OR] | UPDATE [OR] | DELETE [OF jmeno_sloupce] ON jmeno_tabulky [REFERENCING OLD AS stara_hodnota NEW AS nova_hodnota] [FOR EACH ROW [WHEN (podminka)]] BEGIN příkazy END;.
- **Kurzory**
 - Kurzor je ukazatel na řádek **vícerádkového výběru**. Je třeba jej v programu deklarovat pokud budeme zpracovávat vícerádkové výběry. Kurzorem mohu pohybovat a tak se dostanu na další řádky výběru. Zdrojem kurzoru je vždy SQL dotaz a jsou dva typy:
 - * **implicitní** - vytváří se automaticky po provedení příkazu INSERT, UPDATE, DELETE.
 - * **explicitní** - **ručně vytvořený kurzor**. Vytváří se nejčastěji ve spojením s příkazem SELECT.
 - **Příkazy pro práci s kurzorem:**
 - * CURSOR kurzor IS select; - vytvoření kurzoru.
 - * OPEN kurzor - otevře kurzor, tedy nastaví ho na první řádek.
 - * FETCH kurzor INTO promena - příkaz pro pohyb kurzoru. Načte aktuální záznam do proměnné a posune se na další záznam.
 - * CLOSE kurzor - uzavře kurzor.
- **Vázané proměnné**
 - SŘBD kontroluje **jedinečnost dotazu**, pokud už byl dotaz v minulosti proveden, použije se **dříve použitý plán** dotazu místo nového vytváření plánu.
 - Vázané proměnné umožňují **parametrizaci hodnot v dotazu**, odpadá tedy opětovné vytváření plánu pro stejný dotaz s jinou hodnotou.
 - Lze používat i v dynamickém PL/SQL (pomocí **USING**).
 - Použití i při volání z aplikace (podpora v C# i Java).
- **Hromadné operace**
 - Snížení režie na zotavení (zápis do logu) a aktualizace DB (datových struktur). Výsledkem je rychlejší vkládání záznamů do DB.
 - Lze použít pro statické i dynamické SQL.

- **BULK COLLECT**
 - * **Hromadné načtení** (navázání vstupní kolekce s PL/SQL enginem).
 - * `BULK COLLECT INTO collection_name[, collection_name]`
- **FORALL**
 - * Hromadná **operace** (navázání vstupní kolekce před posláním do SQL enginu)
 - * `FORALL index IN lower_bound..upper_bound [INSERT, UPDATE nebo DELETE];`
- **Balíky**
 - Obdoba kníhoven v programovacích jazycích.
 - Specifikace balíku a následně tělo.

36.3 T-SQL (Transact-SQL)

Transact-SQL (T-SQL) je proprietární rozšíření jazyka SQL od společnosti **Microsoft** a **Sybase**, které Microsoft používá v produktu **Microsoft SQL Server**, Sybase Software pak v Adaptive Server Enterprise.

36.3.1 Proměnné, procedury a funkce

- **Proměnné**
 - Deklarace pomocí `DECLARE @TMP INT.`
 - Inicializace pomocí `SET` nebo `SELECT`.
- **Podmínky**
 - `IF <boolean condition> <statement> ELSE <statement>`
 - Více příkazů v jedné větvi musíme obalit do `BEGIN/END`.
- **Cykly**
 - `WHILE <Boolean expression> <code block>`
- **Transakce**
 - **Začátek** - `BEGIN TRANSACTION <nazev_transakce>`
 - **Konec** - `ROLLBACK` nebo `COMMIT`
 - **Nastavení úrovně izolace** - `SET TRANSACTION ISOLATION LEVEL <level>`
- **Výjimky**
 - Bloky `try/catch` (`BEGIN TRY/END TRY` a `BEGIN CATCH/END CATCH`).
- **Uložené procedury**
 - Uloženy v SŘBD.
 - Předkomplilovány, tudíž jsou rychlejší.
 - `CREATE PROC[EDURE] procedure_name [;number] [@parameter data_type [VARYING] [= default] [OUTPUT]] [, . . .] [WITH RECOMPILE | ENCRYPTION | RECOMPILE, ENCRYPTION] [FOR REPLICATION] AS sql_statement`
- **Uložené funkce**
 - Není možné použít `try/catch`, DML atd.
 - Musí vracet hodnotu.
 - `CREATE FUNCTION [schema_name.] function_name ([@parameter_name [AS] [type_schema_name.] parameter_data_type [= default] [READONLY] [,...n]]) RETURNS return_data_type [WITH <function_option> [,...n]] [AS] BEGIN function_body RETURN scalar_expression END [;]`

36.3.2 Kurzory, triggery, dynamické SQL

- **Kurzory**

- `DECLARE cursor_name CURSOR FOR select_statement`
- **Musíme použít** sekvenci příkazů: `OPEN`, `FETCH`, `CLOSE`, `DEALLOCATE`
- Pro testování prázdného kurzoru používáme `@@FETCH_STATUS`

- **Trigger**

- `CREATE TRIGGER [schema_name .] trigger_name ON table | view [WITH <dml_trigger_option> [,...n]] FOR | AFTER | INSTEAD OF [INSERT] [,] [UPDATE] [,] [DELETE] [WITH APPEND] [NOT FOR REPLICATION] AS sql_statement [;] [,...n] | EXTERNAL NAME <method specifier> [;] >`

- **Dynamické SQL**

- Podobné PL/SQL, pomocí příkazu `sp_executesql`
- `sp_executesql [@stmt =] stmt [, [@params =] N'@parameter_name data_type [,... n] , [@param1 =] 'value1 ' [,... n]]`

36.4 Rozdíly PL/SQL vs T-SQL

- **T-SQL neposkytuje operátory** jako `%TYPE` a `%ROWTYPE` pro získání datových typů existujících záznamů.
- **T-SQL nepodporuje** `CREATE OR REPLACE PROCEDURE`, což nás nutí k tomuto zápisu: `/*CREATE*/ ALTER PROCEDURE`.
- **T-SQL** podstatně **omezuje konstrukce**, které můžeme využívat ve funkcích.
- V **T-SQL** musím při práci s kurzory používat `OPEN`, `FETCH`, `CLOSE`, `DEALLOCATE`.
- **T-SQL** nás nutí k dvojitému `FETCH` u kurzorů.
- V **T-SQL** musíme definovat u parametrů procedur a funkcí **délku datového typu** (pokud se u datového typu udává).
- V **T-SQL** musíme více příkazů v jedné větvi obalit do `BEGIN/END`.

37 Základní fyzická implementace databázových systémů: tabulky a indexy; plán vykonávání dotazů.

37.1 Fyzická implementace

Fyzická implementace definuje **datové struktury** pro základní logické objekty:

- **tabulky**,
- **indexy**,
- **materializované pohledy** (materialized views),
- **rozdělení dat** (data partitioning) – data s dlouhou historií. Fyzické rozdělení datových struktur a souboru na více částí.

Fyzická implementace tedy **řeší uložení dat na nejnižší úrovni databáze**. Na úrovni databáze můžeme ovlivnit výběr efektivnějšího plánu vykonávání dotazu případně **dobu vykonávání operací**.

- **ROWID** – jedinečné číslo **označující záznam tabulky**.
- Větší část teorie fyzického návrhu DB je platná pro libovolné SŘBD, dále se musíme řídit **doporučením výrobce**.
- Každé SŘBD obsahuje specifickou reprezentaci tabulek a indexů:
 - **CREATE TABLE** - vytvoření tabulky typu **halda** (Oracle), **shlukování záznamů** (SQL Server 2012).
 - **CREATE INDEX** - vytvoření **B+-stromu**, kde každá položka odkazuje na ROWID záznamu v tabulce.

37.2 Datové struktury

Datové struktury **se skládají** buďto ze **stránek**, nebo z **uzlů** v případě stromové struktury. Jejich realizace přímo ovlivňuje efektivitu **operací** vyhledávání, vkládání, editace a mazání. Konkrétní realizace je závislá na použitém typu tabulky (halda, halda+index, shlukování záznamů).

Základní datové struktury:

- **Blok** (alokační jednotka) – je nejmenší jednotka, se kterou SŘBD manipuluje při zápisu a čtení dat z disku (obvykle 4KB nebo 8KB).
- **Stránka** – nejmenší jednotka s kterou pracuje správce paměti. **Stránka.velikost = X * Blok.velikost** (je-li velikost bloku = 4KB je odpovídá velikost stránky násobkům 4KB).
- **Datový soubor** – fyzický prostor na disku s daty.

37.3 Typy tabulek

37.3.1 Heap Table

- Implicitní pro **CREATE TABLE** téměř všude, pokud tabulka nemá žádné indexy (ani PK). Záznamy nejsou nijak uspořádány.
- **Stránkové perzistentní pole** s velikostí **bloku** nejčastěji **8kB**.
- Záznamy pouze **označovány jako smazané**, pro fyzické smazání slouží speciální operace **shrinking**.
- Obsahuje všechny záznamy a jejich atributy pokupě.
- Při vkládání záznam uložen na **první volnou pozici v tabulce** nebo na **konec pole**.
- Složitost operací: neefektivní vyhledávání $O(n)$, velmi **efektivní vkládání** $O(1)$ i **využití místa**.

37.3.2 Shlukování záznamů (Data Clustering)

- Implicitní pro MS-SQL při CREATE TABLE
- Záznamy v **datovém souboru jsou seřazeny podle zvoleného klíče**, pro implementaci nejčastěji využita nějaká **varianta B-stromu**.
- Listové uzly stromu (bloky) obsahují **kromě klíče i další atributy** tabulky.
- V pokročilejších DBMS lze zvolit, které atributy leží přímo v B+ stromu a které leží na haldě. Lze tak kombinovat shlukování s heap table. Pro získání dat uložených přímo v listu není přístup na haldu poté nutný
- Používá se všude, kde potřebujeme **získat i hodnoty ostatních atributů** (kromě klíče - např. SELECT neklíčových atributů) - vyšší výkon
- Zhoršený výkon** INSERT (data se musí zatřízovat).
- Oracle:** Index Organized Table (IOT), **SQL Server:** Clustered Index.

Eliminace náhodných přístupů přes ROWID

Tabulka se shlukováním záznamů může ušetřit náhodný přístup oproti heap table a indexu. Po vyhledání ve stromě není nutné další náhodný přístup na haldě, protože vše potřebné je v listu. Kvůli více datům v listech může ale být celková výška stromu ve shlukované tabulce mnohem větší. To způsobí více náhodných přístupů při průchodu stromem než v případě heap table a indexu. Co je lepší nelze určit předem ale pouze testem na reálných nebo pseudoreálných datech.

37.3.3 Hašovaná tabulka (Hash Table)

- Záznamy se stejnou hashovanou hodnou jsou uloženy ve stejném nebo velmi blízkém bloku.
- Trochu **plýtvá místem**.
- Musíme znát předem velikost záznamů (alespoň přibližně).

37.3.4 Materializované pohled (Materialized views)

- Uložené **výsledky dotazů**, které bývají často v DB vyhodnocovány.
- Jde spíše o **fragment z tabulky** či několika tabulek.

37.4 Indexy

Indexové soubory slouží k možnosti **rychlého vyhledávání a seřazení tabulky** podle různých atributů. Tabulka je jinak seřazena úplně náhodně (heap table), nebo podle primárního klíče (data-clustering). Index je tedy vázáný ke konkrétní tabulce a konkrétnímu atributu. Index umožňuje **rychlé vyhledávání** dle klíče, **ROWID pak odkazuje na kompletní záznam v heap tabulce**. CREATE [BITMAP] INDEX login ON Student;. Protože ale následný počet náhodných přístupů pomocí ROWID může být mnoho, někdy DB raději provede sekvenční prohledání celé tabulky. Rozhoduje se individuálně na základě statistik o atributech a jejich velikostech.

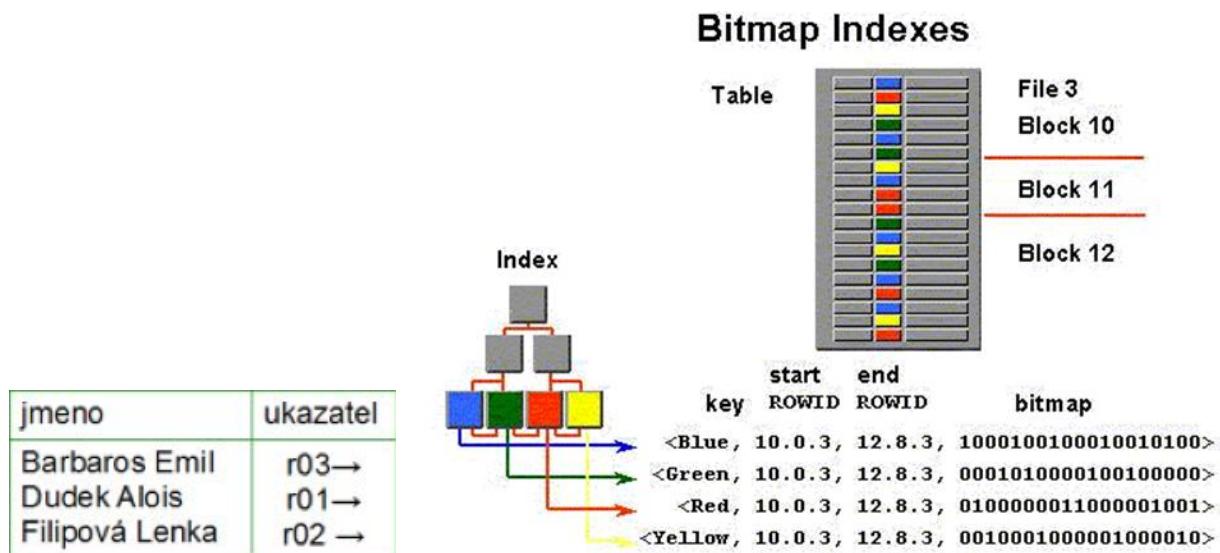
- Primární index (PRIMARY)** – automatický index, který se váže k primárnímu klíči, zajišťuje jedinečnost údajů.
- Unikátní index (UNIQUE)** – stejně jako primární index zajišťuje jedinečnost údajů v atributu, ale neváže se k primárnímu klíči.
- Vedlejší index (INDEX)** – klasické index popsány níže.
- Fulltextový index** – používá se pro optimalizaci fulltextového vyhledávání v daném sloupci.
- Složený index** – Může být kterýkoli výše. Obsahuje více atributů v jednom klíči. Pokud je dotaz jen na druhý či další atribut v indexu, nelze index použít.

37.4.1 Nevýhody indexů

- Při velkém množství různých indexů mohou soubory s indexem zabírat mnohem více než samotná tabulka
- Pro získání dalších dat je nutné hodně náhodných přístupů přes ROWID
- Každý záznam v indexu musí být shodné délku. Tj například stringy nejsou vhodné pro index, protože každý záznam v indexu bude tak dlouhý, jaký je nejdelší možný string daného atributu. Pokud je string kratší, je záznam v indexu vyplněn 0.

37.4.2 Bitmapový index

Odpovídá na všechny možné hodnoty daného atributu. Jde o tabulku jedniček a nul, kde řádky reprezentují záznamy v indexované tabulce a sloupce hodnoty indexovaného atributu. Jednička pak znamená true, že záznam má hodnotu danou sloupcem. **Vyplatí se**, pokud se používají logické operátory (AND, OR, XOR) nad několika bitmapovými indexy atributů.



37.4.3 Shlukovaný index (Cluster Index)

Pokud se pro dvě tabulky často používá operace spojení (JOIN) pro jeden atribut. V tomto případě diskový blok obsahuje záznam z řídicí tabulky a zároveň i závislé záznamy. (v normalním případě obsahuje diskový blok pouze záznamy jedné tabulky).

37.4.4 Kandidáti na index

- Primární klíče a cizí klíče.
- Pokud je index používán pro nalezení malého počtu záznamů.
- Pokud index pokryje jeden nebo více častých dotazů.
- Atributy často se vyskytující v konstrukci WHERE.

