Obsah

Matematické základy informatiky

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

Konečný automat je výpočetní model který přijímá nebo zamítá slova regulárního jazyka. Je definován jako uspořádaná pětice $A = (Q, \sum, \delta, q_0, F)$ kde:

- Q je neprázdná množina stavů
- $\sum$ je množina symbolů které se mohou objevit ve vstupu (*abeceda*)
- $\delta : Q \times \sum$ je přechodová funkce (vyjádřitelná např. tabulkou)
- $q_0 \in Q$ je počáteční stav
- $F \subseteq Q$ je množina přijímajících stavů

Automat začne v počátečním stavu, a poté zpracovává vstupní řetězec symbol po symbolu. Pro každý symbol automat vyhledá pomocí přechodové funkce do kterého stavu má přejít. Vstup je *přijat* pokud po posledním přečteném symbolu je automat v jednom z přijímajících stavů.

Nedeterministický konečný automat slouží k zjednodušení návrhu konečných automatů. Je definován jako uspořádaná pětice $A = (Q, \sum, \delta, I, F)$ kde:

- Q je neprázdná množina stavů
- $\sum$ je množina symbolů které se mohou objevit ve vstupu (*abeceda*)
- $\delta : Q \times \sum \to \mathcal{P}(Q)$ je přechodová funkce
- $I \subseteq Q$ je množina počátečních stavů
- $F \subseteq Q$ je množina přijímajících stavů

Graf automatu:

- Vrcholy jsou stavy ($\in Q$)
- Příchozí šipka označuje počáteční stav (q_0)
- Odchozí šipka nebo dvojitý kroužek označuje přijímající stavy ($\in F$)
- Hrana mezi vrcholy (stavy) q a $q\prime$ se zaznačenými symboly a, b, c ($\in \sum$) označuje přechody $\delta(q, a) = \delta(q, b) = \delta(q, c) = q\prime$

Vlastnosti konečných automatů:

- Dva automaty jsou jazykově ekvivalentní $\equiv$ přijímají shodné jazyky
- Automat je minimální $\equiv$ k němu neexistuje ekvivalentní automat s menším počtem stavů (tj. minimální automat nemá nedosažitelné stavy a je redukovaný)
- Dva minimální automaty jsou izomorfní $\equiv$ mají stejný normovaný tvar

Regulární jazyk je nejjednodušší formální jazyk. Může být popsán regulárním výrazem nebo konečným automatem, a generován regulární gramatikou. Pomocí $q \xrightarrow{w} q\prime$ značíme, že automat přejde přečtením slova w ze stavu q do stavu $q\prime$. Slovo $w \in \sum *$ je přijímáno automatem jestliže $q_0 \xrightarrow{w} F$. Jazyk rozpoznávaný automatem A je množina všech slov přijímaných automatem A, neboli $L(A) = \{w \in \sum * \mid q_0 \xrightarrow{w} F\}$. Jazyk je regulární $\equiv$ množina kvocientů $\{w\backslash L \mid w \in \sum *\}$ je konečná.

Regulární výraz popisuje podmnožinu řetězců dané abecedy.

Regulární gramatika obsahuje pouze pravidla A $\rightarrow$ a a pak buď A $\rightarrow$ aB (levá) nebo A $\rightarrow$ Ba (pravá) ale ne levá i pravá pravidla najednou.

Uzavřenost operací nad regulárními jazyky lze dokázat tak, že ke každé operaci definujeme proces sestrojení konečného automatu který přijme výsledný jazyk.

Myhill-Nerodova věta je nutnou i postačující podmínkou pro důkaz regularity jazyka.

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

Bezkontextová gramatika popisuje bezkontextový jazyk.

Bezkontextový jazyk je formální jazyk jehož slova rozpoznává zásobníkový automat.

Zásobníkový automat přijme slovo pokud po přečtení vstupu skončí v přijímacím stavu. Pokud bychom zároveň chtěli aby byl zásobník prázdný, můžeme provést transformaci:

1. Zaveď nový počáteční stav. Ten povede do původního počátečního stavu, a přidá na zásobník počáteční zásobníkový symbol (dle definice zásobníkového automatu).
2. Definuj nový symbol který se v abecedě zásobníkových symbolů nevyskytuje (např. \$) a použij jej jako počáteční zásobníkový symbol.
3. Každý přijímající stav změň na nepřijímající, a veď z něj přechod do nového přijímajícího stavu, který na zásobníku bude očekávat nově definovaný symbol, a tím zaručí že zásobník je — vyjma nově definovaného symbolu — prázdný.

Redukce bezkontextové gramatiky probíhá následujícím postupem:

1. Sestroj množinu $\mathcal{T}$ obsahující neterminály, které lze přepsat pouze na terminály a/nebo neterminály které již jsou v množině $\mathcal{T}$.
2. Odstraň všechna pravidla obsahující neterminály které nejsou v množině $\mathcal{T}$.
3. Sestroj množinu $\mathcal{D}$ obsahující počáteční neterminál a dále všechny neterminály z něj dosažitelné.
4. Odstraň všechna pravidla obsahující neterminály které nejsou v množině $\mathcal{D}$.

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

Turingovy a RAM stroje jsou výpočetní modely pomocí kterých lze popsat jakýkoliv algoritmus.

Turingovská úplnost označuje stroj nebo jazyk se stejnou výpočetní silou jakou má Turingův stroj.

Lineárně ohraničený Turingův stroj má velikost pásky omezenou podle lineární funkce jejíž vstupem je velikost vstupu. Tento typ stroje umí zpracovat kontextové jazyky.

Složitost algoritmu se používá pro srovnání časové nebo paměťové efektivity algoritmů. Udává se funkcí která vyjadřuje závislost potřebného času/paměti na velikosti vstupu. Máme-li funkci g která přesně popisuje reálný růst algoritmu v závislosti na velikosti vstupu, značíme:

- $O(f)$ pokud funkce f ohraničuje g shora, tj. algoritmus nemůže být pomalejší než jak udává f
- $\Omega(f)$ pokud funkce f ohraničuje g zdola, tj. algoritmus nemůže být rychlejší než jak udává f
- $\Theta(f)$ pokud funkce f a g rostou stejně, tj. pokud $O(f) = \Omega(f)$

Rozhodovací problém (ANO/NE) je rozhodnutelný pokud existuje algoritmus který jej řeší.

Částečně rozhodnutelný problém je takový, pro nějž existuje algoritmus, kde když očekáváme odpověď ANO tak odpoví ANO, ale když očekáváme odpověď NE tak buď odpoví NE nebo se zacyklí.

Riceova věta říká: „Každá netriviální vstupně/výstupní vlastnost programů je nerozhodnutelná."

- Vlastnost je triviální pokud ji mají všechny programy (popř. ji žádný program nemá).
- Vlastnost je vstupně/výstupní pokud to, zda ji program má, plně závisí na tabulce zachycující všechny možné vstupy a jejich výstupy.
  - Vlastnost, že program má více než 10 instrukcí, není vstupně/výstupní. Můžu navrhnout dva programy kde pro každý vstup oba programy dají stejný výstup, ale jeden program bude mít více než 10 instrukcí a druhý program jich bude mít méně.
  - Vlastnost, že program provede zdvojnásobení čísla je vstupně/výstupní. Dva programy které zdvojnásobí vstup musí mít stejnou tabulku vstupů a výstupů.

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

Logaritmická míra říká, že číslo (hodnota buňky RAM stroje) vyžaduje paměť rovnou počtu bitů pro zapsání čísla, a délka trvání instrukce nad číslem je taktéž úměrná velikosti zápisu čísel se kterými instrukce pracuje.

PTIME jsou problémy pro než existuje algoritmus s polynomiální časovou složitostí. Problémy v této třídě považujeme za *prakticky zvládnutelné*. Doplňkové problémy k PTIME patří také do PTIME.

$$PTIME = \bigcup_{k=0}^{\infty} \mathcal{T}(n^k)$$

NPTIME jsou rozhodovací problémy pro než existuje nedeterministický algoritmus s polynomiální časovou složitostí (to, zda jedno konkrétní řešení vrací ANO/NE, je však ověřitelné v polynomiálním čase). Nedeterministický algoritmus rozhoduje ANO/NE problémy tak, že pokud jakýkoliv nedeterministický výpočet vydá *ANO* pak je odpověď na problém *ANO*, a pokud žádný výpočet nevydá *ANO* pak je odpoveď na problém *NE*. Doplňkové problémy k NPTIME nemusí nutně patřit do NPTIME.

NP-těžký problém P je takový, pro který platí že jakýkoliv problém z NP lze na problém P polynomiálně převést (tj. existuje algoritmus s polynomiální časovou složitostí který jeden problém převede na druhý). Do této kategorie patří i nerozhodnutelné problémy (ty samozřejmě nejsou v NP).

NP-úplný problém je takový, který patří do NP a je zároveň NP-těžký. Pokud bychom nalezli deterministický polynomiální algoritmus rozhodující kterýkoliv NP-těžký problém, pak by nutně existoval algoritmus pro každý problém v NP.