37.5 Vykonávání dotazu

Ovlivnění času vykonávání dotazu - parametrizované dotazy, hromadné operace, nastavení transakcí. Na úrovni DB můžeme ovlivnit výběr efektivnějšího plánu vykonávání dotazu, případně dobu vykonávání operací -> **fyzický návrh DB**. Identifikujeme 4 fáze vykonávání dotazu:

1. Parsování dorazu

- Převod původního dotazu do zvolené **interní formy**.
- **Eliminujeme syntaxi jazyka dotazu** (např. SQL).
- Zpracování pohledů, které probíhá v této fázi, znamená, že **nahradíme pohled jeho definicí** (materializované pohledy – fragmenty/výsledky dotazů).

- Interní forma je nejčastěji **nějaký druh dotazovacího stromu** (angl. query tree).

2. Výběr logického plánu

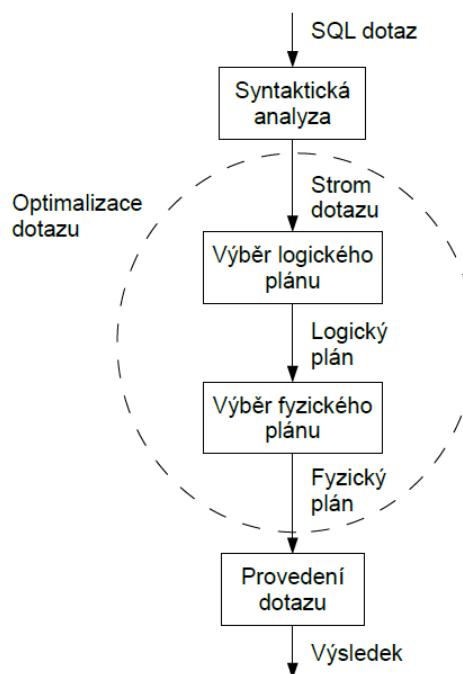
- při převodu do formy relační algebry dochází také k **odstranění** různých povrchových **rozdílů** a především nalezení **efektivnějšího tvaru** než nabízel původní dotaz.
- **Optimalizátor** – transformační pravidla (převádí výraz na ekvivalentní). Fáze transformace/přepsání dotazu (query rewrite). Dotaz tedy není ve skutečnosti vykonán přesně tak, jak byl zadán!

3. Výběr fyzického plánu

- V této fázi se optimalizátor rozhoduje jak bude transformovaný dotaz vykonán.
- Jsou vybírány konkrétní algoritmy a pořadí jejich vykonávání.
- Při výběru se již uvažuje: **existenci indexů, distribuci hodnot, shlukování uložených dat**.
- Každé operaci je přiřazena výsledná **cena = IO cost + CPU cost**, která bude potřebná k celkovému vykonání.

4. Výběr nejlevnějšího plánu a jeho vykonání

- Z množiny dotazovacích plánů pak **optimalizátor** vybírá ten **nejlepší**, tedy **nejlevnější** plán.
- Následně je vykonán a hodnoty vráceny.



37.6 Optimalizátor a ladění dotazů

- **Optimalizátor** pro nalezení nejlepšího plánu využívá systémový katalog, který obsahuje statistiky o uložených datech. Systémový katalog je aktualizován v době nejnižší zátěže serveru, nikoli po každé CRUD operaci. Orientační obsah katalogu:
 - **Tabulka** - kardinalita (mohutnost, počet záznamů), počet diskových stránek
 - **Sloupec** - počet stejných hodnot ve sloupci, minimální a maximální hodnoty, null hodnoty, X nejfrekventovanějších hodnot
 - **Index** - Počet listových stránek, výška stromu
- **Plán** lze v SŘBD většinou **zobrazit** - operace jako průchod tabulkou, přístup k indexu, třídění spojení atd. To lze využít pro **odladění dotazu** - např. uvidíme, že musíme použít index na neindexovaný atribut.
- Dvěma či více **různými dotazy** je možno obdržet **stejná data**.

- **Rychlosť** rôznych dotazov ovšem **nemusí** byť **stejná** i preto, že vracejí stejná data.
- Snažíme dosiahnuť **maximálneho výkonu** se stávajúcimi prostriedky.
- Snažíme sa vytvoriť dotaz, ktorý **bude načítať z úložiště pouze to, co potrebuje**.

37.6.1 Přístupy k ladění

- **Proaktivní** – Analyzujeme fyzický návrh a provádíme změny k lepšímu fungování.
- **Reaktivní** – Reagujeme na problém.

37.6.2 Ladění výkonu SŘBD

- **HW** – RAID, RAM, CPU (Poslení možnost, je to drahé, efektívnejší je **vyladit fyzický návrh**).
- **Parametry SŘBD** – Veľkosťi **cache**, maximum zámku, atď. Nutné optimalizovať na určité použitie.
- **ORM** – Minimum SQL príkazov a objemu prenášených dat. Úroveň izolácie transakcie.

37.6.3 Obsah plánu vykonání dotazu

V plánu lze získat informace:

- **consisteng gets** - logické přístupy + počet záznamů výsledků/velikost pole záznamů,
- **physical reads** - fyzické čtení,
- **sorts(memory)** - počet operací třízení, bez přístupu na disk,
- **sorts(disk)** - počet operací s nejméně jedním přístupem na disk,
- pokud pro logický přístup neexistuje fyzický => **cache hit** (cache hit rate = (cache hit / počet logických čtení) * 100 [%])
- opačný případ **cache miss**,
- vhodné sledovat i další parametry **Bytes received from server, Total execution time**.

S indexem pouze na primárnych klíčoch

OPERATION	OBJECT_NAME	OPTIONS	COST
SELECT STATEMENT			1220
NESTED LOOPS			
NESTED LOOPS			
TABLE ACCESS	STORE_ITEM	FULL	1177
Filter Predicates			
STORE_ITEM.NAME='PRA-2010-10000'			
INDEX	SYS_C00203250	UNIQUE SCAN	0
Access Predicates			
STORE_ITEM.IDPRODUCER=PRODUCER.			
TABLE ACCESS	PRODUCER	BY INDEX ROWID	1

S indexem na selektovaných atributech (WHERE ..)

OPERATION	OBJECT_NAME	OPTIONS	COST
SELECT STATEMENT			5
NESTED LOOPS			
NESTED LOOPS			
TABLE ACCESS	STORE_ITEM	BY INDEX ROWID	5
INDEX	STORE_ITEM_NAME	RANGE SCAN	4
Access Predicates			
STORE_ITEM.NAME='PRA-2010-10000'			
INDEX	SYS_C00203257	UNIQUE SCAN	0
Access Predicates			
STORE_ITEM.IDPRODUCER=PRODUCER.			
TABLE ACCESS	PRODUCER	BY INDEX ROWID	1

38 Objektově-relační datový model a XML datový model: principy, dotazovací jazyky.

38.1 Objektový datový model

Objektový datový model představuje data uložené v **objektové struktuře**. Jde většinou o **objektovou mezivrstvu mezi kódem a databázi**, do které se nasypou data, s kterými pak aplikace pracuje a nezatěžuje dotazy DB server.

Objektové databázové systémy umožňují využití hostitelského objektového jazyka jako je třeba C++, Java, nebo Smalltalk přímo na objekty (skládají se z **atributů** a **metod**) "v databázi"; tj. místo věčného přeskakování mezi jazykem aplikace (např. C) a dotazovacím jazykem (např. SQL) **může programátor jednoduše používat objektový jazyk k vytváření a přístupu k metodám**. Krátce řečeno, ODBMS (Object Database Management System) jsou výborné pro manipulaci s daty. Pokud navíc opomeneme programátorskou stránku, dá se říct, že některé typy dotazů jsou efektivnější než v RDBMS díky dědičnosti a referencím.

38.2 Objektově relační datový model

Objektově orientované DB → **nákladná migrace relačních dat**, proto vzniky **objektově-relační rysy** v relačních DB. (standart **SQL99** - víceméně se nedodržuje, musíme se bavit o konkrétním SŘBD - Oracle).

ORDBMS využívají datový model tak, že "**přidávají objektovost do tabulek**". Všechny **trvalé informace jsou stále v tabulkách**, ale některé položky mohou mít **bohatší datovou strukturu**, nazývanou abstraktní datové typy (**ADT**). ADT je datový typ, který vznikne zkombinováním základních datových typů. Podpora ADT je atraktivní, protože operace a funkce asociované s novými datovými typy mohou být použity k indexování, ukládání a získávání záznamů na základě obsahu nového datového typu. OOSŘBD umožňují používat uživatelské typy, dědičnost, metody tříd, rozlišují pojmy instance a ukazatel na instanci.

38.2.1 Prvky objektově-orientovaných SŘBD

- Uživatelsky definované **datové typy** (s atributy i metodami).
- Uložené **procedury, triggery** (přenesení funkcionality na server).
- Kolekce (**pole** proměnné délky, **vnořené tabulky**).
- Datový typ **ukazatel, reference a dereference**.
- **Dědičnost** – `CREATE TYPE student_type UNDER person_type (... OVERRIDING MEMBER FUNCTION show...)`

38.2.2 Výhody OOSŘBD

- **Objektové typy** a jejich **metody** jsou **uloženy spolu s daty** v databází -> není nutné vytvářet podobné objekty v každé aplikaci.
- Na data je možné nahlížet z relačního i **objektového pohledu**.
- **Objekty** mohou **reprezentovat vazby**, kdy entita se skládá z jiných entit.
- **Metody** jsou **spouštěny na serveru** – nedochází k neefektivnímu přenosu dat po síti, část výkonu přenesena na server.
- Programátor může přistupovat k množině objektů jako by se jednalo o jeden objekt.

38.3 Typy tabulek a datové typy

- **Objektové tabulky** – obsahují pouze objekty, **každý záznam = objekt** (řádkový objekt).
 - **Objekty sdíleny dalšími objekty** by měly být uloženy v objektových tabulkách (mohou být **referencovány**).
 - Buď tabulka s **1 sloupcem objektů** (nad kterými lze provádět objektové metody), nebo tabulka obsahující atributy obj. dat. typu, nad kterou lze provádět relační operace (kompromis mezi O/RDM).
- **Relační tabulky** – Obsahují **objekty spolu s ostatními daty**, mluvíme o tzv. **sloupcovém objektu**.

38.3.1 Objektově relační datové typy a metody

- Data/atributy, operace/metody.
- Normální používání **objektových typů** při definici atributů – `INSERT INTO contacts VALUES (person_type(65, 'John', 'Smith'), 2014-05-29)`
- **Objektový identifikátor (OID)** - identifikuje objekty objektových tabulek, není přístupné přímo, ale jen pomocí reference (typ **REF**). U relačních tabulek s objekty OID nepotřebujeme.
- **Reference na objekt - REF**, ukazuje na objekty stejného typu nebo hodnota NULL, **nahrazuje FK (cizí klíč), asociace/vztahy**. Pokud tabulka obsahuje referenci na objekt jedné tabulky, lze využít `IO: SCOPE IS tabulka`.
- **Dereference** - `SELECT DEREF(e.manager) FROM emp_person_obj_table e;` **Implicitní dereference**: `SELECT e.name, e.manager.name FROM ...`

38.3.2 Kolekce a dědičnost

Používámé pro více hodnotové atributy nebo vazby M:N.

- **Pole – VARRAY** – variable-size array - pole pevné délky s definovanou kapacitou aktuální velikostí.
`DECLARE TYPE Calendar IS VARRAY(366) OF DATE.`
- **Zahnízděná (nested) tabulka** – pole proměnné délky, záznamy ukládány bez ohledu na pořadí, ale po načtení z disku jsou záznamy očíslovány (nezachová hodnotu indexu, jen udrží pořadí).
- **Asociativní pole** – jako hash table - obsahuje dvojice `<key, value>`, kde key = číslo/řetězec, klíče jsou indexovány (zádné sekvenční vyhledávání). Nelze uložit do tabulky, spíše pro malé množství záznamů.
- **Hierarchická dědičnost** – v SŘBD můžeme vytvářet hierarchie typů pomocí dědičnosti. Tato vlastnost nám pak umožní využít všechny rysy objektově-orientovaných technologií, např. mnohotvárnost, polymorfismus.

38.3.3 Typy metod

- **Členské, statické, konstruktor** (pro každý obj. typ definován implicitní).
- **Volání: SELECT c.contact.getID() FROM contacts c;**

```
-- Vytváření objektu s atributy a metodou
CREATE TYPE person_type AS OBJECT (idno NUMBER, first_name VARCHAR2(20),
last_name VARCHAR2(25), MAP MEMBER FUNCTION get_idno RETURN NUMBER);

-- Definice těla metody
CREATE TYPE BODY person_type AS MAP MEMBER FUNCTION get_idno RETURN
NUMBER IS BEGIN RETURN idno; -- vraci id END; END;

-- Objektové datové typy lze používat při definici atributů podobně jako SQL datové typy
CREATE TABLE contacts (contact person_type, contact_date DATE);

-- Záznam uložíme pomocí SQL INSERT:
INSERT INTO contacts VALUES (person_type(65, 'Verna', 'Mills'), '24 Jun 2003');
```

38.4 XML Model a XML (eXtensible Markup Language)

- Značkovací jazyk - W3C standard pro popis slabě strukturovaných dat.
- Data jsou uložena v xml souboru **formou stromu**. (Ve skutečnosti je to graf, ale pro zjednodušení se značí jen jako strom)
- **Logika a význam dat** je součástí xml souboru.
- Skládá se z **hlavičky, kořenového elementu** (právě jeden), **vnořených elementů, atributů a textu** vnořeném uvnitř (ukládané informace).
- Soubor může obshovat také tzv **XML Schema**, viz níže. To může určovat i datový typ a integritní omezení buněk, což je vhodné pro XML databáze.

XML databáze jsou složitější oproti relačním databázím. SQL pracuje s 2 rozměrnými tabulkami, zatímco XML model obsahuje i zanořené atributy. Proto se využívá dotazovací jazyk X-Path. Největší výhodou nativní XML databází je **dynamická změna schémat** uložených dokumentů, která je u relačních databází velmi komplikovaná.

```

<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="books">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="book" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="title" type="xsd:string"/>
              <xsd:element name="author" type="xsd:string"/>
            </xsd:sequence>
            <xsd:attribute name="id" type="IdType" use="required"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:simpleType name="IdType">
    <xsd:restriction base="xsd:string">
      <xsd:length value="9"/>
      <xsd:pattern value="[0-1] +- [0-1] +"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>

```

38.5 Dotazovací jazyky XML

Dotazovací jazyk XML by měl umět: vybrat požadované části dokumentů, transformovat je do požadované podoby, manipulovat s kolekcemi dokumentů a další vlastnosti specifické pro konkrétní problémovou doménu.

38.5.1 XPath

Jazyk používaný pro identifikaci uzlů v XML dokumentu. Pravděpodobně **nejdůležitějším rysem** jazyka XPath je **možnost vyjádření relativní cesty od uzlu k jinému uzlu či atributu**. Od jazyka SQL je značně odlišný. Určuje cestu s dodatečnými podmínkami na názvy uzlů, hodnoty atributů, hloubky zanoření a mnoho dalšího.

Příklad: //zbozi[@sleva >= @cena div 2], který najde všechny elementy zboží, jejichž atribut sleva má hodnotu nejméně poloviny hodnoty atributu cena

38.5.2 XQuery

Připravovaný standard W3C (verze 1.0). XQuery **je silně typovaný jazyk**. Vychází z jazyka Quilt, inspirace XPath 1.0 a pro výběr částí dokumentů používá XPath 2.0.