Polynomiální převeditelnost říká, že rozhodovací problém P_1 je *polynomiálně převeditelný* na rozhodovací problém P_2, značeno $P_1 \triangleleft P_2$ pokud existuje polynomiální algoritmus A, který pro libovolný vstup ω problému P_1 sestrojí vstup $A(\omega)$ problému P_2 tak, že odpověď $P_1(\omega) = P_2(A(\omega))$.

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

Predikátová logika je rozšířením výrokové logiky o proměnné a kvantifikátory. Slouží k popisu vlastností a vztahů mezi objekty daného univerza.

Abeceda jazyka predikátové logiky zahrnuje symboly pro proměnné označující prvky univerza (x, y, z), logické spojky $(\wedge, \vee, \rightarrow, \leftrightarrow, \neg)$, kvantifikátory $(\forall, \exists)$, relace funkční $(f, g, h$ popisující operace) a predikátové $(P, Q, R$ popisující vztahy). Dále můžeme použít pomocné symboly (závorky, čárku).

Pravidla jazyka predikátové logiky definují termy, zahrnující samostatné proměnné a výsledky operací nad termy $(x, f(x), g(f(x)))$, atomické formule kterými jsou vlastnosti termů $(P(x), P(f(x)))$ a rovnost termů $(x = y)$, a neatomické formule což jsou různé kombinace kvantifikátorů a logických spojek nad formulemi.

Vázaná proměnná je taková, která se vyskytuje vedle kvantifikátoru $(\forall x, \exists y)$. Proměnné které nejsou *vázané* jsou *volné*.

Formule je otevřená právě když obsahuje alespoň 1 volnou proměnnou. V opačném případě je formule *uzavřená*.

Valuace je jedno konkrétní přiřazení prvků univerza proměnným.

Interpretace dává všem funkcím a predikátům význam v rámci daného univerza. Poté můžeme rozhodnout zda je valuace v dané interpretaci pravdivá.

Příklady:

- Pro každé x platí, že pokud x má vlastnost P pak x má i vlastnost Q.

  $\forall x(P(x) \rightarrow Q(x))$

- Všechny čtverce jsou zelené ($C(x) =$ je čtverec, $Z(x) =$ je zelený).

  $\forall x(C(x) \rightarrow Z(x))$

- Existuje y takové, že y má vlastnost R.

  $\exists y R(y)$

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

Relace nad množinami $A_1, A_2, \ldots, A_n$ je libovolná podmnožina kartézského součinu $A_1 \times A_2 \times \cdots \times A_n$.

- Relace je *binární* pokud $n = 2$
- Relace je *ternární* pokud $n = 3$
- Relace je *homogenní* pokud $A_1 = A_2 = \cdots = A_n$, v opačném případě je *heterogenní*

Binární relace R je:

- *Reflexivní* pokud $\forall a \in A : (a, a) \in R$

- *Ireflexivní* pokud $\forall a \in A : (a, a) \notin R$
- *Symetrická* pokud $\forall a, b \in A : (a, b) \in R \implies (b, a) \in R$
- *Asymetrická* pokud $\forall a, b \in A : (a, b) \in R \implies (b, a) \notin R$
- *Antisymetrická* pokud $\forall a, b \in A : (a, b) \in R \wedge (b, a) \in R \implies a = b$
- *Tranzitivní* pokud $\forall a, b, c \in A : (a, b) \in R \wedge (b, c) \in R \implies (a, c) \in R$

Uzávěr je nejmenší relace která rozšiřují R tak, aby výsledná relace R' splnila určitou vlastnost (tedy $R \subseteq R'$).

Ekvivalence

Ekvivalence je *reflexivní, symetrická, tranzitivní.*

Třída ekvivalence prvku $a \in A$ se značí $[a]_R$ (nebo zkráceně $[a]$), a je to množina všech prvků ekvivalentních s prvkem a, neboli $[a]_R = \{b \in A \mid (a, b) \in R\}$. Ekvivalence $R \subseteq A \times A$ definuje na A rozklad $\{[a]_R \mid a \in A\}$ (rozklad množiny A je množina vzájemně disjunktních podmnožin A jejichž sjednocení je rovno A).

Systém zbytkových tříd je množina všech tříd rozkladu dané ekvivalence.

Kongruence modulo m je ekvivalencí na množině celých čísel, kde 2 čísla x, y jsou ekvivalentní pokud $x \bmod m = y \bmod m$. Faktorovou množinu nazýváme *systém zbytkových tříd.*

Uspořádání

Neostré uspořádání je *reflexivní, tranzitivní, antisymetrické.*

Ostré uspořádání je *asymetrické, tranzitivní* (asymetrické ⇒ ireflexivní a antisymetrické).

Uspořádání je úplné pokud v něm neexistují vzájemně nesrovnatelné prvky, tj. $\forall a, b \in A$ platí buď $(a, b) \in R$, nebo $(b, a) \in R$, nebo $a = b$.

Prvek a v relaci uspořádání A je:

- *Minimální* pokud v A neexistuje prvek $x < a$ (více prvků může být minimálních)
- *Maximální* pokud v A neexistuje prvek $x > a$ (více prvků může být maximálních)
- *Nejmenší* pokud je menší než všechny ostatní prvky v A (pouze 1 prvek může být nejmenší)
- *Největší* pokud je větší než všechny ostatní prvky v A (pouze 1 prvek může být největší)
- *Infimum* množiny B ($a = \inf B$) pokud je největší ze všech prvků, které jsou menší než všechny prvky z B
- *Suprémum* množiny B ($a = \sup B$) pokud je nejmenší ze všech prvků, které jsou větší než všechny prvky z B

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

Abstraktní algebra je naukou o algebraických strukturách.

Algebraická struktura definuje množinu nosičů (nosič je sám o sobě množina — např. čísel, funkcí, množin) a operací nad těmito nosiči.

Operace na množině A je zobrazení $A^n \to A$ (n = arita funkce).

Vlastnosti homogenních binárních operací:

- Asociativita
- Komutativita
- Idempotentnost
- Existence neutrálního prvku
- Existence jednotkového prvku
- Existence inverzního prvku
- Distributivita (pro 2 operace)

Grupoid je algebra s jedinou operací, pro kterou platí uzavřenost.

Řád grupy je počet prvků grupy (jejího jediného nosiče).

Příkladem grupy která není Abelova jsou symetrie geometrických útvarů (záleží na pořadí rotace a překlopení).

Okruh/Ring je algebraický systém $(A, +, \cdot)$ kde $(A, +)$ je Abelova grupa, $(A, \cdot)$ je pologrupa, a násobení je vzhledem ke sčítání distributivní.

Těleso/Division Ring je asociativní okruh $(A, +, \cdot)$ kde $(A, +)$ je Abelova grupa, $(A \setminus \{0\}, \cdot)$ je grupa.

Svaz je algebra $(L; \cup; \cap)$ s operacemi spojení $\cup$ a průsek $\cap$, pro které platí uzavřenost, komutativita, asociativita, a absorpce.

FCA – formální kontext, formální koncept, konceptuální svazy. Asociační pravidla, hledání často se opakujících množin položek.

Formální konceptuální analýza pracuje s daty, které popisují vztahy mezi objekty a atributy. Lze je zaznamenat do tabulky, kde v řádcích jsou objekty a ve sloupcích jsou atributy.

Formální kontext je uspořádaná trojice (X, Y, I) kde X je množina objektů, Y je množina atributů, a $I \subseteq X \times Y$ je binární relace která nám říká zda objekt X má atribut Y.

Formální koncept je dvojice množin $A \subseteq X$ a $B \subseteq Y$ tak, aby v B byly atributy společné všem objektům v A, a v A byly všechny objekty které mají společné atributy z B. Koncepty odpovídají maximálním obdélníkům které můžeme nakreslit v tabulce s daty.

Konceptuální svaz je hierarchicky uspořádaná množina všech konceptů formálního kontextu. Nad koncepty definujeme operaci $\leq$ pokud koncept na levé straně je podpojmem konceptu na pravé straně, tj. koncept na levé straně je nejvýše tak obecný jako koncept na pravé straně, tj. pro $(A_1, B_1) \leq (A_2, B_2)$ platí že každý objekt z A_1 patří do A_2 nebo každý atribut z B_2 patří do B_1.

Asociační analýza slouží k hledání souvislostí mezi položkami v množině transakcí.

Asociační pravidlo pro množinu položek I je ve tvaru $A \Rightarrow B$, kde $A, B \subseteq I$, $A, B \neq \emptyset$, a $A \cap B = \emptyset$. Říká, že pokud si zákazník kupuje množinu položek A, pak bude pravděpodobně kupovat i množinu položek B.

Transakcí myslíme podmnožinu množiny položek (položky které si někdo koupil najednou).