- \$ – identifikuje proměnné,
- for – iterace přes něco,
- where – omezení výběru,
- order by – řadí,
- return – vrací výsledek,
- let – nastavuje hodnotu pro proměnnou.

```

for $x in doc("books.xml")/bookstore/book
  where $x/price>30
  order by $x/title
  return $x/title

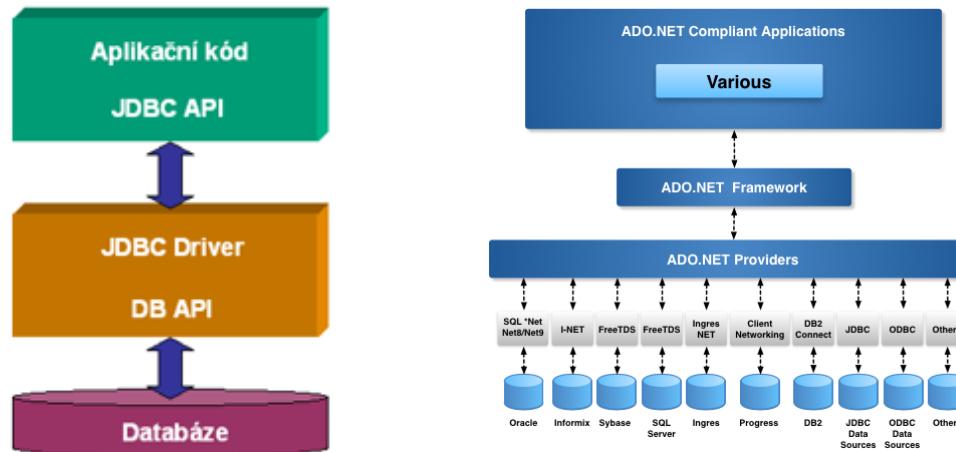
```

39 Datová vrstva informačního systému; existující API, rámce a implementace, bezpečnost; objektově-relační mapování.

39.1 Datová vrstva IS

Datová vrstva informačního systému **odděluje aplikaci od databáze**. Jde o třídy a funkce zajišťující komunikaci s databází. **Překonává propast mezi SŘBD a programovacím jazykem**. Nazývá se také persistentní a je 3. (nebo 1. záleží na úhlu pohledu) vrstvou **třívrstvé architektury IS**.

Jedná se třídy a funkce, které se starají o persistentní ukládání dat, odtud také název. Datová vrstva nemusí nutně ukládat data do databáze, ale klidně do JSONu, XML, nebo přes API na jiných serverech. Proto je oddělená, aby bylo možné měnit „poskytovatele“ dat. Proto se programátor nemusí starat o fyzické uložení dat, pouze o práci s nimi.



- **Nástroje** – programovací jazyky + **SQL**, staví na **API pro dotazování DB** (JDBC, ADO.NET), **embedded SQL** - hostitelský jazyk obsahuje SQL.
 - **ODBC** (Open DataBase Connectivity) – standartizované API pro přístup k databázovým systémům. ODBC se snaží poskytovat přístup nezávislý na programovacím jazyku, databazovém systému a operačním systému.
 - **JDBC** (Java DataBase Connectivity) – Javovské API pro unifikovaný přístup k databazi. Bez ohledu kde jsou data uložena - SQL databáze, Oracle, XML, CSV, DB2, Ingress, ... – se s nimi přes JDBC pracuje stejně.
 - **ADO.NET** (Active Data Object) – specifikace (knihovna) pro přístupu k datovým zdrojům na platformě .NET.
 - **ASP.NET** – knihovny pro informační systémy na platformě .NET. Pro komunikaci s databazi využívají ADO.NET.
 - **Java2EE** – knihovny pro informační systémy na platformě Java.
- **Speciální prostředí** – implementováno výrobcem SŘBD (Oracle Forms, APEX). Jednoduché, šité na míru, není připravené pro velké projekty a týmový vývoj.
- **Architektury** – jsou **komplikované, vícevrstvé**, ale umožňují jednodušší vývoj a zapojení více vývojářů. (3-vrstvá architektura => Business, Data, Presentation layer).

39.1.1 JDBC

- Rozhraní pro **unifikovaný přístup k datům** v DB na platformě **Java**.
- Inspirováno rozhraním **ODBC**.
- Zprostředkování komunikace aplikace s konkrétním typem DB (ovladače k většině).
- **Dotazovací jazyk** – **SQL**. Předá se DB, ovladač vyhodnotí přímo.
- **Schéma provedení dotazu**: 1) Připojení k datovému zdroji. 2) Inicializace. 3) Vytvoření a provedení dotazu. 4) Získání výsledku. 5) Ukončení transakce. 6) Odpojení od datového zdroje.
- Hlavní třída **Connection** - odesílá **Statementy** a obdrží **ResultSety**.
- **Podpora transakcí** (implicitně, každý příkaz jedna transakce, jsou vázány k instanci connection).

39.1.2 ADO.NET

- **Connection** spouští **Commandy**, výsledkem je **DataSet** (použití **DataAdapteru**) nebo čistá data čtená **DataReaderem**.
- **DataSet** → **DataTable[]** → **DataRow[]** → **DataColumn[]**
- **DataRelation** vazba mezi **DataSety** s vazebním **DataColumn**.
- Umožňuje **parametrizovat commandy**.
- Třída **TransactionScope** pro distribuované transakce.

39.2 ORM (Objektově-relační mapování)

Programovací technika **zprístupňující relační či objektově-relační data pro objektové prostředí**. Entita v ORM je objekt, který je uložen v SŘBD (nejčastěji jako jeden záznam). Nejčastěji je implementován jako třída. Díky rozhraní jsou jednotlivé implementace zaměnitelné. Vlastnosti ORM rámců:

- Práce s objektovým modelem (přenositelnost mezi různými SŘBD).
- Překonává propast mezi SŘBD a programovacím jazykem.
- **Rychlejší vytváření aplikací** vs Menší výkon aplikace.
- Nemusí využívat všechny vlastnosti SŘBD (efektivita, bezpečnost apod.). Záleží na komplexnosti a kvalitě knihovny ORM.
- Typová kontrola (nevznikají chyby v SQL).
- Jednoduší testování.
- Může využívat existující API jako JDBC, ADO.NET apod.
- Měly by se používat pouze metody a parametry ORM, nikoli přímo SQL dotazy.
- Využívá známé návrhové vzory **Table data gateway**, **Row data gateway**, **Active record**, **Data mapper**.
- Dobré ORM si dokáže poradit i s dědičností pomocí návrhových vzorů **Single-table inheritance**, **Class table inheritance** a **Concrete table inheritance**.

39.2.1 Pravidla ORM

- **Minimalizace počtu dotazů** zasílaných na server.
- **Minimalizovat objem dat** získaných z DB, stahovat pouze ta data, které potřebujeme, a které jsou zobrazeny uživateli (Lazy Loading).

39.2.2 Vlastní implementace ORM

- Využití **API pro komunikaci s DB** (JDBC, ADO.NET).
- Používá se **SQL**.
- **Výhody:** plná kontrola nad výkonem (stahujeme jen to co chceme) a prováděnými příkazy. Můžeme využít konkrétní prvky daného SŘBD.
- **Nevýhody:** aplikace je závislá na určitém typu SŘBD, při rozšiřování systému je často nutné rozšířit také ORM.

39.2.3 Automatizované

- Frameworky **Hibernate** (Java), **LINQ2SQL**, **EntityFramework** (ASP.NET).
- Může výrazně **degradovat výkon** (musí provádět interpretaci do SQL, vytváří objektový model atd.).
- Většinou mají vlastní **cache**. Většinou jsou **nezávislé na daném SŘBD**.

39.3 Bezpečnost

(Ne)Omezení práv - uživatel může destruktivně vymazat celou DB, nebo číst data, která nemá. **Řešením** je omezení práv na jednotlivé akce, či úplné zamezení viditelnosti tabulek. Pro zobrazení jejich částí lze využít pohledy.

SQL Injection - zranitelnost vznikající při nedostatečném **ošetření vstupů užívaných** v SQL dotazech. Např.: při přihlašování na webových stránkách zadá útočník příkaz SQL, které změní funkčnost dotazovaného příkazu. **Řešení**:

- **Parametrizované dotazy** (hodnoty jsou předány zvlášť, nemůže dojít k úpravě).
- Uložené procedury.
- Ošetření vstupů (`htmlspecialchars`, `mysql_real_escape_string`) - zneužití např. pomocí jiného kódování!
- Uživatel musí zadávat komplikovaná (neslovníková) hesla, **časové omezení počtu** pokusů. Logování neúspěšných pokusů o autentizaci.

39.4 Způsoby doménově-relačního chování

- **Table data gateway** - Vrací Dataset DTO bez znalosti domény. Dataset obsahuje pole řádků, i když je navrácen pouze 1. Jediná instance drží i více řádků tabulky. Protože nezná doménu, každá funkce obsahuje tolik parametrů, kolik je sloupců.
- **Row data gateway** - Každý řádek je 1 objekt. Obsahuje funkce pro `update` `delete`. Je potřebná nějaká statická třída `Finder` pro vytváření nových instancí, protože pokud nejsou data, neexistuje ani objekt.
- **Active record** - Každá instance je 1 řádek. Obsahuje i funkce `insert` a `update` atd. Ale také metody pro práci v programu, různé výpočetní apod.
- **Data mapper** - Mapper třída mapuje z DB data na objekty a zpět. Každý objekt obsahuje veškeré parametry a také funkce pro práci s nimi na úrovni programu. Do mapperu se nevkládají jednotlivá data jako parametry ale vždy daný objekt, který samotný funkce pro práci s DB postrádá.

40 Distribuované SŘBD, fragmentace a replikace.

40.1 Distribuované SŘBD

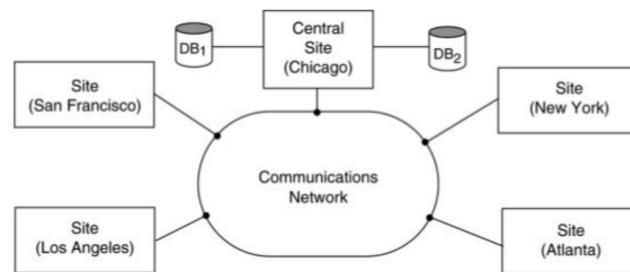
Distribuovaný databázový systém (DDBS) je tvořen **distribuovanou bází dat a programovým vybavením**, skládajícím se z **lokálních SŘBD** a dalších programů potřebných k jejich **koordinaci a řízení**, k zabezpečení celosystémových úloh spojených s přístupem uživatelů k DDBS, s udržováním integrity a provozuschopnosti v prostředí počítačové sítě. **Uživateli se jeví celá distribuovaná báze jako by byla lokální** (na jednom místě) a přistupuje k ní stejným způsobem.

Podle toho jak je dist. databáze řešena a kam směřují dotazy ji dělíme na:

40.1.1 Centralizované

Mají popis a řízení DDBS soustředěno na **1 centrální počítač**. Toto centrum nemusí být v centru počítačové sítě. Jsou zde soustředěny **popisky všech dat** tvořících DDBS a **centrálně se řídí: přístup k datům, provádění změn** ve struktuře dat, **provádění a synchronizace transakcí** + všechny další činnosti systému.

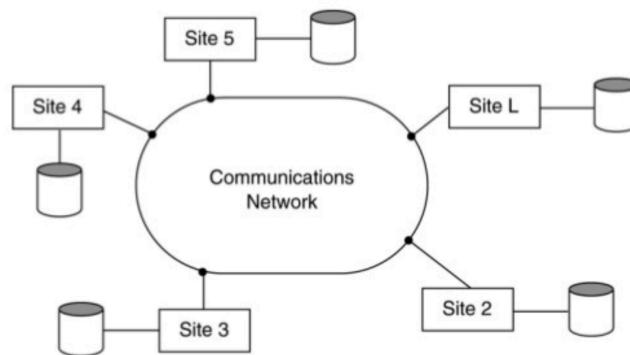
- **Výhody** – jednoduchost řízení.
- **Nevýhody** – vysoké náklady na komunikaci, pomalost (každý přístup k datům musí být povolen centrem) a nebezpečí výpadku tohoto jednoho prvku.



40.1.2 Decentralizované

Jsou tvořeny počítačovou sítí, kde žádný uzel nemá privilegované postavení. **Všechny počítače mají stejné informace o DDBS a stejnou odpovědnost za integritu DDBS**.

- **Výhody** – větší stabilita (výpadek počítače způsobí max. ztrátu přístupu k lokálním datům).
- **Nevýhody** – složitější algoritmy pro řízení transakcí než v centralizovaných DDBS.



40.2 Rozmístění dat (fragmentace a replikace)

U DDBS jsou nejvíce časově náročné operace přesunu dat po komunikační lince. Vzhledem k vyšší možnosti poruch je třeba relace ukládat tak, aby jejich **dostupnost byla zajištěna** i v případě výpadku části sítě. Relací zde rozumíme logický celek, ale fyzicky i více souborů. Používají se **2 hlavní způsoby** fyzické implementace:

- **Replikace** – (přítomnost duplicitních dat) **uchovávání kopií relací v různých uzlech, porucha uzlu neznemožní přístup k relaci** (díky tomu se dist. databáze i samozálohuje), problém je v aktualizaci všech kopií relací. Proto se v DDBS ukládají v kopiích ta data, ke kterým je třeba **rychlý přístup** a nejsou často aktualizována a jsou velmi důležitá.

- **Fragmentace** – znamená **rozložení relací na menší části** (fragmenty), které jsou umístěny v různých uzlech sítě. Může jít o **horizontální** fragmentaci, kdy se v různých uzlech sítě ukládají části relace rozložené do skupin řádků nebo **vertikální** fragmentaci, kdy se v různých uzlech ukládají různé projekce relací. Fragmentace se provádí, aby se **původní relace získala zpět standardními operacemi nad relační databází (sjednocení, spojení)**.

Obě metody se často kombinují tak, že se v distribuované bázi uchovávají kopie fragmentů v různých uzlech. V distribuované databázi musí být unikátní jména všech položek. Tedy ani 2 lokální položky nesmí používat stejná jména pro různé položky. Jednou z možností jak tento problém vyřešit je **centrální slovník**, ovšem ten je zase úzkým místem systému.

Část V

Počítače a sítě

41 Architektura univerzálních procesorů. Principy urychlování činnosti procesorů.

41.1 Architektura mikroprocesorů

Architektura procesoru je náčrt struktury a funkčnosti systému. Je charakterizována výčtem **registrov** a jejich funkcí, vnitřních a vnějších **sběrnic**, způsobem **adresování** a **instrukčním souborem**.

Registr je malé úložiště dat v mikroprocesoru s rychlým přístupem, které slouží jako **pracovní paměť** během výpočtu.

Sběrnice je soustava vodičů pro **přenos informací** mezi více účastníky na principu „jeden vysílá, ostatní přijímají.“ Podle typu přenášené informace je dělíme na **datové, adresové a řídící**. V praxi však díky multiplexu může jít o jedny dráty.

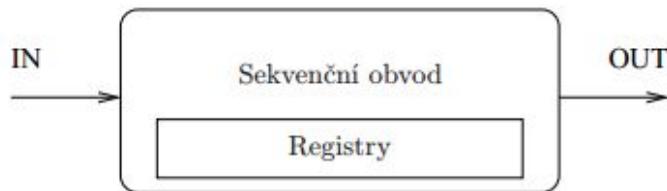
41.2 Procesory CISC a RISC

V dnešní době se ustáli do dělení počítačů do dvou základních kategorií podle typu používaného procesoru:

- **CISC** – počítač se složitým souborem instrukcí (*Complex Instruction Set Computer*)
- **RISC** – počítač s redukovaným souborem instrukcí (*Reduced Instruction Set Computer*)

41.2.1 CISC

- Procesory s **komplexním instrukčním souborem**.
- Instrukce mají **proměnlivou délku i dobu vykonání**.
- **Vysoká složitost instrukcí** → nutný systematický návrh řadiče procesoru.
- Vykonání strojové instrukce probíhá posloupností mikrooperací (předepsána mikroinstrukcí v řídící paměti).
- Procesor obsahuje relativně **nízký počet registrů**.
- Operace provedená i **složenou instrukcí** (např. násobení) může být **nahrazena** sledem jednodušších strojových instrukcí (*sčítání a bitové posuny*) → mohou být ve výsledu vykonány rychleji, než hardwarově implementovaná složená varianta.
- Označení **CISC** bylo zavedeno jako **protiklad** až poté, co se prosadily procesory RISC, které mají instruční sadu naopak maximálně redukovanou (pouze jednoduché operace, tj. žádné složené, jsou stejně dlouhé a jejich vykonání trvá stejnou dobu).
- Obvyklou chybou je domněnka, že procesory CISC mají více strojových instrukcí, než procesory RISC. Ve skutečnosti nejde o absolutní počet, ale o **počet různých druhů operací**, které procesor sám přímo umí vykonat na hardwarové úrovni (tj. již z výroby). Procesor CISC tak může například paradoxně obsahovat jen jednu strojovou instrukci pro danou operaci (např. *logické operace*), zatímco procesor RISC může tuto operaci obsahovat jako několik strojových instrukcí, které stejnou operaci umí provést nad různými registry.