Množina transakčních dat je seznam všech zkoumaných transakcí (přesněji, je to funkce $T(k)$ která nám vrátí k-tou transakci). Sledujeme:

- Podporu dané množiny položek K ($supp(K)$), což je poměr transakcí obsahujících všechny položky z K k počtu všech transakcí (říká nám kolik procent transakcí obsahuje všechny položky K).
- Spolehlivost pravidla $A \Rightarrow B$ ($conf(A \Rightarrow B)$), což je podpora $supp(A \cup B)$ dělena podporou $supp(A)$ (říká nám kolik procent transakcí, které obsahují všechny položky z A, obsahují zároveň i všechny položky z B).

Rymon tree je graf (strom) zobrazující všechny kombinace položek, počínaje prázdnou množinou. Lze použít pro optimalizaci hledání položek s danou minimální podporou (po přidání položky do množiny podpora nemůže narůst, takže můžeme zanedbat celou větev stromu).

Algoritmus apriori je jeden ze způsobů hledání *frequent itemsetů*, tedy množin položek které dosahují dané minimální podpory. Algoritmus generuje strom kandidátů podobný Rymon tree do šířky, a na konci každé iterace zkontroluje kteří kandidáti dosáhli požadované minimální podpory a ty v další iteraci rozšiřuje.

Metrické a topologické prostory – metriky a podobnosti.

Metrický prostor je struktura pomocí které definujeme koncept vzdálenosti mezi prvky množiny.

Metrika je zobrazení, které dvojici prvků dané množiny přiřadí *vzdálenost* (nezáporné reálné číslo). Metrika splňuje 4 axiomy:

- Nezápornost ($\forall x \forall y(\rho(x, y) \geq 0)$)
- Totožnost ($\forall x \forall y(\rho(x, y) = 0 \leftrightarrow x = y)$); pseudometrika tento axiom vynechává
- Symetrie ($\forall x \forall y(\rho(x, y) = \rho(y, x))$)
- Trojúhelníková nerovnost ($\forall x \forall y \forall z(\rho(x, y) + \rho(y, z) \geq \rho(x, z))$)

Podobnost je zobrazení které dvěma prvkům přiřadí nezáporné reálné číslo, které nějakým způsobem vyjadřuje podobnost těchto prvků. Podobnost musí splňovat axiomy nezápornosti, totožnosti, a symetrie. Pokud nám podobnost dává hodnotu nejvýše 1, můžeme ji převést na vzdálenost (např. $d(x, y) = 1 - s(x, y)$).

Příklady metrických prostorů:

- Množina reálných čísel s metrikou $|x - y|$
- Množina bodů v 2D euklidovském prostoru s manhattanskou metrikou
- Množina bodů v 2D euklidovském prostoru s euklidovskou metrikou
- Množina textových řetězců s Levenštejnovou vzdáleností
- Množina vrcholů neorientovaného a souvislého grafu s metrikou nejkratší cesty mezi dvěma vrcholy
- Množina množin s Jaccardovou vzdáleností (převedení Jaccardovy podobnosti tak, že vzdálenost $d(x, y) = 1 - J(x, y)$)

Příklady podobností:

- Kosinova podobnost (kosinus úhlu mezi dvěma vektory)
- Jaccardova podobnost množin (velikost průniku dvou množin dělen velikostí jejich sjednocení)

Topologie je matematický obor, studující vlastnosti útvarů, které se nemění při spojitých transformacích (tj. dva útvary/prostory jsou topologicky ekvivalentní pokud můžeme jeden přeměnit na druhý bez toho, aby se někde roztrhl nebo spojil). Příkladem zkoumaných útvarů jsou např. geometrické tvary, nebo struktury sítí.

Spojité zobrazení je takové zobrazení, kde platí, že pro libovolný bod x prostoru X a jeho obraz $f(x)$ se okolí bodu x zobrazí do okolí bodu $f(x)$. U metrik to znamená, že pokud si jsou dva body blízké, pak si jsou blízké i jejich odrazy; u topologie to znamená, že množina a obraz množiny si zachovají topologické vlastnosti (tj. vzor otevřené množiny v prostoru zůstane otevřenou množinou v zobrazeném prostoru).

Otevřená množina je taková množina, kde pro každý bod který do této množiny patří je v množině i nějaké jeho okolí. Jedná se o zobecnění otevřeného intervalu reálných čísel.

Topologický prostor je zobecněním metrického prostoru. Topologickým prostorem nazveme dvojici (X, τ) kde X je množina, a τ (nazýváme topologií) je kolekce podmnožin množiny X splňující následující axiomy (podle definice pomocí otevřených množin, což je jedna z možných definic):

- $\emptyset \in \tau$
- $X \in \tau$
- $\forall A, B \in \tau : A \cap B \in \tau$
- $\forall A \in \tau, \forall i \in I : \cup_{i \in I} U_i \in \tau$ (I je indexová množina která může být nekonečná)

Shlukování.

Shlukování přiřazuje prvky množiny do shluků tak, aby prvky ve shluku byly mezi sebou *podobné*, a naopak odlišné od prvků v jiných shlucích.

Hierarchické shlukování dělíme na divisivní (začínají s jedním shlukem zahrnující všechny prvky a postupně jej rozdělují na menší) a aglomerativní (začínají s jedním shlukem pro každý prvek a postupně je spojují dohromady). Nevyžadují znát cílový počet shluků, ale musíme proces nějakým způsobem zastavit. Výsledkem hierarchického shlukování je *dendrogram*.

Nehierarchické shlukování zahrnuje všechny ostatní metody shlukování (např. metody založeny na vzdálenosti od centra, nebo metody založeny na hustotě).

Aglomerativní hierarchické shlukování postupně spojuje dva nejbližší shluky podle jedné z následujících metod které počítají vzdálenost dvou shluků:

- Single linkage (vzdálenost shluků je vzdáleností dvou nejbližších prvků v těchto shlucích)
- Complete linkage (vzdálenost shluků je vzdáleností dvou nejvzdálenějších prvků v těchto shlucích)
- Average linkage (vzdálenost shluků je průměrem vzdáleností všech párů prvků z různých shluků)

Divisivní hierarchické shlukování má exponenciální náročnost (oproti aglomerativnímu které lze provést polynomiálně), proto se buď používají metody s heuristikami, nebo se použije aglomerativní nebo nehierarchické shlukování.

K-means začne náhodným výběrem prvků které se stanou centroidy (jejich počet je definován uživatelem). Poté proběhne zpřesňování tak, že se přiřadí každý prvek k nejbližšímu centroidu, a poté se pozice všech centroidů přepočítá tak aby byly v těžišti shluku (zprůměruje se pozice všech prvků); proces se opakuje dokud se shluky mění. Jednou z nevýhod je, že funguje pouze u dat ve kterých hledáme shluky kulovitého tvaru.

DBSCAN je algoritmus založený na hustotě (tvoří shluky v oblastech které jsou hustě pokryty prvky datového souboru), funguje tedy dobře u dat kde hledáme shluky libovolného tvaru. Oproti k-means umí také klasifikovat odlehlé body (noise) jaké oddělené od shluků.

Silueta je jeden ze způsobů kontroly zda jsou shluky dobře utvořeny. Silueta konkrétního prvku udává hodnotu, která vyjadřuje blízkost prvku k ostatním prvkům ve stejném shluku, a vzdálenost prvku od nejbližšího (jiného) shluku. Koeficient siluety tuto ideu vyjadřuje pro všechny prvky datového souboru.

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

Náhodný pokus je děj, jehož výsledek není určen počátečními podmínkami (není tedy deterministický).

Základní prostor (ω) je množinou všech možných výsledků náhodného pokusu.

Náhodný jev je podmnožinou ω. Pokud je podmnožina jednoprvková, mluvíme o elementárním jevu, v opačném případě mluvíme o složeném jevu.

Náhodná veličina je funkce, která každému elementárnímu jevu přiřadí číselnou hodnotu.

Distribuční funkce je jeden ze způsobů popisu rozdělení náhodné veličiny (plně náhodnou veličinu definuje). Každému reálnému číslu přiřazuje pravděpodobnost, že náhodná veličina bude mít hodnotu menší než dané reálné číslo ($F(x) = P(X < x)$). Pro *spojitou veličinu* bude grafem distribuční funkce spojitá, pro *diskrétní veličinu* bude *schodovitá*. Platí, že funkce:

- Nabývá hodnoty v intervalu $[0; 1]$
- Je neklesající
- Je zleva spojitá
- Začíná v 0 ($\lim_{x \to -\infty} F(x) = 0$)
- Končí v 1 ($\lim_{x \to \infty} F(x) = 1$)

Diskrétní náhodná veličina X nabývá konečného množství hodnot $x \in X$. Pravděpodobnostní funkce $P(X = x_i) = P(x_i)$ udává pravděpodobnost, že náhodná veličina nabývá hodnoty x_i (pravděpodobnost nesmí být záporná, a platí $\sum_{i \in I} P(x_i) = 1$). Distribuční funkci vyjádříme jako $F(x) = \sum_{x_i < x} P(x_i)$.

Spojitá náhodná veličina může nabýt všech hodnot (reálných čísel) v daném intervalu. Pravděpodobnostní funkce je nulová (nemá smysl ptát se na pravděpodobnost, že se bude rovnat konkrétnímu reálnému číslu, když jich může být nekonečno); namísto toho se používá *hustota pravděpodobnosti* $f(x)$, což je nezáporná funkce jejíž integrál (plocha pod křivkou) je roven 1 (hustota nesmí být v žádném bodě záporná, ale může být větší než 1). Distribuční funkci vyjádříme jako $F(x) = \int_{-\infty}^{x} f(t)dt$.

Číselná charakteristika popisuje nějakou vlastnost náhodné veličiny. Patří k nim střední hodnota, rozptyl, kvantily, atd.

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

Diskrétní náhodné veličiny

Binomické rozdělení popisuje náhodnou veličinu, která udává počet úspěchu v daném počtu Bernoulliho pokusů. Bernoulliho pokusy jsou posloupností nezávislých pokusů, kde každý pokus má stejnou šanci na úspěch. Rozdělení $Bi(n, p)$ je definováno počtem pokusů n a pravděpodobností na úspěch p. Příklady:

- Kolikrát padne na kostce číslo 2, pokud házíme 5x?
- Kolik je chlapců v rodině s 8 dětmi?
- Kolik je vadných výrobků v prodejně s 30 výrobky? (Musíme dávat pozor, aby byl výběr nezávislý, počet vad může záviset na konkrétní šarži.)

Alternativní rozdělení je speciální případ binomického rozdělení, kde počet pokusů je 1 ($A(p) = Bi(1, p)$).

Hypergeometrické rozdělení popisuje náhodnou veličinu, která udává počet úspěchů v daném počtu závislých pokusů. Rozdělení $H(N, M, n)$ je definováno celkovou velikostí základního souboru N, počtem prvků M které mají danou vlastnost (u kterých je pokus *úspěšný*), a velikostí výběru n. Příklady:

- Kolik je vadných z 10 výrobků, které jsme vybrali z dodávky 30 výrobků mezi kterými bylo 12 výrobků vadných?
- Kolik je dívek ve skupině 5 děti, které jsou ze třídy s 6 chlapci a 9 dívkami?

Negativně binomické rozdělení popisuje náhodnou veličinu, která udává počet Bernoulliho pokusů do k-tého výskytu úspěchu (někdy se počítá i samotný k-tý výskyt). Rozdělení $NB(k, p)$ je definováno požadovaným počtem úspěchů k a pravděpodobností úspěchu p.

Geometrické rozdělení je speciální případ negativně binomického rozdělení, kde požadujeme pouze první úspěch ($G(p) = NB(1, p)$).

Poissonův proces popisuje počet náhodných událostí v uzavřené oblasti (časový interval, plocha, objem). Důležitou vlastností je beznáslednost — pravděpodobnost výskytu události není závislá na čase který uplynul od předchozí události. Parametrem Poissonova procesu je λ značící rychlost výskytu událostí (popř. hustotu výskytu na dané ploše nebo v daném objemu). Kromě beznáslednosti vyžadujeme také aby λ bylo konstantní, a aby nenastával příliš velký počet událostí ve velmi krátkém čase (chceme tzv. *řídké jevy*).

Poissonovo rozdělení popisuje náhodnou veličinu, která udává počet výskytů události v časovém intervalu délky t, nebo počet výskytů události na ploše t (v objemu t). Rozdělení značíme $Po(\lambda t)$. Lze jej aproximovat pomocí binomického rozdělení, kdy rozdělíme interval t na n částí kde pravděpodobnost výskytu události v subintervalu je $\frac{\lambda t}{n}$, a vezmeme limitu $\lim_{n\to\infty}$ pravděpodobnostní funkce rozdělení $Bi(n, \frac{\lambda t}{n})$. Odvození:

$$
\begin{aligned}
P(X=k) &= \lim_{n\to\infty} \binom{n}{k} \cdot \left(\frac{\lambda t}{n}\right)^k \cdot \left(1-\frac{\lambda t}{n}\right)^{(n-k)} \\
&= \lim_{n\to\infty} \frac{n!}{(n-k)!k!} \cdot \frac{(\lambda t)^k}{n^k} \cdot \left(1-\frac{\lambda t}{n}\right)^{(n-k)} \\
&= \frac{(\lambda t)^k}{k!} \cdot \lim_{n\to\infty} \frac{n!}{(n-k)!} \cdot \frac{1}{n^k} \cdot \left(1-\frac{\lambda t}{n}\right)^{(n-k)} \\
&= \frac{(\lambda t)^k}{k!} \cdot \lim_{n\to\infty} \frac{n!}{(n-k)!n^k} \cdot \lim_{n\to\infty} \left(1-\frac{\lambda t}{n}\right)^{n} \cdot \lim_{n\to\infty} \left(1-\frac{\lambda t}{n}\right)^{-k} \\
&= \frac{(\lambda t)^k}{k!} \cdot 1 \cdot \frac{1}{e^{\lambda t}} \cdot 1 \\
&= \frac{(\lambda t)^k e^{-\lambda t}}{k!}
\end{aligned}
$$

Spojité náhodné veličiny

Rovnoměrné rozdělení je rozdělení jehož hustota pravděpodobnosti je konstantní na daném spojitém intervalu $(a;b)$. Zapisujeme $R(a,b)$. Hustota pravděpodobnosti $f(x) = \frac{1}{b-a}$ pro $x \in (a;b)$, a 0 pro $x \notin (a;b)$.

Exponenciální rozdělení popisuje náhodnou veličinu, která udává dobu do výskytu první události (popř. dobu mezi dvěma událostmi) Poissonova procesu s parametrem λ. Zapisujeme $Exp(\lambda)$. Hustota pravděpodobnosti $f(x) = \lambda \cdot e^{-\lambda t}$ pro $x > 0$, a 0 pro $x \leq 0$. Kvůli požadavku beznáslednosti je vhodné pouze pro události které nastávají náhodně bez vlivu na stáří (opotřebení), tedy jsou v období stabilního života.

Weibullovo rozdělení je zobecněním exponenciálního rozdělení, podle kterého lze modelovat nejen oblast stabilního života, ale i období časných poruch (předchází stabilnímu životu) a období stárnutí (následuje po stabilním životu). Rozdělení $W(\Theta, \beta)$ je definováno parametrem měřítka Θ a parametrem tvaru β. Exponenciální rozdělení je tedy speciální případ Weibullova rozdělení s parametry $\Theta = \frac{1}{\lambda}$ a $\beta = 1$.

Erlangovo rozdělení popisuje náhodnou veličinu, která udává dobu do výskytu k-té události Poissonova procesu ($Erlang(k, \lambda)$). Používá se pro modelování období stárnutí.

Náhodné rozdělení popisuje náhodnou veličinu, jejíž hodnoty se symetricky shlukují okolo střední hodnoty a tvarem tvoří Gaussovu křivku. Rozdělení $N(\mu, \sigma^2)$ je definováno střední hodnotou μ, a rozptylem σ^2. Protože v distribuční funkci je integrál, používá se normované náhodné rozdělení $N(0,1)$ jehož hodnoty distribuční funkce a hustoty pravděpodobnosti najdeme v tabulkách a existuje převod na normální rozdělení s jakýmikoliv parametry. Pro náhodnou veličinu $X \sim N(\mu, \sigma^2)$ můžeme zavést náhodnou veličinu $Z = \frac{X-\mu}{\sigma}$. Veličina Z má normované normální rozdělení, a platí vztah $F(X) = \phi(\frac{x-\mu}{\sigma})$, kde hodnoty ϕ najdeme v tabulce.

Pravidlo 3σ říká, že máme li data pocházející z normálního rozdělení, pak 99.8% z nich leží v intervalu $(\mu \pm 3\sigma)$.

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

Popisná statistika (také explorační statistika) slouží k číselnému popisu a vizualizaci hromadných jevů (jevů které zkoumáme u větší části populace a/nebo jevy které zkoumáme opakovaně).