Obrázek 1: Procesor (CISC) jako sekvenční obvod

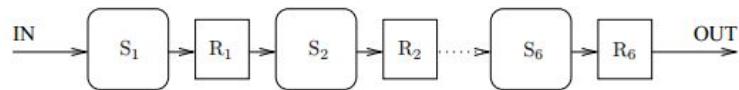
Krok	Význam
1.	VI Výběr Instrukce
2.	DE Dekódování
3.	VA Výpočet Adresy
4.	VO Výběr Operandu
5.	PI provedení Instrukce
6.	UV Uložení Výsledku

	T ₁	T ₂	T ₃	T ₄	T ₅	T ₆	T ₇	T ₈	T ₉	T ₁₀	T ₁₁	T ₁₂	T ₁₃	
VI	I ₁						I ₂							...
DE		I ₁						I ₂						
VA			I ₁						I ₂					
VO				I ₁						I ₂				
PI					I ₁						I ₂			
UV						I ₁						I ₂		

Tabulka 4: Postup provádění instrukcí procesorem CISC

41.2.2 RISC

- Počet instrukcí a způsobů adresování je malý, ale zůstává úplný, aby bylo možno provést vše → v tomhle se liší od CISC.
- Instrukce jsou vytvořeny pomocí obvodu → jednodušší na výrobu než CISC.
- Je menší počet instrukcí, takže složitější instrukce se nahradí větším počtem jednodušších.
- To způsobuje nárůst kódu. Zároveň ale vznikly rychlejší sběrnice, tj rychlejší proud dat do procesoru.
- Používá se **zřetězené zpracování instrukcí** (Blíže popsáno níže).
- Instrukce se provádějí jen nad registry.
- Navýšený počet registrů → delší program.
- Instrukce mají **jednotný formát** – délku i obsah.
- Komunikace s pamětí pouze pomocí instrukcí **LOAD / STORE**, adresování i práce je celkově rychlejší.
- V návaznosti je využívána cache pro co nejrychlejší přístup dat.
- Využívá se **predikce skoků**, takže se začnou načítat data, která pravděpodobně budou v další instrukci potřeba.
- **Každý strojový cyklus znamená dokončení jedné instrukce.**
- Řešení problémů s frontou instrukcí.
- Mikroprogramový řadič může být nahrazen rychlejším obvodem.
- Představitelé **ARM, MOTOROLA 6800, INTEL i960, MIPS R6000**.



Obrázek 2: Zřetězené zpracování (v procesoru RISC)

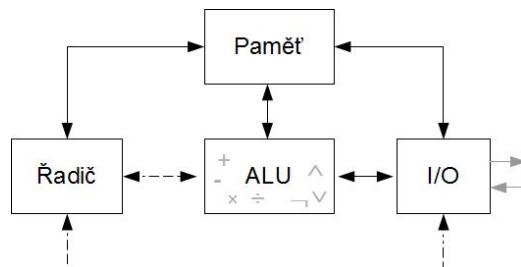
	T ₁	T ₂	T ₃	T ₄	T ₅	T ₆	T ₇	T ₈	T ₉	T ₁₀	T ₁₁	T ₁₂
VI	I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇	...				
DE		I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇				
VA			I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇			
VO				I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇		
PI					I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇	
UV						I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇

Tabulka 5: Zřetězené provádění instrukcí procesorem RISC

41.3 Von Neumannovo schéma počítače

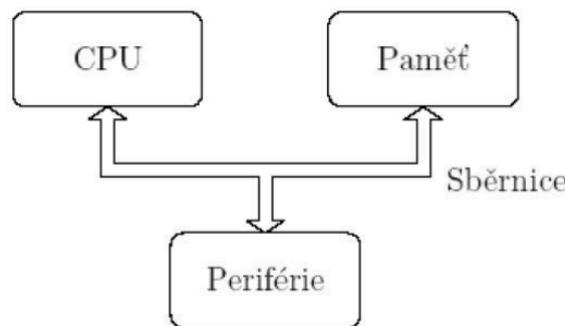
John Von Neumann definoval v roce **1945** základní koncepci počítače (EDVAC) **řízeného obsahem paměti**. Od té doby se objevilo několik odlišných modifikací, ale v podstatě se **počítače v dnešní době** konstruují podle tohoto modelu. Ve svém projektu si von Neumann stanovil určitá kritéria a principy, které musí počítač splňovat, aby byl použitelný univerzálně. Můžeme je ve stručnosti shrnout do následujících bodů:

- Počítač se skládá z paměti, řídící jednotky, aritmeticko-logické jednotky, vstupní a výstupní jednotky.
 1. **ALU** – aritmeticko-logická jednotka (arithmetic-logic unit) => jednotka provádějící veškeré aritmetické výpočty a logické operace. Obsahuje sčítáčky, násobičky a komparátory.
 2. **Operační paměť** – slouží k uchování zpracovávaného programu, zpracovávaných dat a výsledků výpočtu.
 3. **Řídící jednotka** – řídí činnost všech částí počítače. Toto řízení je prováděno pomocí řídících signálů, které jsou zasílány jednotlivým modulům. Řadič jsou pak zpět zasílané stavové hlášení. Dnes řadič spolu s ALU tvoří jednu součástku, a to procesor neboli CPU (Central Processing Unit).
 4. **Vstup/Výstup** – zařízení určené pro vstup dat, a výstup zpracovaných výsledků.
- Struktura pc je **nezávislá na typu řešené úlohy** (univerzálnost), **počítač se programuje obsahem paměti**.
- Následující krok počítače je závislý na kroku předešlém.
- **Instrukce a data** jsou v téže paměti.
- Paměť je rozdělena do **paměťových buněk stejné velikosti (Byte)**, jejichž pořadová čísla se využívají jako adresy.
- Program je tvořen posloupností instrukcí, které se vykonávají jednotlivě v pořadí, v jakém jsou zapsány do paměti.
- Změna pořadí prováděných instrukcí se provádí **skokovými instrukcemi** (podmíněné nebo nepodmíněné skákání na adresy).
- Čísla, instrukce, adresy a znaky se značí v **binární soustavě**.



41.3.1 Nevýhody Von Neumannovy koncepce ve srovnání s dnešními PC

- Podle von Neumannova schématu počítač pracuje **vždy nad jedním programem**. Toto vede k velmi špatnému využití strojového času. Dnes je obvyklé, že počítač **zpracovává paralelně více programů zároveň** - tzv. **multitasking**.
- Počítač může mít i více jak jeden procesor.
- Podle Von Neumanova schématu mohl počítač pracovat pouze v tzv. **diskrétním režimu**, kdy byl do paměti počítače zaveden program, data a pak probíhal výpočet. V průběhu výpočtu již nebylo možné s počítačem dále interaktivně komunikovat.
- Dnes existují **vstupní/výstupní** zařízení, např. pevné disky a páskové mechaniky, které umožňují vstup i výstup.
- Program se do paměti nemusí zavést celý, ale je možné zavést pouze jeho část a ostatní části zavádět až v případě potřeby.



Výhody

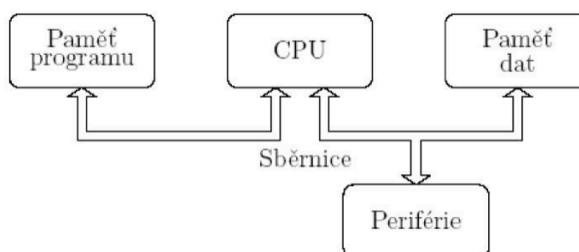
- + **Rozdělení paměti** pro kód a data určuje programátor, řídící jednotka přistupuje pro data i instrukce jednotným způsobem.
- + **Jedna sběrnice** -> jednodušší levnější výroba.

Nevýhody

- **Společné uložení dat a kódu** může mít za následek přepsání vlastního programu.
- **Jedna sběrnice** je omezující.

41.4 Hardvardské schéma počítače

Několik let po von Neumannovi, přišel vývojový tým odborníků z Harvardské univerzity s vlastní koncepcí počítače, která se sice od Neumannovy příliš nelišila, ale odstraňovala některé její nedostatky. V podstatě jde pouze o **oddělení paměti pro data a program**. Abychom si mohli obě koncepce porovnat, můžeme vycházet ze zjednodušených schémat.



Výhody

- + **Program se nepřepíše** (oddělené paměti pro data a program).
- + Dvě sběrnice umožňují **paralelní** načítání instrukcí a dat.
- + Paměti mohou být vyrobeny **odlišnými technologiemi** a každá může mít jinou nejmenší adresovací jednotku (8 bitů pro instrukce a 8, 16 nebo 32 pro data).

Nevýhody

- 2 sběrnice mají **vyšší nároky na vývoj** řídící jednotky a jsou také dražší a složitější na výrobu.
- Paměť je **rozdělena** už od **výrobce**.
- Nevyužitou část dat **nelze využít** po program a obráceně.

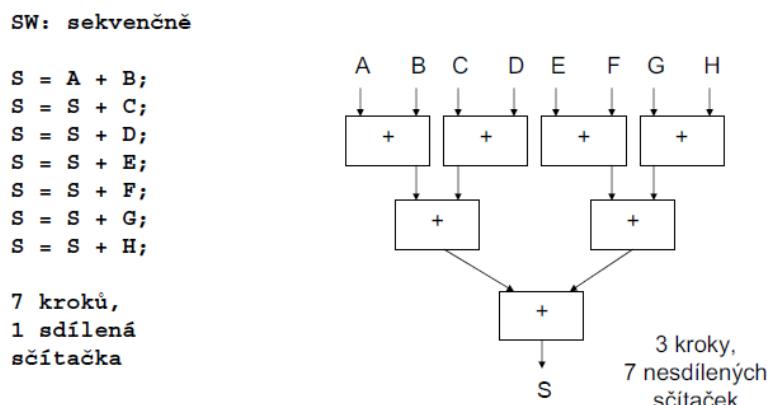
41.5 Principy urychlování činnosti procesorů

- Speciální kódování dle potřeby dané úlohy.
 - Speciální výpočetní jednotky dle potřeby dané úlohy (FFT – rychlá fourierova transformace).
 - Paralelní zpracování (násobné výpočetní jednotky).
 - **Zřetězové zpracování instrukcí** (pipelining).
 - Využití cache pamětí (L1, L2, L3).
 - Predikce skoků

41.5.1 Paralelní zpracování

Zpracování více elementárních úloh běží současně.

Př. $S = A + B + C + D + E + F + G + H$



41.5.2 Zřetězené zpracování instrukcí (pipelining)

Princip zřetězení se značně překrývá s principy procesorů RISC. Základní myšlenkou je **rozdělení zpracování jedné instrukce** mezi různé části procesoru a tím i dosažení možnosti **zpracovávat více instrukcí** najednou. Pro dosažení tohoto zřetězení je nutné rozdělit úlohu do posloupnosti dílčích úloh, z nichž každá může být vykonána **samosatně**, např. oddělit načítání a ukládání dat z paměti od provádění výpočtu instrukce a tyto části pak mohou běžet souběžně. To znamená že musíme osamostatnit jednotlivé části sekvenčního obvodu tak, aby každému obvodu odpovídala jedna fáze zpracování instrukcí. Všechny fáze musí být **stejně časově náročné**, jinak je rychlosť **degradována** na nejpomalejší z nich. Fáze zpracování je rozdělena minimálně na 2 úseky:

- Načtení a dekódování instrukce.
 - Provedení instrukce a případné uložení výsledku.

Zřetězení se stále vylepšuje a u novějších procesorů se již můžeme setkat stále s více řetězci rozpracovaných informací (více pipelines), dnes je standardem 5 pipelines.

41.5.3 Problém a predikce skoků

Největší problém spočívá v **plnění zřetězené jednotky**, hlavně při provádění **podmíněných skoků**, kdy během stejného počtu cyklů se vykoná více instrukcí. U pipelingu se instrukce následující po skoku vyzvedává dřív, než je skok dokončen. **Primitivní implementace** vyzvedává vždy **následující instrukci**, což vede k tomu, že se vždy mylí, pokud je skok nepodmíněný. Pozdější implementace mají **jednotku předpovídání skoku (1bit)**, která vždy správně **předpoví nepodmíněný skok** a s použitím cache se záznamem předchozího chování programu se pokusí předpovědět i cíl podmíněných skoků nebo skoků s adresou v registru nebo paměti. V případě, že se predikce nepovede, bývá nutné vyprázdnit celou pipeline a začít vyzvedávat instrukce ze správné adresy, což znamená relativně **velké zdržení**. Související problémem je přerušení.

41.5.4 Plnění fronty instrukcí

Pokud se dokončí skoková instrukce, která odkazuje na jinou část kódu, musejí být instrukce za ní zahozeny (*problém plnění fronty instrukcí*).

- U malého zřetězení **neřešíme**.
- Používání bublin na vyprázdnění pipeline, **naplnění prázdnými instrukcemi**.
- **Predikce skoku** – vyhrazen jeden bit předurčující, zda se skok provede či nikoliv.
- **Statická** – součást instrukce → řeší programátor nebo kompilátor
- **Dynamická**
 - **jednobitová** – zaznamenává jestli se skok provedl, či ne (1/0)
 - **dvoubitová** – metoda zpožděného skoku → v procesoru řeší se např. tabulkou s 4 kB instrukcí

Zřetězené zpracování přináší urychlení výpočtu nejen v procesorech, ale i jiných číslicových obvodech (např. pro zpracování obrazu, bioinformatických dat apod.). Pokud použijeme zřetězené zpracování, musíme dodat řadu podpůrných obvodů a řešit řadu nových problémů. **Moderní procesory používají kromě zřetězení i další koncepty**:

- **Superskalární architektura** (zdvojení) – když nastane podmíněný skok, začnou se vykonávat instrukce obou variant, nepotřebná část se pak zahodí. Tento způsob, pak vyžaduje vyřešit ukládání výsledku.
- **VLIW procesory** (Very long instruction word) – umožňuje instrukční paralelismus (vykonávání několika nezávislých instrukcí souběžně), velmi dlouhé instrukční pakety.
- **Vektorové procesory** – je navržený tak, aby dokázal vykonávat matematické operace nad celou množinou čísel v daném čase. Je opakem skalárního procesoru, který vykonává jednu operaci s jedním číslem v daném čase.
- **Vícevláknové procesory**

42 Základní vlastnosti monolitických počítačů a jejich typické integrované periférie. Možnosti použití.

42.1 Monolitické počítače (mikroprocesory)

- Mikroprocesory, mikrokontroléry, minipočítače jsou další názvy pro monolitické počítače.
- Jsou to **malé počítače integrované v jediném pouzdře** (all in one) s nízkým výkonem (oproti běžným CPU).
- Mají širokou oblast využití.
- Využívá se Harvardské koncepce, což umožňuje aplikovat paměti pro data a program různých technologií.
- Zjednodušené rysy architektury RISC.
- INTEL 8051 (standard), ATTEL, MICROCHIP PIC.
- V monolitických počítačích můžeme najít dva základní typy periférií (**vstupní/výstupní**).
- Často umožňují se přepnout celý MCU do režimu spánku, a tím výrazně snížit svou spotřebu. Poté běží pouze malá část periferií a ostatní jsou odpojeny.