Vlastnosti zkoumané u kategoriálních proměnných:

- Nominální i ordinální
  - Četnost (počet výskytů dané hodnoty)
  - Relativní četnost (četnost dělena velikostí souboru)
  - Modus (nejčetnější hodnota)
- Ordinální
  - Kumulativní četnost (počet výskytů dané hodnoty, a hodnot které jsou pořadí před ní)
  - Kumulativní relativní četnost (kumulativní četnost dělena velikostí souboru)

Vizualizace kategoriálních proměnných:

- Nominální i ordinální
  - Sloupcový graf
  - Koláčový graf
- Ordinální
  - Lorenzova křivka (pozor, ve skriptech obrázek spojnicového grafu není Lorenzova křivka)
  - Paretův diagram (kombinace sloupcového a čárového grafu, kde data jsou seřazeny podle četnosti od největší po nejmenší, sloupce jsou četnosti, a čáry jsou kumulativní četnosti)

Vlastnosti zkoumané u numerických proměnných:

- Aritmetický průměr
- Vážený průměr
- Geometrický průměr (používá se např. u veličin s procentuálním růstem; vynásobíme všechny hodnoty a provedeme n-tou odmocninu kde n je velikost souboru)
- Harmonický průměr (používá se např. u zjištění průměrného času u úkolů které jsou prováděny současně; převrátíme všechny hodnoty $(1/x)$, spočítáme převrácenou hodnotu aritmetického průměru, a to vše opět převrátíme)
- Modus
  - Pro diskrétní proměnné se jedná o nejčetnější hodnotu
  - Pro spojité proměnné může mít proměnná více modů, jedná se o hodnotu(y) kolem níž je největší koncentrace hodnot.
- Kvantil (rozděluje uspořádaný soubor hodnot na dvě části; např. medián rozděluje v polovině)
- Interkvartilové rozpětí (vzdálenost mezi horním kvartilem — 75% — a dolním kvartilem — 25%)
- Medián absolutních odchylek od mediánu ($med(|x_i - med(X)|)$)
- Rozptyl ($\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}$)
- Směrodatná odchylka (odmocnina rozptylu)
- Šikmost (říká jak jsou rozloženy hodnoty okolo průměru — zda jsou rozloženy symetricky, nebo převažují hodnoty větší/menší než průměr)
- Špičatost (říká jak špičatá je křivka rozdělení v porovnání s normálním rozdělením)

Vizualizace numerických proměnných:

- Empirická distribuční funkce
- Krabicový graf
- Číslicový histogram

Identifikace odlehlých pozorování u numerických proměnných:

- Metoda vnitřních hradeb
- Metoda z-skóre
- Metoda $x_{0,5}$

Metody statistické indukce. Intervalové odhady. Princip testování hypotéz.

Populace (základní soubor) je statistický soubor který zkoumáme při statistickém výzkumu.

Výběr (výběrový soubor) je podmnožina populace vybrané na základně nějakých kritérií.

Statistická indukce slouží ke stanovení vlastností celé populace na základě jeho části (výběru). Řadíme k ní teorii odhadů a testování hypotéz.

Reprezentativní výběr je takový, který dobře odráží charakteristiky celé populace. *Výběrová chyba* nastane, když je výběr nereprezentativní, buď v důsledku špatného procesu výběru nebo pokud existuje systematický důvod proč např. někteří lidé neodpoví na dotazník.

Výběrová charakteristika je vhodná funkce popisující náhodný výběr nějakého parametru populace, jejíž analýzou získáme informace o daném parametru vzhledem k celé populaci. Variabilita výběrové charakteristiky záleží na velikosti populace a vzorku, a samotném způsobu náhodného výběru reprezentantů do vzorku.

Výběrový průměr je průměr hodnot ve výběru. Pochází-li výběr z normálního rozdělení, pak má výběrový průměr také normální rozdělení se stejným průměrem (střední hodnotou), a rozptylem který je $\frac{1}{n}$ původního rozptylu (n = velikost výběru).

Zákon velkých čísel říká, že s rostoucím rozsahem výběru se výběrový průměr stále silněji soustřeďuje okolo opravdového průměru.

Centrální limitní věta dále říká, že dostatečně velký náhodný výběr (okolo 30 pozorování) má přibližně normální rozdělení, nezávisle na rozdělení proměnné ze které výběr provádíme ($\sum_{i=1}^{n} x_i \sim N(n\mu_x, n\sigma_x^2)$).

Teorie odhadů

Bodový odhad aproximuje parametr populace jedním číslem. Takový odhad je sám o sobě náhodnou veličinou. O odhadu říkáme že je *nestranný* pokud se jeho střední hodnota rovná zkoumanému parametru, a *konzistentní* pokud se s rostoucím rozsahem výběru zpřesňuje (blíží se zkoumanému parametru a rozptyl se blíží k nule).

Nejlepší bodový odhad střední hodnoty u normálního rozdělení je aritmetický průměr hodnot ve výběru. Stejně tak to platí u rozptylu (výběrový rozptyl), směrodatné odchylky (výběrová směrodatná odchylka), relativní četnosti (výběrová relativní četnost), a určitě i jinde.

Interval spolehlivosti je typ intervalového odhadu, který aproximuje parametr populace intervalem, ve kterém hodnoty parametru leží s pravděpodobností (spolehlivostí odhadu) $1 - \alpha$. Čím větší

spolehlivost, tím větší je interval a tím menší je jeho vypovídací schopnost, hledáme tedy kompromis mezi spolehlivostí a významností. Hladinou významnosti nazveme α.

Interval spolehlivosti střední hodnoty u normálního rozdělení počítáme pomocí kvantilů normovaného normálního rozdělení ($x_{(1-\alpha)}$ pro jednostranné odhady a $x_{(1-\frac{\alpha}{2})}$ pro oboustranné odhady). Kvantil(y) vynásobíme $\frac{\sigma}{\sqrt{n}}$ a přičteme/odečteme od výběrového průměru. Pokud neznáme směrodatnou odchylku σ, můžeme při dostatečně velkém výběru použít výběrovou směrodatnou odchylku s.

Intervalový odhad střední hodnoty u nenormálního rozdělení provádíme pomocí *robustních metod*, mezi které patří např. odhad mediánu, Gastwirthův medián, nebo Bootstrap.

Intervalový odhad směrodatné odchylky snad nebude zkoušející zajímat.

Testování hypotéz

Statistická hypotéza je výrok o rozdělení pozorované náhodné veličiny. Rozlišujeme *parametrické hypotézy* týkající se parametrů rozdělení (např. střední hodnota, medián) a *neparametrické hypotézy* týkající se jiných vlastností (např. typ rozdělení, nezávislost výběru).

Testování hypotéz proti sobě pokládá dvě tvrzení — *nulovou hypotézu* a *alternativní hypotézu*.

Nulová hypotéza říká, že testovaný parametr je roven očekávané hodnotě, tedy že sledovaný efekt je nulový.

Alternativní hypotéza popírá nulovou hypotézu tvrzením, že testovaný parametr je buď menší, větší, nebo neroven očekávané hodnotě, nebo že je roven hodnotě jiné (kterou možnost zvolíme závisí na zadání problému). Hypotézu, že testovaný parametr je neroven očekávané hodnotě, nazýváme *oboustrannou*.

Testem statistické hypotézy buď *zamítáme nulovou hypotézu* ve prospěch alternativní hypotézy, nebo *nulovou hypotézu nezamítáme*.

U klasického testu formulujeme nulovou a alternativní hypotézu, zvolíme výběrovou charakteristiku, ověříme předpoklady testu, a určíme *kritický obor*. Kritický obor je množina hodnot, pro kterou je nepravděpodobné že by hodnota výběrové charakteristiky do této množiny padla, za předpokladu že nulová hypotéza platí. Pokud hodnota výběrové charakteristiky padne do kritického oboru, pak nulovou hypotézu zamítáme.

U čistého testu postupujeme podobně, ale namísto kritického oboru spočítáme p-hodnotu, což je nejnižší hladina významnosti na níž můžeme nulovou hypotézu zamítnout. Říkáme pak, že nulovou hypotézu můžeme zamítnout na hladině významnosti p (jinak řečeno, zamítnout se spolehlivostí $1-p$). Rozhodnutí s hladinou významnosti α provedeme tak, že nulovou hypotézu zamítneme pokud $p < \alpha$.

Chyba I. druhu nastane, pokud je nulová hypotéza pravdivá ale my ji zamítneme (false positive, zamítnutí se počítá jako *pozitivní* výsledek).

Chyba II. druhu nastane, pokud je nulová hypotéza nepravdivá ale my ji nezamítneme (false negative).

Testy dobré shody slouží pro ověření shody pozorovaného a výběrového rozdělení. Řadíme k nim například:

- χ^2 (chí kvadrát) test dobré shody
- Kolmogorovův-Smirnovův jednovýběrový test (má větší sílu testu než χ^2 a funguje i s menším rozsahem výběru, ale rozdělení musí být určeno jednoznačně i s parametry což u χ^2 vyžadováno není)

Jednovýběrový test je test o parametru jedné populace. Řadíme k nim například:

- Test o rozptylu — nulovou hypotézou je rovnost rozptylu zadané hodnotě, předpokladem je normalita populace.
- Jednovýběrový z test a t test — nulovou hypotézou je rovnost střední hodnoty zadané hodnotě, předpokladem je normalita populace. Pokud známe rozptyl použijeme z test, pokud neznáme rozptyl použijeme t test.
- Jednovýběrový Wilcoxonův test — nulovou hypotézou je rovnost mediánu zadané hodnotě, předpokladem je symetrie rozdělení (používá se tedy namísto z/t testu u nenormálního rozdělení).

Dvouvýběrový test je test o parametru dvou populací. Řadíme k nim například:

- Test o shodě rozptylů — nulovou hypotézou je rovnost dvou rozptylů, předpokladem je nezávislost výběrů a normalita populací.
- Dvouvýběrový z test — nulovou hypotézou je rovnost střední hodnoty zadané hodnotě, předpokladem je nezávislost výběrů, normalita populací, a známé rozptyly
- Dvouvýběrový t test — nulovou hypotézou je rovnost střední hodnoty zadané hodnotě, předpokladem je nezávislost výběrů, normalita populací, a rozptyly jejichž hodnoty neznáme ale víme že jsou shodné
- Aspinové-Welchův test — nulovou hypotézou je rovnost střední hodnoty zadané hodnotě, předpokladem je nezávislost výběrů, normalita populací, a rozptyly jejichž hodnoty neznáme a nevíme zda jsou shodné

Vícevýběrový test je test o parametru tří nebo více populací. Řadíme k nim například:

- Bartlettův, Leveneův, Hartleyův, a Cochranův test o shodě rozptylu
- ANOVA

ANOVA porovnává střední hodnoty více než dvou populací.

- Nulovou hypotézou je rovnost středních hodnot všech populací, alternativní hypotézou je negace nulové hypotézy.
- Předpokladem je nezávislost výběrů, normalita rozdělení, a homoskedasticita (shodnost rozptylů) — při narušení závislosti použijeme Friedmanův test, při narušení normality nebo homoskedasticity použijeme Kruskalův-Wallisův test.
- Prvním krokem je *explorační analýza* (tu provedeme např. vizualizací dat pomocí krabicového grafu pro každý výběr, nebo spočtením číselných charakteristik). Odstraníme odlehlá pozorování.
- Spočítáme p-hodnotu a rozhodneme o zamítnutí nulové hypotézy.
- Pokud zamítneme nulovou hypotézu, následuje *post hoc analýza*, během které zjistíme mezi kterými dvojicemi populací existují statisticky významné rozdíly. K metodám post hoc analýzy patří Fisherovo LSD, Bonferroniho metoda, Scheffého metoda, a další.

Softwarové inženýrství

Softwarový proces. Jeho definice, modely a úrovně vyspělosti.

Softwarové inženýrství je disciplína zabývající se metodami návrhu, tvorby, a údržby softwaru.

Softwarový proces definuje kroky vedoucí k vyhotovení softwaru, a jeho následné údržbě. Typicky rozlišujeme mezi sekvenčními procesy, které kladou důraz na naplánování celého vývoje jako sekvence kroků, a iterativními nebo agilními procesy které rychle reagují na změny požadavků v průběhu vývoje. Mezi těmito dvěma extrémy najdeme i hybridní metodiky které oba přístupy kombinují.

Vodopádový model je typickým příkladem sekvenčního procesu. Existuje několik variací základního modelu, představím jednu která rozděluje vývoj do 6 fází:

1. Analýza a specifikace požadavků
2. Návrh architektury softwaru
3. Implementace
4. Testování a oprava chyb
5. Instalace
6. Údržba

Typicky je nutno aktuální fázi dokončit než lze začít fázi následující. Nevýhodou je, že vodopádový model je nepraktický — požadavky na software se často mění nebo nejsou na začátku úplně jasné, chybí komunikace mezi lidmi kteří mají na starost různé fáze, takže se může např. ukázat že daný návrh nelze dobře implementovat, může trvat velmi dlouho než klient uvidí výsledný software, atd.

Proto vznikly modifikace, např. model Sašimi kde se jednotlivé fáze překrývají, nebo inkrementální model.

Inkrementální model rozděluje požadavky do menších *inkrementů*, ze kterých vznikne sekvence menších vodopádů. Na konci každého vodopádu/inkrementu je daný požadavek plně implementován a otestován (poté už se nepočítá s tím, že by se požadavek měnil) a výsledkem je funkční softwarový produkt. Takto se postupně staví celý produkt až do konce posledního vodopádu kdy jej považujeme za dokončený.

Agilní metodiky zahrnují řadu *iterativních* modelů které sdílí tyto vlastnosti:

- Rozdělení práce do krátkých iterací, na jejichž konci máme funkční produkt.
- Návrh, implementace, a testování daného požadavku probíhá v iteracích, jejichž cílem je využití zpětné vazby k dalšímu zlepšování. Tím pádem se i rychle adaptuje na změny v požadavcích.
- Použití technologií a vzorů které tento styl podporují, např. nástroje pro průběžnou integraci (CI), vývoj řízený testy (TDD), párové programování, atd.

Agilní metody nejsou vhodné pro každý typ projektu, a někdy snaha implementovat čistě agilní metody může dopadnout katastroficky:

- Agilní metody nemusí být vhodné pro rozsáhlé projekty, protože není možné plánovat daleko do budoucna a tím pádem ani naplánovat rozpočet celého projektu.
- Některé typy projektů (např. v medicíně) vyžadují zdlouhavé testování a certifikaci, což rozporuje častému vydávání aktualizací softwaru.
- Hrozí, že se nahromadí *technický dluh* pokud se tým zaměřuje příliš hodně na přidávání nové funkcionality namísto práce na správném návrhu a refaktorování kódu.
- Na metody které vyžadují denní meetingy si často vývojáři stěžují jako na ztrátu času (což může být spíš chyba těch, kteří ty meetingy vedou, ale špatná implementace může být horší než žádná implementace).

Extrémní programování je agilní metodikou která vychází z obecně používaných postupů, ale vede je do extrému — např. neustálé párové programování, 100% pokrytí kódu automatickými testy, denní komunikace se zákazníkem, a velmi krátké iterace (tj. velmi časté vydání nových verzí).

Scrum je jednou z dalších agilních metodik která cílí na týmy s malým počtem lidí. Rozděluje vývoj do *sprintů*, které jsou časově omezené (v rámci týdnů, ne delší než 1 měsíc). Každý sprint začíná definováním cíle sprintu, naplánováním jednotlivých úkolů a odhadnutím kolik práce úkoly budou vyžadovat. Na konci sprintu bude existovat funkční produkt který lze dodat zákazníkovi. Další částí metodiky jsou denní schůze týmu na kterých se diskutuje práce provedena předchozí den, a plán práce pro dnešek. To např. způsobuje problém u týmů, které jsou rozprostřeny po světě v různých časových pásmech.

Unified Process je základem řady modelů, které kombinují iterativní a inkrementální filozofii. Je navržen tak, aby byl vhodný i pro rozsáhlé projekty a velké týmy, a aby si model mohl každý upravit pro jeho potřeby — existují varianty jako jsou Rational Unified Process, Agile Unified Process, Open Unified Process atd. Rozděluje vývoj do 4 fází — počátek, příprava, tvorba, a předání — a v rámci každé z nich definuje procesy které často probíhají souběžně a v iteracích (např. správa požadavků je nejdůležitější v počáteční fázi a fázi přípravy, ale do určité míry pokračuje i ve fázi tvorby aby mohl projekt reagovat na změny požadavků; proces testování začíná ve fázi přípravy, a průběžně se opakuje až do fáze předání). Je kladen důraz na vizualizaci a dokumentaci procesů pomocí jazyka UML (ať už jde o návrh samotné implementace, nebo konfigurace propojení jednotlivých komponent celého systému).

Objektivní měření vyspělosti softwarového procesu ve firmě je prováděno například standardem *Capability Maturity Model Integration* (nahrazuje dřívější CMM), který definuje *úroveň zralosti* a *úroveň schopnosti* pro procesy souvisejícími s řízením organizace, řízením projektů, samotnou realizací softwaru, a podpůrnými procesy (např. quality assurance).

Vymezení fáze „sběr a analýza požadavků“. Diagramy UML využité v dané fázi.

Sběr a analýza požadavků je první fází vývoje softwaru. Cílem je zjistit jaké má zákazník požadavky na software — jaké má mít funkce, kdo ho bude používat a na jakých platformách, jak dlouho bude vývoj trvat, atd.

Existuje řada technik které s touto fází pomáhají — v raném začátku analýzy požadavků se používají *případy užití* které definují to, jak bude probíhat interakce uživatelů se softwarem, v pozdějších fázích můžeme např. vytvářet prototypy aby bylo možno získat zpětnou vazbu co nejdříve. Alternativou

k případům užití jsou např. *user story*, které popisují funkcionalitu systému z pohledu uživatele; ty občas najdeme u analýzy požadavků agilních metodik, ale případy užití jsou celkem univerzální.

Případ užití (use case) definuje aktéra, který má v systému nějakou roli (uživatel, správce, nebo i externí systém), a seznam interakcí které popisují jak aktér bude typicky software využívat. Pomocí *diagramu případu užití* zaznamenáme aktéry, systémy, a akce které mohou aktéři v systémech provádět. Ke každému případu užití je ještě dobré písemně zapsat jaké jsou podmínky akce, a jednoduše popsat *scénář* toho jak bude akce probíhat. Cílem je, aby diagram pochopil zákazník který nemá technické znalosti.

DIAGRAMY VIZ POSLEDNÍ OTÁZKA.

Vymezení fáze „Návrh“. Diagramy UML využité v dané fázi. Návrhové vzory – členění, popis a příklady.