42.1.1 Rozdělení paměti

- **Pro data** – používáme většinou paměti energeticky závislé typu **RWM–RAM** (*Read–Write Memory–Random Access Memory*), tedy paměť s libovolným přístupem pro čtení i zápis. Jsou vyráběny jako statické (uchování paměti po celou dobu napájení), jejich pamětové buňky jsou realizovány jako klopné obvody.
- **Pro program** – se používají paměti typu **ROM** (*Read–Only Memory*) určené především ke čtení (paměť je uchována i po odpojení napájení). Mezi nejčastěji používané paměti patří **EPROM**, **EEPROM** (*Electrically Erasable Programmable Read–Only Memory*), **PROM** (*Programmable Read Only Memory*) a **Flash**.

42.2 Organizace paměti

- **Střadačové (pracovní) registry** – ve struktuře procesoru jsou obvykle **1–8–16** základních pracovních registrů, jsou nejpoužívanější. Ukládají se do nich **aktuálně zpracovávaná data** a jsou nejčastějším operandem strojových instrukcí (to na co se instrukce v závorkách odkazují). A také se do nich nejčastěji ukládají výsledky operací. Nejsou určeny pro dlouhodobé ukládání dat. Nejrychlejší.
- **Univerzální zápisníkové registry** – jsou jich desítky až stovky. Slouží pro ukládání **nejčastěji používaných dat**. Instrukční soubor obvykle dovoluje, aby se část strojových instrukcí prováděla přímo s těmito registry. Formát strojových instrukcí ovšem obvykle nedovoluje adresovat velký rozsah registru, proto se implementuje několik stejných skupin registru vedle sebe, s možností mezi skupinami přepínat – **registrové banky**.
- **Paměť dat RWM** – slouží pro ukládání **rozsáhlejších** nebo **méně používaných dat** (z těch předešlých nejméně používaných). Instrukční soubor obvykle nedovoluje s obsahem této paměti přímo manipulovat, kromě instrukcí přesunových. Těmi se data přesunou např. do pracovního registru. Některé procesory dovolují, aby data z této paměti byla použita jako druhý operand strojové instrukce, výsledek ale nelze zpět do této paměti uložit přímo. Nejpomalejší.

42.3 Zdroje synchronizace

MCU umožňuje využít interní oscilátory nebo na jeden vstup přivést externí oscilátor a tím tak určit přesný takt procesoru. Možnosti zdrojů jsou:

- **krystal** (křemenný výbrus) – jsou drahé ale přesné,
- keramický rezonátor,
- obvod RC – snadno integrovatelný,
- obvod LC – méně časté.

42.4 Ochrana proti rušení

Na prvním místě jde o ochranu **mechanickou**. Odolávat náhodným nárazům, nebo i trvalým vibracím nebo **elektromagnetickým** vlivům z okolí. Pro odstranění chyb, které nastanou působením vnějších vlivů nebo chyby programátora, je v mikropočítáčích implementován speciální obvod nazývaný **WATCHDOG** →. Jedná se o interní čítač, který musí být programem pravidelně resetován. Pokud nedojde k resetu watchdogu a ten přeteče, je celý MCU RESETován. Nastává nejčastěji při zacyklení. Watchdog (WDT) se řadí mezi elektrické ochrany. Tam také můžeme zařadit **BROWN-OUT** – ochrana proti podpěti.

42.5 Typické periferie

Periferie - obvody, které zajišťují komunikaci mikropočítace s okolím.

1. **Vstupní a výstupní brány** – Nejjednodušší a nejčastěji používané rozhraní pro vstup a výstup informací je u mikropočítáčů **paralelní brána - port**. Bývá obvykle organizována jako **x jednobitových vývodů**, kde lze současně zapisovat i čist logické informace 0 a 1. Nejčastější jsou 8 bitové porty, existují ale také 4, 6, 7 bitové i další. U většiny bran lze jednotlivě **nastavit**, které bitové vývody budou sloužit jako **vstupní** a které jako **výstupní** a v průběhu programu tyto vlastnosti měnit. Na vstupu je **Schmittův klopný obvod**. U mnoha mikropočítáčů jsou brány implementovány tak, že s nimi instrukční soubor může pracovat jako s množinou vývodu, nebo jako s jednotlivými bity.
2. **Čítače a časovače** – Do skupiny nejpoužívanějších periferií mikropočítáče určité patří čítače a časovače. Časovač se od čítače příliš neliší. Není, ale **inkrementován** vnějším signálem, ale **přímo vnitřním hodinovým signálem** používaným pro řízení samotného mikropočítáče. Lze tak podle přesnosti zdroje hodinového signálu zajistit řízení událostí a chování v **reálném čase**. Při přetečení časovače se i zde může automaticky předávat signál do přerušovacího podsystému mikropočítáče.
Ve většině MCU je čítač i časovač implementován jako 1 periferie a lze programově nastavit, je li řízena vnějším či vnitřním hodinovým signálem. Navíc lze také programově zařazovat děličky, čímž lze snížit celkovou frekvenci inkrementace.
3. **Sériové linky** – Sériový přenos dat je v praxi stále více používán. Dovoluje efektivním způsobem přenášet data na relativně velké vzdálenosti při použití minimálního počtu vodičů. Hlavní nevýhodou je však **nižší přenosová rychlosť**, a to že se data musí kódovat a dekódovat.
 - **USART (RS232)** +/-12V je transformována na TTL/RS422/RS485.
 - **I2C** (Philips) komunikace mezi integrovanými obvody (přenos dat uvnitř elektronického zařízení).
 - **SPI**
4. **A/D a D/A převodníky** – Fyzikální veličiny, které vstupují do mikropočítáče, jsou většinou reprezentovány **analogovou formou** (napětím, proudem, nebo odporem). Pro zpracování počítáčem však potřebujeme informaci v digitální (číselné) formě. K tomuto účelu slouží analogově-číslicové převodníky.
A/D převodník je nejčastěji realizován **komparátorem**, kde se postupně zvyšuje napětí a porovnává se se vstupní hodnotou. Tím se určí výsledná úroveň, která má ale určité rozlišení. Je omezený počet kroků, ve kterých se hodnota porovnává. Tento převod ale trvá nějakou dobu a nelze tak měřit velmi rychle se měnící signál.
D/A převodník většinou v MCU neexistuje. Pro převod digitálního signálu na analogový se používá **PWM** (pulsně šířková modulace). Tedy velmi rychle se měnící signál plného a žádného napětí, který je následně kondenzátorem vyhlazen.
5. **Obvody reálného času** (RTC - Real Time Clock) - V mnoha aplikacích s použitím mikropočítáčů je potřeba dodržovat přesnou časovou souvislost řízených událostí. Jde tedy o řízení v reálném čase. Ne vždy, ale taková posloupnost dostačuje a je nutno pro potřebu řízení udržovat skutečný čas, tedy hodiny, minuty, sekundy a případně i zlomky sekund. Pro tyto účely slouží obvody **RTC**. Při jejich použití je obvykle nutné vyřešit dva základní problémy:
 - **záložní zdroj** – je třeba zajistit záložní zdroj pro udržení nepřetržité činnosti obvodu (může dojít k výpadku proudu a tak i k ztrátě skutečného času).
 - **čtení dat** – čas je hodnota neustále se měnící. Např. pokud zahájíme čtení hodnoty v čase 10:59:59, může se stát, že po přečtení prvních dvou hodnot, v našem případě hodin, se čas posune na 11:00:00 a čtení dalších hodnot bude neplatné (řešení technicky pomocnými registry v RTC obvodu, nebo vhodným programovým řešením).

42.6 I²C

Multi-masterová počítačová sériová sběrnice vyvinutá firmou Philips, která je používána k připojování nízkorychlostních periferií k základní desce nebo vestavěnému systému.

- Dvoudrátová, dvouvodičová sběrnice se sériovým přenosem.
- Obsahuje **master** (zahajuje a ukončuje komunikaci; generuje hodinový signál SCL) a **slave** (zařízení adresované masterem) obvody.
- Lze propojit až 128 zařízení (Master, slave).
- **Adresa zařízení** – skládá se ze 7 bitů (horní 4 určuje výrobce, dolní 3 jdou nastavit libovolně).
- **Signály** – SCL (synchronous clock), SDA (synchronous data).

43 Protokolová rodina TCP/IP.

43.1 Model ISO/OSI

Počítačové sítě vyvíjelo více firem, zpočátku to byly uzavřené a nekompatibilní systémy. Hlavním účelem sítí je však vzájemné propojování, a tak vytvárala potřeba stanovit pravidla pro přenos dat v sítích a mezi nimi. **Mezinárodní ústav pro normalizaci ISO** (International Standards Organization) vypracoval tzv. referenční model **OSI** (Open Systems Interconnection), který rozdělil práci v síti do **7 vzájemně spolupracujících vrstev**.

Princip spočívá v tom, že vyšší vrstva převezme úkol od podřízené vrstvy, zpracuje jej a předá vrstvě nadřízené. Vertikální spolupráce mezi vrstvami (nadřízená s podřízenou) je **věcí výrobce** sítě. Model **ISO/OSI doporučuje**, jak mají vrstvy **spolupracovat horizontálně** – dvě stejné vrstvy modelu mezi různými sítěmi (či sítové prvky různých výrobců). Model je důležitý především pro výrobce sítových komponent. V praktické práci se sítí jej moc nevyužijeme. Umožňuje však pochopit principy práce sítových prvků a zároveň patří k základní terminologii sítí.

Každá vrstva je samostatná **entita**, jejíž výstup je **PDU** – protocol data unit. Těmito PDU komunikují entity mezi sebou. Každá další entita zapouzdří (přidá svou hlavičku) entitu z předchozí vrstvy a pošle dále. Vrstva 7–4 je v koncových zařízeních, vrstva 3–1 je v sítově orientačních zařízeních.

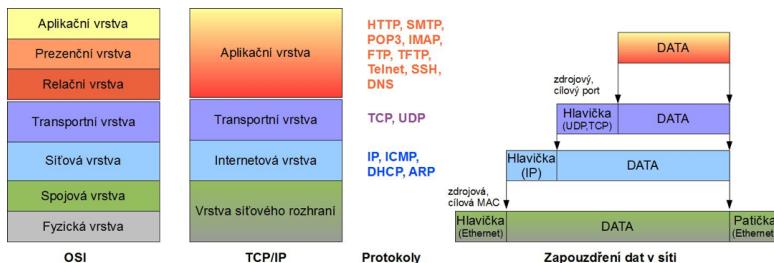
- **7. Aplikační vrstva** - je **určitou aplikací** (např. oknem v programu) zpřístupňující uživatelům sítové služby. Nabízí a zajišťuje přístup k souborům (na jiných počítačích), vzdálený přístup k tiskárnám, správu sítě, elektronické zprávy (včetně e-mailu). Protokoly **HTTP**, **FTP**, **POP3** apod.... PDU této vrstvy se nazývá **zpráva**.
- **6. Prezentační vrstva** - má na starosti **konverzi dat**, přenášená data mohou totiž být v různých sítích různě kódována. Tato vrstva zajišťuje sjednocení formy vzájemně přenášených údajů. Dále data komprimuje, případně šifruje. V praxi často splývá s relační vrstvou.
- **5. Relační vrstva** - **navazuje** a po skončení přenosu **ukončuje spojení**. Může provádět **ověřování** uživatelů, **zabezpečení** přístupu k zařízením.
- **4. Transportní vrstva** - vrstva **zajišťuje přenos dat mezi koncovými uzly**. Jejím účelem je poskytnout takovou kvalitu přenosu, jakou požadují vyšší vrstvy. Vrstva nabízí spojové (TCP) a nespojové orientované (UDP) protokoly. (Platí pouze pro **TCP/IP**). PDU této vrstvy se nazývá **segment**.
- **3. Síťová vrstva** - je zodpovědná za spojení a **směrování mezi dvěma počítači nebo celými sítěmi** (tj. uzly), mezi nimiž neexistuje přímé spojení. Stará se o sítové adresování. Na této vrstvě pracuje router. PDU u této vrstvy se nazývá **paket**. Protokoly CLNP (OSI) a IP (TCP/IP).
- **2. Linková (spojová) vrstva** - poskytuje **spojení mezi dvěma sousedními systémy**. **Uspořádává data** z fyzické vrstvy do logických celků (neboli PDU) známých jako **rámce** (frames). Seřazuje přenášené rámce, stará se o nastavení parametrů přenosu linky, oznamuje neopravitelné chyby. Formátuje fyzické rámce, opatruje je fyzickou adresou a poskytuje synchronizaci pro fyzickou vrstvu. Na této vrstvě pracuje **switch**, tj vše je řízeno pomocí MAC adres. Protokoly jsou **Ethernet**, **TokenRing**, **FrameRelay**, **IEEE 802.11...**
- **1. Fyzická vrstva** - fyzická vrstva definuje všechny elektrické a fyzikální vlastnosti zařízení. Jakým signálem je reprezentována logická jednička, jak přijímací stanice rozezná začátek bitu, jaký je tvar konektoru, k čemu je který vodič v kabelu použit. Na této vrstvě pracuje rozbočovač (hub).

43.2 TCP/IP

Rodina protokolů TCP/IP (Transmission Control Protocol/Internet Protocol) obsahuje **sadu protokolů** pro komunikaci v počítačové síti a je hlavním protokolem celosvětové sítě **Internet**.

Komunikační protokol je množina pravidel, která určuje syntaxi a význam jednotlivých zpráv při komunikaci. Architektura TCP/IP je členěna do čtyř vrstev (na rozdíl od referenčního modelu OSI se sedmi vrstvami):

1. **Vrstva síťového rozhraní** (Network interface)
2. **Síťová (IP) vrstva** (Internet layer)
3. **Transportní vrstva** (Transport layer)
4. **Aplikační vrstva** (Application layer)



Komunikace mezi **stejnými vrstvami dvou různých systémů** je řízena **komunikačním protokolem** za použití spojení vytvořeného sousední nižší vrstvou. Architektura umožňuje výměnu protokolů jedné vrstvy bez dopadu na ostatní. Příkladem může být možnost komunikace po různých médiích fyzické vrstvy modelu OSI - ethernet (optické vlákno, kroucená dvojlinka, Wi-Fi, sériová linka).

43.3 Vrstva síťového rozhraní

Nejnižší vrstva umožňuje **přístup k fyzickému přenosovému médiu**. Je specifická pro každou síť v závislosti na její implementaci. Příklady sítí:

- **Ethernet** – název souhrnu technologií pro počítačové sítě (LAN, MAN) z větší části standardizovaných jako **IEEE 802.3**, které používají kably s **kroucenou dvoulinkou**, optické kably (ve starší verzích i koaxiální kably) pro komunikaci přenosovými rychlostmi od 1 Mbit/s po 100 Gbit/s.
- **Token ring** – technologie lokální sítě. Principem je předávání vysílacího práva pomocí speciálního rámce (tzv. tokenu) mezi adaptéry, zapojenými do logického kruhu.
- **FDDI** – síť kruhovou topologií (dva kruhy pro opačné směry přenosu), používá optické kably.
- **X.25** (později nahrazena Frame Relay), **SMDS**.

43.4 Síťová vrstva

Vrstva zajišťuje **především síťovou adresaci, směrování** a předávání datagramů (**packety**). Je implementována ve všech prvcích sítě - **směrovačích i koncových zařízeních**. Protokoly:

- **IP (Internet Protokol)** – základní protokol pro přenos dat po síti. Sám nezaručuje nic. Podle IP adres jen směruje paket k cíli.
- **ICMP (Internet Control Message Protocol)** – slouží pro odesílání chybových zpráv např. že služba není dostupná.
- **ARP (Adress Resolution Protocol)** – převádí IP adresy na MAC adresy
- **DHCP (Dynamic Host Configuration Protocol)** – požívá se pro automatickou konfiguraci počítačů připojených do sítě. Přiděluje například IP adresy.

43.4.1 Protokol IP (Internet Protocol)

Od nadřazených protokolů transportní vrstvy obdrží datové segmenty s požadavkem na odeslání. K segmentům připojí vlastní hlavičku a vytvoří IP datagram. V IP hlavičce je především IP adresa příjemce a odesilatele. IP protokol je **nespojový** (před zahájením výměny dat nevytváří relaci) a **nespolehlivý** (předání paketů na místo určení není kontrolováno). Paket IP se tedy může ztratit, být doručen mimo pořadí, zdvojen nebo zpožděn. Protokol IP neobsahuje prostředky pro zotavení z chyb tohoto typu. To vše má zajistit nadřízená transportní vrstva – protokol **TCP**.

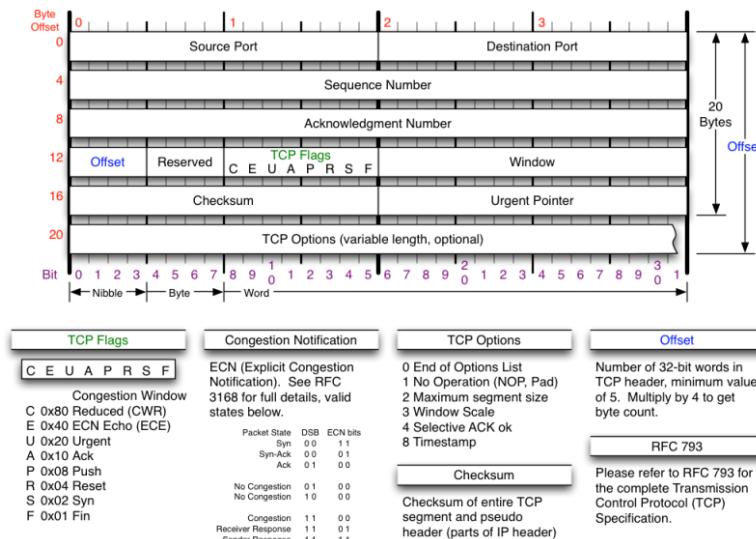
43.5 Transportní vrstva

Transportní vrstva je implementována až v **koncových zařízeních** (počítačích), a umožňuje proto přizpůsobit chování sítě potřebám aplikace. Poskytuje transportní služby kontrolovaným spojením **spolehlivým** protokolem **TCP** (transmission control protocol) nebo nekontrolovaným spojením **nespolehlivým** protokolem **UDP** (user datagram protocol).

43.5.1 TCP (Transmission Control Protocol)

Vytváří **virtuální okruh** mezi koncovými aplikacemi a zajišťuje tedy spolehlivý přenos dat.

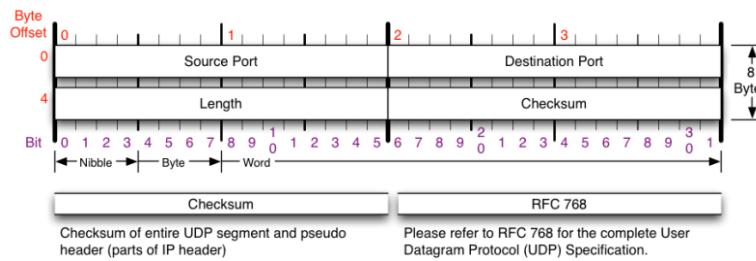
- Spolehlivá transportní služba, doručí adresátovi všechna data **bez ztráty** a ve **správném pořadí**.
- Služba se spojením, má fáze **navázání spojení** (3-way handshake), **přenos dat** a **ukončení spojení**.
- Transparentní přenos libovolných dat.
- **Plně duplexní spojení**, současný obousměrný přenos dat.
- Rozlišování aplikací pomocí portů.
- Komunikace je řízená pomocí **příznakových bitů** (ACK, SYN, atd.)



43.5.2 UDP (User Datagram Protocol)

Poskytuje **nespolehlivou** transportní službu pro takové aplikace, které nepotřebují spolehlivost, jakou má protokol TCP a jde jim čistě o **rychlost**. Nemá fázi navazování a ukončení spojení a už první segment UDP obsahuje aplikační data. UDP je používán aplikacemi jako je **DHCP**, **TFTP**, **SNMP**, **DNS** a **BOOTP**.

- Nespolehlivá transportní služba, **neověřuje** zda data došla v pořádku nebo ve správném pořadí.
- Nižší režie než u TCP (**rychlejší**).
- Zajištění spolehlivosti je na aplikacích vyšší vrstvy.
- Nemá fázi navázání a ukončení spojení, rovnou zasílá data.
- Hlavička UDP má pouze 4 části (délku, zdrojový/cílový port, checksum)



43.6 Aplikační vrstva

Jedná se přímo o programy (procesy), které využívají přenosu dat po síti ke konkrétním službám pro uživatele. Příklady: **Telnet** (TCP 23), **FTP** (TCP 20, 21), **HTTP** (TCP 80), **DHCP** (TCP/UDP 67, 68), **DNS** (TCP/UDP 53), **SSH** (TCP 22), **SMTP** (TCP/UDP 25), **POP3** (TCP 110), **IMAP** (TCP 143).

Aplikační protokoly používají vždy jednu ze dvou základních služeb transportní vrstvy: **TCP** nebo **UDP**, případně obě dvě (např. DNS). Pro rozlišení aplikačních protokolů se používají tzv. **porty**, což jsou domluvená číselná označení aplikací. Každé sítové spojení aplikace je jednoznačně určeno **číslem portu** a **transportním protokolem** (a samozřejmě adresou počítače).

44 Metody sdíleného přístupu ke společnému kanálu.

Metody sdílení přenosového kanálu se dělí na:

- **deterministické** (bezkolizní) – řídí se algoritmem, který přesně definuje kdo, kdy bude vysílat a ke kolizím nedochází.
- **nedeterministické** (kolizní) – algoritmus je založen na náhodě (náhodné časové prodlevy) a musí řešit kolize (situace, kdy chce naráz na kanálu vysílat více stanic). **Kolizní slot** udává, kolik se nejvýše času ztratí na nevyužití kanálu vlivem kolize.

44.1 Bezkolizní (deterministické) metody sdílení média

Definován jednoznačný algoritmus určující, v jakém pořadí mohou stanice na kanál přistupovat. Díky tomu na kanál nebude nikdy přistupovat více stanic současně. Deterministické metody se dělí na – **Centralizované řízení** a **Distribuované řízení**

44.1.1 Centralizované řízení - pooling

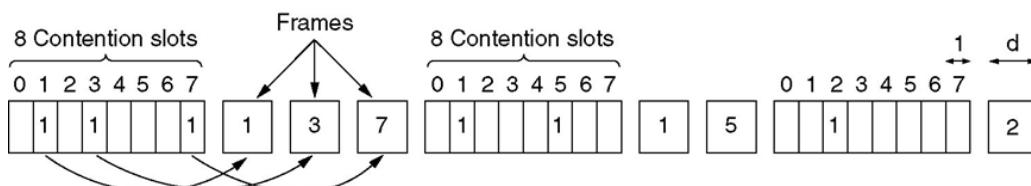
Jedna ze stanic je **master** a přiděluje ostatním právo vysílat. **Efektivní**, ale část kanálu použita pro komunikaci s masterem.

- **Přidělování na výzvu** - nejstarší, původně v terminálových systémech nad protokoly BSC a HDLC. Stanice smí vysílat jen, když je k tomu vyzvána masterem.
 - **Cyklická výzva** - vyzývaná stanice **bud' vyšle rámcem, nebo odmítne výzvu**, příp. neodpoví. Použitelné pro **malé zpoždění** na kanále. Rozumné pro vysoké využití kanálu. Pro malé zatížení a velký počet stanic neefektivní.
 - **Binární vyhledávání** - pro kanál, u kterého může stanice rozpoznat, **zda vysílá jedna nebo více stanic**. Při malém zatížení a velkém počtu stanic je efektivní vyhledávat stanici připravenou vysílat binárním vyhledáváním. Stanice se zorganizují do stromu podle jednotlivých bitů adres, řídící stanice postupně vyzývá skupiny stanic, aktivní stanice ve vyzvané skupině odpoví signálem na sdíleném kanále. Pokud stanice zjistí, že je jediná, která odpovídá, může začít vysílat, jinak se výzva posune o jednu úroveň dolů ve stromu. **Rychlejší pro malé zátěž**.
- **Přidělování na žádost** - žádosti od stanic přicházejí k řídící stanici po **vyhrazených kanálech**, vyhrazených kanálech se v klasických počítačových sítích LAN/WAN prakticky nepoužívá, spíše u sběrnic počítačů. Použití možné i v rádiových sítích, využití vyhrazeného **nízkorychlostního** podkanálu časového multiplexu.

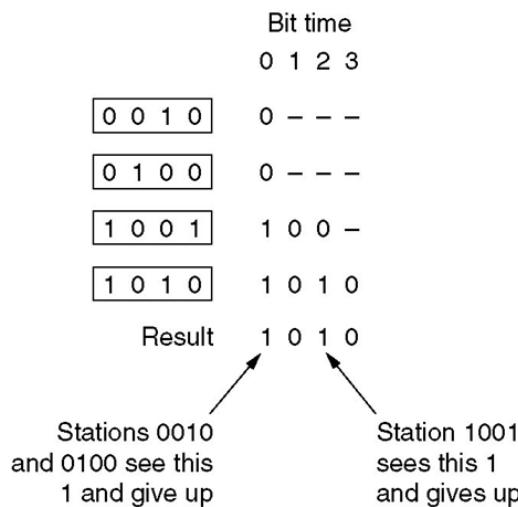
44.1.2 Distribuované řízení

Nezávislé na řídící stanici.

- **Rezervace kanálu** - rezervační rámec, ve svém slotu může každá stanice **požádat o přidělení slotu** v datovém kanále, datové sloty následují podle okamžitého požadovaného počtu za rezervačním rámcem. **Neefektivní pro velký počet stanic** na rozlehlé síti s malou zátěží. Na prázdném kanále neustálé opakování rezervačních slotů.



- **Binární vyhledávání** - stanice nejprve postupně vysílají bity své adresy od nejvyššího řádu, na sběrnici se logicky sčítají [OR]. Jakmile **stanice vysílá 0 a čte 1**, chce vysílat někdo s vyšší prioritou a stanice musí **zmlknout**.



- **Logický kruh [Token Passing Bus]** - adresy stanic tvoří cyklickou posloupnost, každá stanice zná svou adresu a adresu následníka. Mezi stanicemi se cyklicky předává právo k vysílání (token). Stanice vlastníci tokenu smí vysílat, do určité doby však musí předat token následníkovi. Problém počátečního ustavení posloupnosti, odpojování a připojování stanic do logického kruhu za provozu (rekonfigurace). Velké zpoždění při malé zátěži a velkém počtu stanic.
- **Virtuální logický kruh** - po odvysílání rámce je každý další stanici vyhrazen **časový interval**, kdy smí začít vysílat, nevyužije-li jej, následuje interval další stanice. Nutnost synchronizace stanic. V oblasti malých zátěží efektivnější než logický kruh.

44.2 Kolizní (nedeterministické) metody sdílení média

44.2.1 Aloha

Původně na rádiové sítě univerzity na Havajských ostrovech. **Netestuje se obsazenost média**, dodnes použití pro rádiové a družicové sítě, kde velké zpoždění signálu nebo konstrukce anténních obvodů zamezují příposlechu vlastního vysílání.

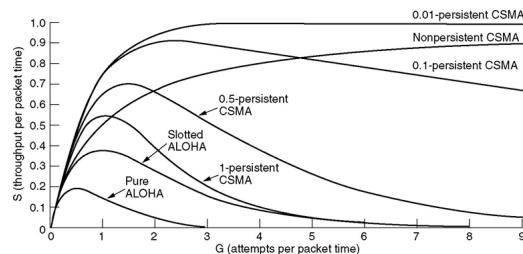
- **Prostá Aloha**
 - Stanice vysílají bez ohledu na cokoliv.
 - **Časový limit pro potvrzení**, při vypršení náhodné pozdržení (aby nedošlo k opakování kolizi) a opakování pokusu.
 - **Kolizní slot: $2*t_0$** [t₀-doba pro vyslání rámce]. **Dvojnásobek**, protože může být zarušen koncem jiného rámce. Kolizní slot udává, kolik času se nejvýše ztratí na nevyužití kanálu vlivem kolize.
- **Taktovaná Aloha**
 - Vysílat se smí začít jen v okamžicích začátků **časových úseků** pro odeslání jednoho rámce.
 - **Kolizní slot je poloviční** než u prosté Alohy.
 - **Exponenciální závislost**, malý vzrůst zátěže může významně zvýšit počet opakování a snížit průchodus kanálu.
- **Řízená Aloha**
 - **Řízená změna intenzity opakování** podle stavu sítě: vyšší intenzita opakování => rychlejší předání rámce, při blížícími se zablokování se intenzita sníží.
 - **Případně sledování provozu na kanále** [poměru neobsazených slotů] a nastavování intenzity.

44.2.2 Carrier Sense Multiple Access (CSMA)

Skupina metod **náhodného přístupu s příposlechem nosné**, tj. využití znalosti o obsazení kanálu. Podmínky pro aplikaci: **dokonalá slyšitelnost stanic, malé zpoždění signálu** (což platí v LAN).

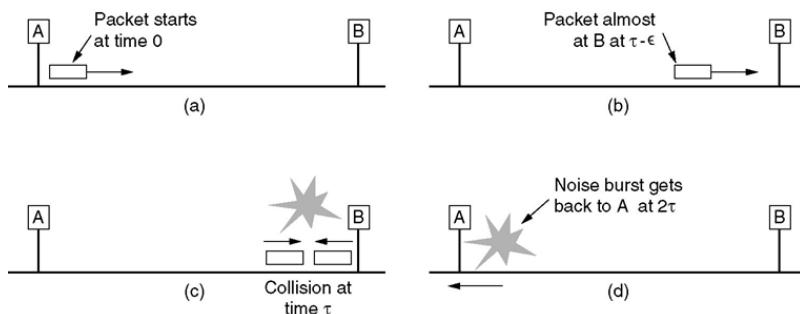
1. **CSMA (naléhanící CSMA, 1-persistent CSMA)** – před odesláním rámce se testuje stav kanálu, je-li obsazen, odloží se vysílání na okamžik jeho uvolnění. Riziko kolize stanic čekajících na uvolnění kanálu. Při obsazení kanálu čekání náhodnou dobu před dalším pokusem.

2. **Nenaléhající CSMA (non-persistent CSMA)** – při detekci obsazeného kanálu se počká náhodnou dobu pak opět test obsazení. Čekací doba se obvykle volí jako k-násobek doby průchodu signálu sběrnicí.
3. **P-naléhající CSMA (p-persistent CSMA)** – při obsazení kanálu v okamžiku potřeby vysílání se počká na uvolnění kanálu (nebo byl volný okamžitě), pak se s pravdě podobností p začne vysílat, s pravděpodobností $(1-p)$ se počká krátký interval, pak se opakuje do úspěšného odeslání rámce. Volbou p lze nastavit optimální využití kanálu pro danou zátěž. Pro $p=1$ jde o naléhající CSMA.



44.2.3 Carrier Sense Multiple Access with Collision Detection (CSMA/CD)

- CSMA s detekcí kolize (sledování vlastního vysílání).
- Využívá se u klasického **Ethernetu**.
- Před vysíláním musí být na médiu klid po dobu kolizního slotu.
- Jinak postup odpovídá naléhající CSMA.
- **Okamžité ukončení vysílání po detekci kolize** - kanál se zbytečně nezaplňuje rámcem, který je stejně zkolidován.
- Závislost maximální délky segmentu na **rychlosti šíření signálu** (vztah s dobou vysílání rámce a minimální délkou rámce)
- Nutnost kódování, které umožňuje detekci kolize (otevřený kolektor, měření napětí na médiu generovaného proudovými zdroji vysílačů).
- Při detekci kolize stanice vysílá kolizní signál **[jam]**, aby kolizi rozpoznaly všechny kolidující stanice. O opakování se stanice pokusí až po **náhodné časové prodlevě**, stabilitu metody nutno zajistit řízením intenzity opakování.