Návrh je druhou fází vývoje softwaru po sběru a analýze požadavků. Cílem návrhu je vzít požadavky, a podle nich definovat komponenty celého systému, jak budou propojeny a jaké budou mít rozhraní, dále navrhnout třídy které se poté budou implementovat, a popsat konkrétní procesy.

V této fázi se používají následující UML diagramy:

- Diagram tříd pro popis tříd objektově orientovaného programování
- Sekvenční diagram nebo diagram komunikace pro popis komunikace mezi objekty v rámci jedné události nebo akce
- Diagram nasazení pro popis komponent a architektury celého systému

DIAGRAMY VIZ POSLEDNÍ OTÁZKA.

Návrhové vzory jsou obecnými řešeními klasických návrhových problémů. Dělíme je na vzory:

- Vytvářející, které souvisí s tvorbou objektů
- Strukturální, které souvisí se strukturováním objektů, což zahrnuje kompozici, rozhraní, apod.
- Behaviorální, které se zaměřují na určité způsoby komunikace mezi objekty
- A několik dalších typů, ale ty tady nejsou důležité...

Vytvářející návrhové vzory

Továrna poskytuje rozhraní pro vytváření instancí tříd, které mají několik implementací (např. GUI prvky pro různé operační systémy — na jednom místě se vytvoří konkrétní továrna která vytváří GUI prvky pro daný systém, a zbytek kódu se už nemusí starat o to jakou konkrétní instanci dostane).

Builder je abstrakcí postupné konstrukce objektu. Klasický GOF Builder definuje oddělený builder pro každý typ objektu (např. pokud bychom chtěli sestavit dokument v různých formátech, měli bychom jeden builder pro dokumenty typu Microsoft Word, jeden builder pro Markdown formát, a každý builder by měl stejné rozhraní — metodu pro přidání textu, metodu pro přepínání tučnosti a kurzívy, atd.) ale často najdeme i podobné vzory kde existuje jeden builder kterému postupně předáváme parametry, a na konci provede validaci parametrů a sestrojí finální objekt (např. u konstrukce seznamu který je pouze pro čtení můžeme použít builder do kterého postupně přidáváme prvky seznamu, a na konci — protože finální seznam bude neměnný — se třeba rozhodne že pro seznam s jedním prvkem použije jinou implementaci než pro seznam s více prvky).

Object pool se používá když máme objekty, které je efektivnější recyklovat než vytvářet odznova. Používá se např. na mobilních platformách, které používají jazyky s garbage collectorem, kde konstrukce a následné čištění nepotřebných objektů způsobuje vysokou zátěž procesoru.

Singleton se používá pokud chceme aby existovala pouze jedna instance nějakého objektu. Většinou je tato instance vytvořena jako třídní (statický) atribut a konstruktor je ukryt, některé jazyky jako Kotlin mají speciální deklarace kdy namísto deklarace třídy `class Abc` lze rovnou deklarovat singleton `object Abc`.

Strukturální návrhové vzory

Kompozit tvoří hierarchii objektů které všechny implementují stejné rozhraní. Volání metody kompozitního objektu pak zavolá tuto metodu i ve všech obsažených objektech. Typickým příkladem je grafické API, kde můžu z několika jednoduchých tvarů seskládat jeden komplikovaný tvar.

Adaptér převádí rozhraní jedné třídy na rozhraní druhé třídy (např. pokud mám knihovnu pro práci s grafikou která definuje rozhraní s metodami pro získání a změnu barvy pixelu, a chci aby pracovala s třídou `Image` ve Windows API, tak vytvořím adaptér který zabalí instanci `Image` z Windows API, a poté implementuje rozhraní knihovny tak aby metody pro práci s pixely volaly ekvivalentní metody v rozhraní třídy `Image`).

Dekorátor zabaluje instanci třídy, a dává možnost přidat nebo odebrat chování metodám jejího rozhraní (např. pokud máme rozhraní reprezentující datový proud s metodou zápisu, tak můžeme vytvořit dekorátor který datový proud zabalí, a všechna data která projdou metodou zápisu třeba zkomprimuje; dekorátor pak bude možno použít pro zabalení jakékoliv implementace datového proudu — např. soubory, síťovou komunikaci, data v paměti).

Fasáda definuje jednoduché rozhraní ke komplikovanému systému. Pokud mám nějakou komplikovanou sekvenci metod které jsou volány na několika objektech, můžu vytvořit fasádu která celou sekvenci zahrne do jedné metody (např. pokud bych měl emulátor herní konzole který se skládá z mnoha komponent, tak komplikované sekvence jako je např. spuštění systému, načtení herní kazety, apod. vyjádřím fasádou která je shrne do samostatných metod jako `startSystem` a `loadCartridge`).

Behaviorální návrhové vzory

Iterátor umožňuje postupně procházet sekvenci prvků nezávisle na její vnitřní struktuře. Konkrétní iterátor nad polem prvků bude fungovat trochu jinak než iterátor lineárního seznamu, ale nad konceptem iterátorů můžeme stavět řadu funkcí (například velkou část LINQ v .NETu).

Observer je vzor o oznamování a příjmu událostí. Existuje příjemce a pozorovaný objekt, příjemce sám sebe registruje u pozorovaného objektu. Pozorovaný objekt si uchovává seznam všech příjemců, a v momentu kdy nastane pozorovaná událost odešle všem příjemcům oznámení pomocí metody. Je na něm založen systém *eventů* v .NETu, a je často využíván v .NETovských API (např. event pro změnu velikosti okna, dokončení práce na pozadí, časovače).

Strategie zapouzdřuje implementaci nějakého algoritmu s cílem vybrat konkrétní strategii za běhu programu. Strategii můžeme předat parametrem, popřípadě konstruktorem a mít ji jako atribut objektu, což navíc umožňuje nahradit dědičnost kompozicí.

Visitor reprezentuje algoritmus, který provádí nějakou akci s jednotlivými prvky nějaké struktury, ale není závislý na přesné podobě samotné struktury (např. pokud mám XML dokument, tak třída XML dokumentu má metodu která přijme visitora, začne procházet celou strukturu, a postupně předává visitorovi všechny elementy a atributy; visitor pak může třeba provést validaci obsahu nebo převádět data do jiného formátu).

Objektově orientované paradigma. Pojmy třída, objekt, rozhraní. Základní vlastnosti objektu a vztah ke třídě. Základní vztahy mezi třídami a rozhraními. Třídní vs. instanční vlastnosti.

Objektově orientované programování je paradigma programování které je založeno na konceptu objektů.

Objekt je něco, co obsahuje data (atributy) a operace (metody) pomocí kterých objekty mezi sebou komunikují. Objekt je instancí nějaké *třídy*.

Třída je předpis, který definuje typy atributů které bude každá instance obsahovat a jejich počáteční hodnoty, a metody zároveň s jejich implementací. Každá třída má alespoň jeden *konstruktor*, což je funkce která vytvoří instanci třídy (to se většinou děje implicitně, tj. konstruktor má implicitní návratovou hodnotu kterou je instance třídy).

Třídní vlastnost jsou atributy nebo metody které jsou definovány staticky, tedy patří nějaké třídě ale za běhu programu existují pouze jednou a v případě atributů mají jednu globální hodnotu.

Rozhraní určuje metody (a v některých jazycích i atributy) které má třída implementovat (obsahovat) a veřejně zpřístupnit. Rozhraní samotné většinou obsahuje pouze signatury metod bez implementace. Cílem je definovat kontrakt pro komunikaci mezi objekty, který ukrývá interní implementaci konkrétních tříd. Jedna třída může implementovat libovolný počet rozhraní.

Abstraktní třída je něco mezi třídou a rozhraním — může definovat atributy i metody, a některé i implementovat, ale nelze je konstruovat — k tomu už potřebujeme konkrétní třídu která abstraktní třídu rozšiřuje.

Klíčové vlastnosti OOP:

- Kompozice (objekty mohou obsahovat další objekty)
- Zapouzdření (ukrývá vnitřní reprezentaci objektů a dovoluje přístup zvenčí pouze pomocí veřejných metod)
- Dědičnost (umožňuje aby jedna třída rozšiřovala druhou, a tvořila tak hierarchii tříd; potomek pak dědí všechny vlastnosti rodiče, a může přidávat vlastní atributy a metody, a rozšířit nebo změnit chování které zdědil)
- Polymorfismus (umožňuje rodičovský typ nebo rozhraní nahradit konkrétní instancí třídy která je buď potomkem rodičovského typu, nebo implementuje požadované rozhraní)

Dědičnost může být i vícenásobná, kdy jeden potomek dědí ze dvou rodičů. Většina moderních jazyků to nepovoluje, namísto toho používají rozhraní, a aby se zjednodušila práce a rozhraní se trochu přiblížila vícenásobné dědičnosti, tak některé jazyky dovolují definovat implementaci metod rozhraní.