44.2.4 Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA)

Rezervace intervalu po odvysílání rámce pro potvrzení (v předchozích metodách jsme na potvrzení nahlíželi jako na přídavnou zátěž). Před začátkem vysílání paketu stanice určitý čas poslouchá, zda je přenosové médium volné. Pokud ano, oznámí vysílání a rezervuje přenosové médium a může zahájit vysílání. V opačném případě čeká na konec právě probíhajícího vysílání. Využití v **Wi-Fi**.

45 Problémy směrování v počítačových sítích. Adresování v IP, překlad adres (NAT).

45.1 Směrování

Směrování (routování) označuje v informatice určování cest datagramů v prostředí počítačových sítí. Směrování zajišťují nejen routery, ale i koncové stanice (při vysílání) a jejich úkolem je doručit datagram (resp. paket) adresátovi, pokud možno co **nejefektivnější cestou**. Směrování zajišťuje **sítová vrstva** modelu ISO/OSI a je využíváno v lokálních sítích LAN i na Internetu, kde jsou dnes směrovány zejména **IP packety**.

Sítová infrastruktura mezi odesílatelem a adresátem paketu může být velmi složitá, a proto se směrování zpravidla nezábývá celou cestou paketu, ale řeší vždy **jen jeden krok**, tj. komu datagram předat jako dalšímu. Hledání cest v síti:

- Ve **spojoře orientovaných sítích** (jednobodové, vícebodové) při vytváření spojení:
 - Nastavování **spojovacích polí přepínacích prvků** na cestě.
 - Budování **přepínacích tabulek** virtuálního kanálu.
- V sítích s **přepínáním paketu** (obvykle obecně polygonálních a s alternativními cestami) při přenosu každého jednotlivého paketu => **každý paket může jít jinou cestou**.

45.1.1 Směrování v počítačových sítích a v Internetu

Abychom mohli paketovou síť směrovat pakety od zdroje k cíli, potřebujeme správným způsobem naplnit **směrovací tabulky** všech směrovačů na trase. V malých sítích nebo v sítích, z nichž veškerý provoz ven odchází po jediné implicitní (default) cestě, toto lze vyřešit manuálním vložením potřebných informací, tj. **statickým směrováním**. V rozlehlejších sítích s měnící se topologií (z nichž největší je bezesporu Internet) jsou však nutné **dynamické směrovací protokoly**, které zajistí správné naplnění směrovacích tabulek automaticky na základě výměny informací mezi směrovači.

45.1.2 Neadaptivní – Statické směrování

- **Směrovací tabulky** v jednotlivých směrovačích **konfigurovány "ručně"** - pracnější.
- Odpadá **režie směrovacích protokolů** (zabraná šířka pásma, čas na zpracování).
- **Bezpečnější** (omezení možnosti generování falešných směrovacích informací, odposlouchání topologie sítě).
- Při **výpadku** linky nutný **ruční zásah**.
- Použitelné, pokud se topologie **příliš často nemění** (vlivem výpadků a modifikací sítě).
- V intranetech používáno až překvapivě často.

45.1.3 Adaptivní – Dynamické směrování

- **Automaticky reaguje** na poměry v síti (topologie, zátěž, ...).
- Nutnost **provozu směrovacích protokolů**.
- Užitečné **při častých změnách** (příp. i obecně neznámé) topologie sítě (typicky v Internetu). Algoritmy **DVA** a **LSA**.
- V praxi často používána **kombinace statického a dynamického směrování**, staticky nakonfigurované cesty mají obvykle přednost.

45.1.4 Hierarchické směrování, autonomní systémy

Současný Internet je natolik **rozsáhlý** a **proměnlivý**, že není reálné udržovat ve směrovačích úplnou informaci o jeho topologii. Tato informace by navíc byla velmi nestabilní, protože by se měnila s výpadkem nebo zapojením linky kdekoli na světě. Proto bylo rozhodnuto směrování v Internetu řešit **hierarchickým způsobem**. Předpokladem jeho použití je rozdělení Internetu do tzv. **autonomních systémů (AS)**. Autonomním systémem rozumíme souvislou skupinu sítí a směrovačů, které jsou pod společnou správou a řídí se společnou směrovací politikou. Pod společnou směrovací politikou si představme zejména dohodnutý vnitřní směrovací protokol (např. OSPF nebo RIP), ale také speciální požadavky administrátorů na směrování některých druhů

provozu (traffic engineering, load balancing). Příkladem autonomního systému tak může být autonomní systém jednoho konkrétního **poskytovatele Internetu (ISP)** nebo velké firmy.

Cílem hierarchického směrování je **vždy nejprve doručit paket** určený pro **některou ze sítí autonomního systému** na hranice tohoto autonomního systému. O další směrování ke konkrétní síti uvnitř AS se již postará vnitřní směrovací protokol, který topologii (nebo alepoň cesty ke všem sítím) svého vlastního AS zná. Směrovač, který je na hranici autonomního systému a účastní se jak na směrování mezi AS tak ve směrovacím protokolu svého AS, se nazývá **hraniční směrovač** (angl. border gateway). V případě neznámé cesty, cesta putuje **default cestou**, která jej pošle nadřazenému AS a ten se o její zpracování dále postará.

45.1.5 Vnitřní a vnější směrovací protokoly

Při směrování v rámci jednotlivých autonomních systémů se používají tzv. **vnitřní směrovací protokoly** - Interior Gateway Protocols, IGP. Naopak pro směrování mezi autonomními systémy se používají **vnější směrovací protokoly** - Exterior Gateway Protocols, EGP. Typickými vnitřními směrovacími protokoly jsou dnes např. **OSPF** nebo starší **RIP**, jako vnější směrovací protokol se používá téměř výhradně protokol **BGP**.

45.2 Směrovací algoritmy

Rozlišujeme dvě základní třídy směrovacích protokolů: protokoly založené na **vektoru vzdálenosti - DVA** (distance-vector) a na **stavech linek - LSA** (link-state).

45.3 DVA - Distance Vector

Sousední směrovače si navzájem vyměňují své směrovací tabulky a doplňují si informace, které se naučí od sousedů, v určitých časových intervalech. **Topologii** celé sítě však **neznají**, musí se spokojit s adresami sousedů, přes která mají posílat pakety do jednotlivých cílových sítí a **vzdálenostmi** k těmto sítím, které společně tvoří tzv. **distanční vektory**.

- Na začátku směrovací tabulka obsahuje **pouze přímo připojené sítě** (staticky nakonfigurováno administrátorem).
- Směrovače poté **periodicky zasílají** směrovací tabulky sousedům.
- Z došlých **směrovacích tabulek sousedů** (vzdáleností sousedů od jednotlivých sítí) si směrovač postupně **upravují a budují** svou směrovací tabulku.
- Pokud cesta **nebyla delší dobu sousedem inzerována**, ze směrovací tabulky se **odstraní**.

Vlastnosti

- **Metrikou je počet "přeskoků"** (hop count) na cestě mezi zdrojem a cílem – nezohledňuje parametry jednotlivých linek (přenosovou rychlosť, zpoždění, okamžitou zátěž, ...).
- **Pomalá konvergence při změnách** topologie – o změně se informuje až při příštím periodickém broadcastu směrovací tabulky.
- **Zátěž** od broadcastu celých směrovacích tabulek.
- **Příliš "optimistické"** – směrovač se rychle učí dobré cesty, ale **špatně zapomíná při výpadcích**.
 - **Čekání** na timeout cesty, která přestala být inzerována.
 - Žádný směrovač nikdy nemá metriku horší, než minimum z metrik sousedů + 1 => pomalé šíření špatných zpráv.

45.3.1 Routing information protocol (RIP)

- Velmi starý, stále však **často implementovaný** v malých sítích.
- Hodnota "nekonečna" řešící **problém počítaní do nekonečna** a v technice Poisson reverse reprezentována číslem 16..
- **Maximální počet** přeskoků je omezen na **16**.

45.3.2 Interior Gateway Routing Protocol (IGRP)

- Cisco proprietary.
- Lepší metrika, než pouhý počet přeskoků **bandwidth**, **delay**, **(reliability, load)**.
- Není omezení na 16 přeskoků (zvýšeno na 255).

45.4 LSA - Link State

- Směrování na základě **znalosti "stavu"** jednotlivých linek sítě (funkčnost, cena).
- Směrovače (uzly grafu) **znají topologii celé sítě** (graf) a **ceny jednotlivých linek** (ohodnocení hran). Tyto informace udržují v topologické databázi. (Všechny směrovače mají stejnou topologickou databázi).
- **Každý směrovač počítá strom nejkratších cest** ke všem ostatním směrovačům (a k nim připojeným sítím) pomocí **Dijkstrova algoritmu**. (Na rozdíl od DVA všechny směrovače počítají na základě **stejných a úplných dat**)
- Každý směrovač **neustále sleduje stav a funkčnost** k němu připojených linek.
- Testováním linek k sousedním směrovačům – **Hello protokol**.
- Při **změně okamžitě šíří informaci** o aktuálním stavu svého okolí všem ostatním směrovačům. Ty si ji vloží do topologické databáze.
- **Šíří se pouze změny** (ale do celé sítě) - žádné periodické rozesílání směrovacích tabulek.
- **Okamžitá reakce na změnu stavu linek** (výpadek, náběh) => rychlá konvergence
- Zástupci: **OSPF (Open Shortest Path First)**

45.5 Adresování

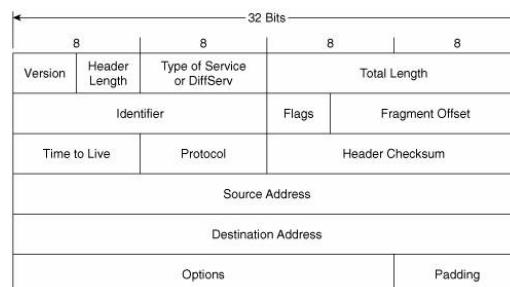
Adresování sítě přiděluje oblastní správce (pro Evropu **RIPE**). IP adresa je **logická adresa** zařízení v počítačové síti (na 3. vrstvě podle OSI modelu), která používá IP protokol. V IPv4 je tato adresa velká 32bitů (4 byte) a zapisuje se pomocí **dot-decimal notation**. To znamená pomocí čtyř dekadických hodnot (každá velká jeden byte), označovaných jako oktet, oddělených tečkou. Příkladem IP adresy je 193.222.5.15.

IP adresa se skládá z několika částí. V základu jde o dvě části, **prefix** identifikující síť a **adresa uzlu** v rámci podsítě.

Pro to, abychom určily, která část IP adresy je pro **podsíť** a která pro **uzel**, se používá **maska podsítě** (subnet mask). Ta v binárním tvaru obsahuje jedničky následované nulami a pomocí jedniček, vymaskovává „část síťového prefixu v IP adrese. Jinak řečeno, tam kde jsou v masce jedničky je v IP adrese část podsítě a kde jsou nuly je část uzlu. Síťová část IP adresy určuje podsít a používá se ke směrování. Část uzlu identifikuje stanici uvnitř daného subnetu.

Maska se v IPv4 zapisuje stejně jako IP adresa pomocí dot-decimal form s tím, že jsou validní pouze adresy, které v binárním zápisu mají zleva jedničky následované nulami (za první nulou smí být pouze nuly). Příkladem síťové masky je 255.255.255.0. Maska 255.255.255.254 není platná, protože uvnitř takového subnetu by se nacházelo 0 uzlů. Maska 255.255.255.255 není maskou podsítě, ale určuje jeden uzel.

Občas se používá speciální zápis, který se označuje jako **inverse mask** nebo **wildcard mask**. Jedná se o obrácenou masku, jednoduše řečeno tam, kde jsou v tradiční masce jedničky jsou nuly a naopak. Pro výpočet v dekadické formě můžeme použít pro každý oktet hodnota = 255 – hodnota oktetu. Například pro masku 255.255.255.240 je inversní maska 0.0.0.15.



Obrázek 45.1: Hlavička IPv4

45.5.1 Veřejné a privátní adresy

Prvotní princip IP sítí byl takový, že IP adresa musela být unikátní v rámci celé sítě a jednotlivé uzly (stanice, servery a další zařízení s přiřazenou IP adresou) spolu mohly komunikovat. S rozvojem internetu se ukázalo, že **počet adres**, které je možno v IPv4 vytvořit, rozhodně **není dostatečný**. Začaly se tedy používat různé metody, jak se s nedostatkem adres vypořádat. Nejrazantnější je nová verze IP protokolu IPv6, která obsahuje mnohem více adres, ale její nasazování není jednoduché. Dále se objevila technika CIDR (Classless Inter-Domain Routing, tj. „beztrídní směrování“) a také se IP adresy rozdělily na dva typy, na veřejné a neveřejné IP adresy.

Veřejné IP adresy (public address) tvoří hlavní část adresního rozsahu internetu a tyto adresy jsou rountovatelné v rámci celého internetu. Jednoduše řečeno počítač s veřejnou adresou může být dostupný z celého internetu. Tyto adresy musí být unikátní v celé síti (internetu).

Oproti tomu privátní IP adresy (private address) by se měly používat pouze v rámci LAN sítí a v internetu by přes ISP neměli komunikovat. Firma pak používá jednu (nebo více) veřejnou adresu a pomocí techniky NAT (Network Address Translation) se při komunikaci mimo LAN překládají privátní adresy na tuto veřejnou. Stejně privátní adresy se tak mohou nacházet na mnoha místech v internetu, ale nemohou spolu přímo komunikovat. Následující tabulka ukazuje jednotlivé privátní rozsahy.

sítě	adresa sítě	broadcast adresa	adresy uzlů
10.0.0.0/8	10.0.0.0	10.255.255.255	10.0.0.1 – 10.255.255.254
192.168.0.0/16	192.168.0.0	192.168.255.255	192.168.0.1 – 192.168.255.254
172.16.0.0/12	172.16.0.0	172.31.255.255	172.16.0.1 – 172.31.255.254

IP adresy se dělí na:

- **unicast** – adresa jednoho konkrétního počítače,
- **multicast** – adresa pro více počítačů najednou,
- **broadcast** – adresa na všechny počítače. Šíří se jen v rámci segmentu počítače, dál není propuštěna (255.255.255.255).
- **loopback** – zpětnovazební adresa, pošle paket zpátky na vlastní počítač (127.0.0.1).

45.6 IPv6

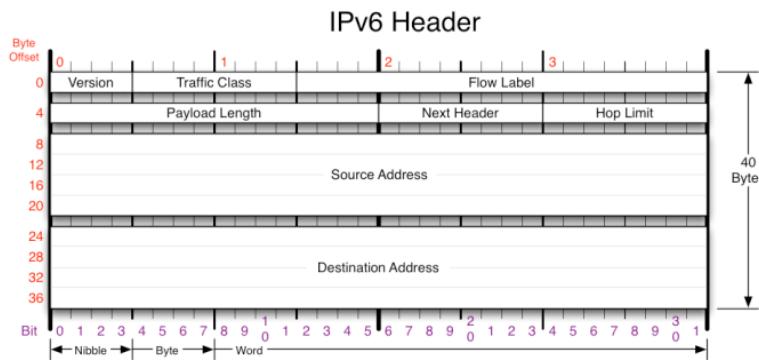
IPv6 nahrazuje dosluhující protokol IPv4. Přináší zejména **masivní rozšíření adresního prostoru** a zdokonalení schopnosti přenáset **vysokorychlostné data**. Starší protokol IPv4 poskytuje omezený adresní prostor – teoreticky 2^{32} adres. IPv6. Obsahuje celkem 2^{128} adres. Většina přenosových a aplikačních vrstev protokolů vyžaduje malé nebo žádné změny pro funkčnost s IPv6. Výjimkami jsou protokoly aplikací zahrnující adresy síťové vrstvy např.: FTP. **Multicast** je součástí základní specifikace IPv6 na rozdíl od IPv4, kde byl zaveden později. IPv6 nepoužívá **broadcast** na místní linku. Každá adresa má **128b**. **Odstraněna potřeba NAT**. Protokol pro IP vrstvu šifrování a autentizaci **IPsec** je integrální součástí souboru protokolů IPv6, na rozdíl od IPv4, kde je přítomen volitelně (obvykle ale implementován).

IPv6 adresy se obvykle zapisují jako osm skupin čtyř hexadecimálních číslic: **2001:0db8::1428:57ab**. Vzhledem k zdlouhavému zápisu se může **nejdelší sekvence nul nahradit ::, 4 nuly můžeme nahradit jednou**.

45.6.1 IPv6 Packet

Paket IPv6 se skládá ze dvou hlavních částí: hlavičky a těla.

- **Verze** - 4 bity, verze 6
- **Dopravní třídu** - 8 bitů na prioritu paketu. Úroveň priority se dělí na rozsahy: kde zdroj podporuje kontrolu přetížení a bez podpory kontroly přetížení.
- **Pojmenování toku** - 20 bitů pro správu QoS. Původně určeno pro speciální obsluhu aplikací reálného času, nyní se nepoužívá.
- **Délka těla** - 16 bitů pro délku těla paketu. Při vynulování se nastaví „jumbo“ tělo (skok za skokem).
- **Následující hlavička** - 8 bitů, určuje další vnořený protokol. Hodnoty se shodují s hodnotami definovanými pro IPv4.
- **Zdrojová a cílová adresa** - 128 bitů na každou adresu.
- **Hop limit** - 8 bitů, číselně definuje počet povolených přechodů síťovými prvky. Každý přechod znamená snížení čísla o 1.



45.7 Překlad síťových adres NAT

NAT (Network address translation) se většinou používá pro přístup více počítačů z **lokální sítě** na **Internet** pod **jedinou veřejnou adresou**. Překládá zdrojovou a cílovou IP adresu, je realizován na **routerech, firewallech** většinou zařízeních 3 vrstvy. Umožňuje připojit více počítačů na jednu veřejnou IP adresu - řeší se tak nedostatek přidělených veřejných IP adres. Využívá se **překladové tabulky**.

Výhody:

- Zvyšuje bezpečnost počítačů připojených za NATem (potenciální útočník nezná opravdovou IP adresu).
- Umožňuje připojit více počítačů na jednu veřejnou IP adresu, čímž se obchází nedostatek IPv4 adres

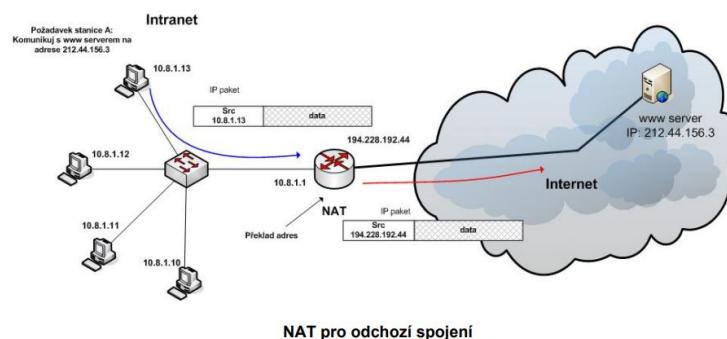
Nevýhody:

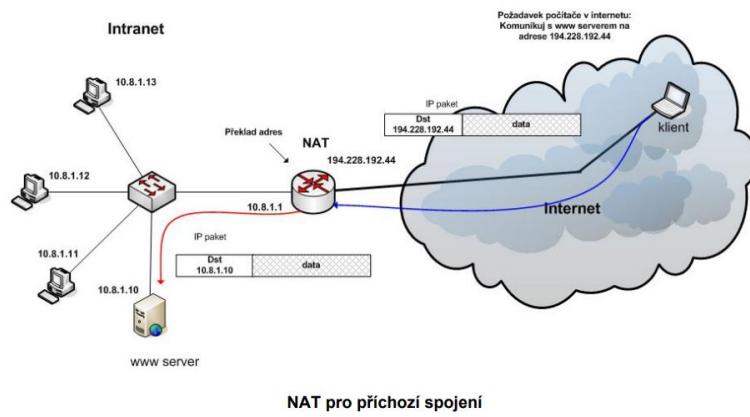
- Zařízení za NATem nemají skutečné připojení k Internetu (není možné se snadno připojit na zařízení za NATem.)
- NAT znemožňuje správnou funkcionalitu některých software

45.7.1 Typy NATu

Typicky se využívá kombinace obou níže zmíněných řešení.

- **Statický NAT** - Překladová tabulka je konfigurována manuálně administrátorem.
- **Dynamický NAT** - Obsah překladové tabulky je vytvářen dynamicky v závislosti na síťovém provozu. Veřejné adresy jsou alokovány jednotlivým spojením jejich vypůjčením z **NAT Poolu**.
- **Network address and port translation NAPT** - Několik uzlů využívá pouze jednu veřejnou IP adresu. Jednotlivé uzly jsou identifikovány pomocí různých čísel portů.

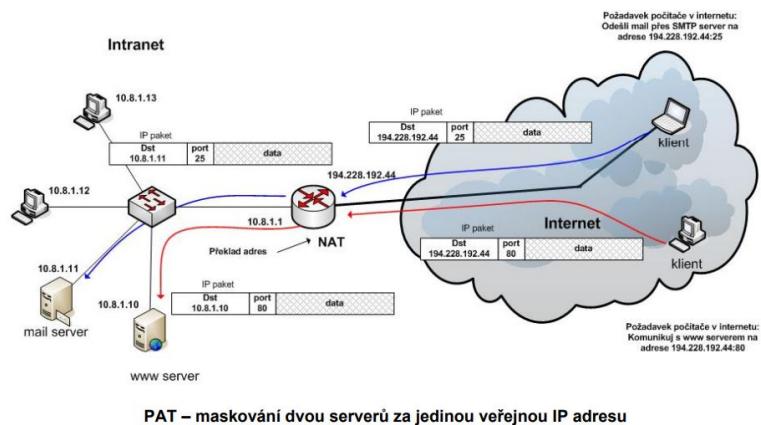




NAT pro příchozí spojení

45.7.2 PAT (Port Address Translation)

V případě PAT dochází k překladu jak adres tak i příslušných portů v IP komunikaci. Výhodou zde je, že za jednu venkovní IP adresu můžeme maskovat celou řadu služeb, které jsou hostovány na různých serverech. Typickým příkladem jsou FTP a www servery umístěné v DMZ.



PAT – maskování dvou serverů za jedinou veřejnou IP adresu

46 Bezpečnost počítačových sítí s TCP/IP: útoky, paketové filtry, stavový firewall. Šifrování a autentizace, virtuální privátní sítě.

46.1 Ve spojení s bezpečností v počítačových sítích definujeme tyto pojmy

- **Utajení** (confidentiality) – posluchač na kanále datům nerozumí.
- **Autentizace** (authentication) – jistota, že odesílatel je tím, za koho se vydává.
- **Integrita** (integrity) – jistota, že data nebyla na cestě zmodifikována.
- **Nepopiratelnost** (non-repudiation) – zdroj dat nemůže popřít jejich odeslání.

46.2 Útoky

Cílem všech typů útoků je nějakým způsobem ublížit uživateli, často ve formě: zahlcení sítě pakety, narušení konfigurace nastavení, zabránění uživateli v přístupu ke službě, **zatížení CPU**, pád OS. Typy útoků:

- **ARP dotazy** – falšování ARP odpovědí (falešný překlad IP-to-MAC). **Oprava** užitím statických ARP záznamů.
- **Routing protocol** – falšování routovacích informací propagovaných routovacím protokolem (RIP atd). Oprava **filtrováním** zdrojů routovacích informací.
- **Switchované sítě** – proti přetížení přepínací tabulky. Oprava užitím limitování počtu MAC na portu, statickým listem povolených MAC.
- **Brute Force** – zadávání hesel pomocí hrubé sily.
- **Denial of service (DoS)** - cílem útočníka vyčerpání systémových prostředků (paměť, CPU, šířka pásma) sítového prvku nebo serveru a jeho zhroucení nebo změna požadovaného chování.
 - **Smurf** – zahlcení cíle ICMP pakety (ping), jejich zpracování mívá někdy přednost před běžným provozem; útočník pošle žádost o ping všem (broadcast) a jako odesílatele uvede cíl útoku. **Řešení:** packetové filtry.
 - **SYN flood** – neustálé navazování TCP spojení (příznak SYN), server alokuje prostředky a pošle (SYN-ACK) a čeká na odpověď, které se nedočká. **Řešení:** zkrácení doby čekání na odpověď.
- **Distributed DoS (DDoS)** – DoS útok je vedený z mnoha stanic, které byly již před tím napadeny a nyní jsou využity k tomuto útoku. Je obtížně blokovatelný kvůli přístupu mnoha stanic.

46.3 Filtrování provozu

46.3.1 Paketové filtry (nestavové)

Filtrování probíhá dle informací v **hlavičce 3 a 4 vrstvy**. Pravidla udávají, ze které adresy a portu na kterou adresu a port může být paket procházející rozhraním routeru propuštěn. Na routerech CISCO je realizován jako **Access Control List (ACL)** prostřednictvím sekvence záznamu, které povolují/zakazují přenos paketu, které odpovídají daným kritériím. Samotný paketový filtr je rychlý, nenáročný na systémové zdroje, ale úroveň kontroly je relativně malá. **Reflexivní ACL** - automaticky propouští vstupní provoz, který odpovídá povolenému provozu výstupnímu.

46.3.2 Stavový firewall

(též stavový paketový filtr, anglicky stateful firewall) odděluje důvěryhodnou (interní) síť od nedůvěryhodné (externí) sítě. Funguje stejně jako jednoduchý paketový filtr, ukládá **navíc ale i informace o povolených spojeních**, podle kterých pak může rozhodovat, zda procházející pakety patří do již povoleného spojení a mohou být propuštěny nebo zda musí znova projít kontrolou. Firewall je **velmi rychlý**, poskytuje slušnou úroveň zabezpečení a snazší konfiguraci. Poskytuje urychlené zpracování paketu již povoleného spojení. Obdobou stavového firewallu je **nestavový firewall**, který se rozhoduje pouze na základě informací obsažených v konkrétním paketu (pracuje na nižší síťové vrstvě ISO/OSI modelu) a aplikační firewall, který pracuje naopak na vyšší síťové vrstvě.



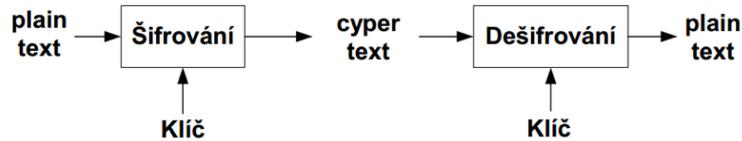
46.4 Virtuální privátní síť

(zkratka VPN, anglicky virtual private network) je v informatice prostředek k **propojení několika počítačů prostřednictvím (veřejné) nedůvěryhodné počítačové sítě**. Lze tak snadno dosáhnout stavu, kdy spojené počítače budou mezi sebou moci komunikovat, jako kdyby byly propojeny v rámci jediné uzavřené privátní (a tedy důvěryhodné) sítě. Při navazování spojení je totožnost obou stran ověřována pomocí **digitálních certifikátů**, dojde k autentizaci, vytvoření duplexního kanálu, veškerá komunikace je šifrována, a proto můžeme takové propojení považovat za bezpečné.

46.5 Šifrování

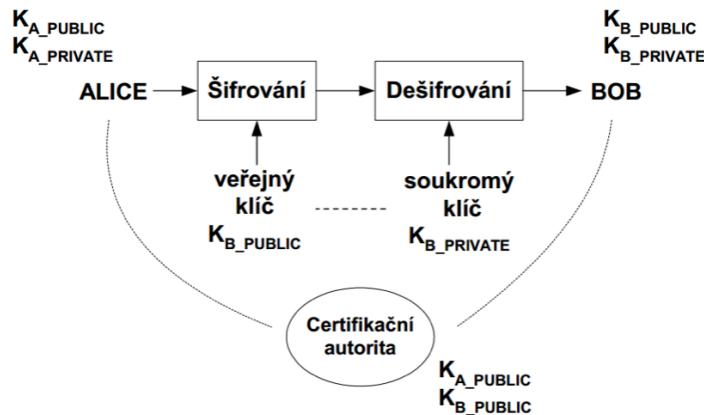
46.5.1 Symetrické šifrování

Pro šifrování i dešifrování se používá **pouze jeden klíč**, který musí mít všichni účastníci, kteří chtějí data šifrovat nebo dešifrovat. **Nebezpečí hrozí při distribuci** tohoto klíče, který pokud bude prozrazen tak všichni účastníci musí začít používat nový klíč. Šifrování je **rychlé** a **jednoduché** pro implementaci (možná implementace přímo v HW). Nejznámější **AES**, **DES**, **3DES**, **IDEA**. **DES** již dnes není bezpečný, používá se **3DES**, který **DES** zašifruje 3x po sobě.



46.5.2 Asymetrické šifrování

Podstatou je **generace dvou šifrovacích klíčů**, které spolu spolupracují. Tyto klíče se vzájemně matematicky doplňují a je možné oba použít jak pro dešifrování tak šifrování. Veřejný klíč je volně šířitelný pro všechny, kteří chtějí šifrovat posílaná data. Soukromý klíč je tajný a je pouze pro potřebu pro dešifrování toho, co bylo zašifrováno veřejným klíčem. Uživatel tedy pro šifrovanou komunikaci potřebuje oba klíče. Výhodou je, že **není třeba veřejný klíč speciálně ukrývat** (je vyřešena metoda distribuce), pouze je třeba zajistit mechanizmus proti modifikaci veřejných klíčů při přenosu - **certifikovaná autorita (CA)** klíč digitálně podepisuje. Nevýhodou je, že v porovnání se symetrickým je **pomalejší** z důvodu použitých matematických funkcí. Asymetrické systémy jsou např. **RSA** nebo **DSA**. Stupeň bezpečnosti se **odvíjí od délky** použitého klíče.



46.5.3 Certifikační autorita

- Entita, které je důvěrováno.
- Registruje (podepsané) veřejné klíče.
- První kontakt s certifikační autoritou **musí proběhnout osobně** (získání dvojice podepsaný veřejný-privátní klíč).
- Veřejný klíč certifikační autority musí být **důvěryhodným** způsobem zaveden do každého systému.

46.5.4 Autentizace

Je proces ověření proklamované identity subjektu. Probíhá nejčastěji jednou ze tří metod:

- Řekneme něco, co známe (**heslo**, PIN)
- Ukážeme něco, co máme (ID karta, **hardwareový klíč**)
- Necháme systémem změřit něco našeho (**biometrické údaje**, otisk prstů, sítnice)

Existuje **několika stupňové ověření**, například kombinací PIN a biometrických údajů. Po ověření identity následuje autorizace což je souhlas k provedení operace či umožnění přístupu. **Často se používá v souladu s šifrováním:**

- U **symetrického** odesílatel šifruje uživatelské jméno sdíleným klíčem a příjemce kontroluje platnost tohoto uživatelského jména.
- U **asymetrického** se užívá digitálních podpisů Certifikační autority.

46.6 Zabezpečení na jednotlivých vrstvách OSI-RM

- L7** – S/MIME,
- L4** – SSL (jen TCP),
- L3** – IPSec (bezpečnostní rozšíření IP protokolu založené na autentizaci a šifrování každého IP datagramu) **nezávislé na aplikaci**,
- L2** – hop-by-hop, neefektivní.

46.6.1 SSL/TLS

Secure Sockets Layer, SSL (doslova vrstva bezpečných socketů) je protokol, resp. **vrstva vložená mezi vrstvu transportní** (např. TCP/IP) a **aplikáční** (např. HTTP), která **poskytuje zabezpečení komunikace** šifrováním a autentizací komunikujících stran. Protože SSL již je ve všech verzích **deprecated**, mělo by se používat pouze **TLS** neboli Transport layer security.

Protokol SSL se nejčastěji využívá pro bezpečnou komunikaci s internetovými servery pomocí HTTPS, což je zabezpečená verze protokolu HTTP. Po vytvoření SSL spojení (session) je komunikace mezi serverem a klientem šifrovaná a tedy zabezpečená.

Ustavení SSL spojení funguje na principu **asymetrické šifry**, kdy 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.