Implementace polymorfismu je možná např. pomocí tabulky virtuálních metod. Protože každá třída která dědí z jiné nebo implementuje rozhraní může mít svou vlastní verzi každé metody, tak pro každou třídu existuje kompilátorem generovaná tabulka, která obsahuje adresu každé metody která ve třídě existuje. Při konstrukci instance třídy je s objektem uložen odkaz na tabulku dané třídy. Kompilátor pak při volání metody objektu použije adresu metody kterou najde v tabulce konkrétního objektu.

Mapování UML diagramů na zdrojový kód.

Třídní diagram — popsat:

- 4 úrovně viditelnosti
- Vícenásobné vazby (0..1 pomocí atributu který může být `null`, 1 pomocí atributu který nemůže být `null`, 0..* a 1..* pomocí seznamu)
- Dodatečné informace u atributů (typ, multiplicita (násobnost vazby), *readOnly*, hodnota na kterou je atribut inicializován)
- Vazbu závislosti (metoda která třídu používá jako parametr/návratovou hodnotu)
- Vazbu agregace (atribut + konstruktor který jej přijme)
- Vazbu kompozice (atribut + konstruktor který jej vytvoří)
- Dědičnost
- Interface + implementaci

Diagram aktivit — popsat:

- Akce (volání funkcí)
- Podmínky a cykly
- Rozdělení toku na více větví (např. pomocí vláken — nezapomenout na `join` — nebo asynchronních úkolů v `C#` kde je možno využít `await Task.WhenAll`)

Sekvenční diagram / diagram komunikace lze popsat jako sekvenci volání metod na různých objektech.

Správa paměti (v jazycích C/C++, JAVA, C#, Python), virtuální stroj, podpora paralelního zpracování a vlákna.

Pro správu paměti existuje řada technik:

- Většina lokálních proměnných, které jsou uvnitř funkcí nebo bloků kódu, jsou alokovány na zásobníku v momentu kdy kód vstoupí do tzv. rozsahu platnosti proměnné, tedy do oblasti kódu ve kterém je proměnná definována, a poté je automaticky uvolněna ze zásobníku v momentu kdy z té oblasti odejde. Tento systém je nutnost pro jakýkoliv jazyk podporující přímou nebo nepřímou rekurzi, nebo paralelismus (například u Fortranu toto platí pouze pokud explicitně proměnnou označíme jako *automatickou*). Často však zásobník nestačí, takže jakékoliv větší objekty jsou alokovány na haldě, což řeší následující odstavce.
- U jazyků jako jsou `C` a `C++` spravujeme alokaci a uvolnění paměti na haldě manuálně. Pokud alokujeme kus paměti a poté ho neuvolníme, dostaneme tzv. *memory leak*. Takto uniklou paměť dostaneme zpět až v momentu ukončení procesu — dnešní operační systémy spravují paměť každého procesu takže umí všechnu paměť uvolnit při ukončení procesu, horší je však když chceme aby proces běžel velmi dlouhou dobu (hrozí že využije všechnu dostupnout paměť a spadne), nebo ještě v horším případě pokud únik paměti nastane v samotném jádru systému nebo systémových ovladačích.
- Zajímavou možností manuální správy paměti je koncept vlastnictví paměti kterou popularizoval jazyk `Rust`. V něm každá proměnná (včetně referencí na haldě) má vlastníka, kterým je určitý rozsah platnosti (funkce, blok kódu, struktura). Pokud program odejde z rozsahu platnosti proměnné, pak je automaticky uvolněna, je však možno vlastnictví proměnné předat jinam

(např. předáním do funkce jako parametr, nebo návratem proměnné z funkce ven). Klíčem je, že kompilátor sleduje kdo je vlastníkem proměnné, a vyžaduje aby vždy měla v každý moment přesně jednoho vlastníka — tím pádem ví, že když program odejde z rozsahu vlastníka, tak může všechny proměnné daného vlastníka uvolnit. Jedná se tedy o manuální správu paměti, ale namísto programátora se o ni stará kompilátor.

- Automatickou správu paměti zařizuje *Garbage Collector*. Ten si uchovává informace o všech referencích na kus paměti na haldě které existují, a průběžně hledá kusy paměti na které již neexistují žádné reference. Přesněji, rozlišujeme *silné reference* a *slabé reference*; pokud na kus paměti neexistují silné reference, pak je uvolněna i když můžou stále existovat slabé reference. Typy GC:
  - Tracing collector pracuje s referencemi jako s hierarchií/stromem — pokud struktura obsahuje referenci, pak k ní existuje cesta stromem. Při úklidu projde celý strom referencí, a všechnu paměť ke které se nedostal uvolní. Tato kategorie zahrnuje řadu GC s různými přístupy k samotnému průběhu úklidu — *stop-the-world* (pozastaví celý program během úklidu) vs. inkrementální (rozdělí pozastavování programu do více fází) vs. konkurentní (běží zároveň s programem). Patří tam i generační GC, které rozdělují reference do skupin podle délky jejich života, a každou skupinu uklízejí v různých intervalech.
  - Reference counting collector u si u každého kusu paměti pamatuje počet referencí. Při vzniku reference je čítač inkrementován, při zániku je dekrementován, a paměť je uvolněna když čítač padne na nulu.

Virtuální stroj provádí emulaci nějakého systému. Může se jednat o emulaci celého operačního systému, kdy je možno provozovat na jednom *hostovi* několik operačních systémů najednou které jsou od sebe odděleny, ale v kontextu programování jde o vrstvu abstrakce která emuluje instrukce programovacího jazyka (případně jeho mezikódu) tak, aby fungoval nezávisle na platformě — program lze spustit na jakékoliv platformě na které běží virtuální stroj.

Typickým příkladem jazyka který využívá virtuální stroj je Java. Javovské zdrojáky jsou kompilovány do mezikódu (Java bytecode). Javovský virtuální stroj se skládá z několika částí:

- Class loader, který načítá kompilované soubory a jejich dependence které se mohou nacházet v knihovnách. Po načtení bytekódu je provedeno ověření jeho správnosti, aby se zabránilo kritickým chybám.
- Interpreter bytekódu, který může být doplněn (nebo i nahrazen) just-in-time kompilátorem. Interpreter překládá bytekód za běhu, což je většinou pomalé. To řeší just-in-time kompilátor, který kompiluje často používané kusy bytekódu do nativního kódu platformy na kterém program právě běží.
- Garbage collector pro automatickou správu paměti; Javovský stroj dokonce dává možnost vybrat si z několika implementací GC (všechny jsou založeny na *tracing collectoru*).

Paralelní zpracování označuje výpočty které běží současně. Můžeme je rozdělit do několika úrovní:

- Cluster několika počítačů
- Jeden počítač s vícejádrovým procesorem
- Jedno jádro procesoru na kterém běží více procesů
- Jeden proces ve kterém běží více vláken
- Paralelismus na úrovní instrukcí — moderní procesory umí provést několik nezávislých instrukcí najednou

Vlákno je nějaká část instrukcí která je prováděna nezávisle. Vlákna jsou součástí procesu a sdílí s procesem paměť. O vlákna se stará *plánovač* operačního systému, který rozhoduje na kterém jádru procesoru každé vlákno běží, a protože vláken je většinou více než jader procesoru, tak se stará i o přepínání vláken (jedno vlákno na pozastaví a nahradí jej jiným vláknem, aby každé vlákno mělo šanci chvíli běžet). Při práci s vlákny musíme dávat pozor na časté problémy:

- *Race condition* nastane když dvě vlákna přistupují ke sdílené paměti ve stejný moment, a alespoň jedno z nich paměť mění. Řešením je synchronizační mechanismus *mutex* který dočasně zamkne přístup k paměti (popřípadě změna návrhu programu tak, aby nepotřeboval sdílenou paměť).
- *Deadlock* nastane při špatném použití *mutexu*, kdy první vlákno čeká až druhé vlákno uvolní jeden *mutex*, a druhé vlákno čeká až první vlákno uvolní druhý *mutex*, a tím pádem se obě vlákna zaseknou.

Zpracování chyb v moderních programovacích jazycích, princip datových proudů – pro vstup a výstup. Rozdíl mezi znakově a bytově orientovanými datovými proudy.

Zpracování chyb můžeme rozdělit do 3 kategorií:

- Low-level API u systémových funkcí hlásí chyby návratovou hodnotou. Typicky např. v `C` při volání systémové funkce, která může selhat, bude její návratovou hodnotou nula pro úspěch, nebo jiné číslo označující kód chyby, a je pak na programátorovi aby chyby ošetřil (což se vždy neděje).
- Zlepšením této idey jsou monadické typy, které vychází z funkcionálního programování, ale čím dál častěji je nalézáme v dalších jazycích. Namísto chybového kódu funkce vrací strukturu, která zapouzdřuje možnost úspěchu i chyby, kterými jsou například:
  - Generický typ `Either<L, R>` který má dvě strany `L` a `R`, jedna z nich označuje opravdovou návratovou hodnotu funkce za předpokladu že uspěla, a ta druhá označuje strukturu která popisuje chybu za předpokladu že funkce neuspěla. Například při čtení souboru můžeme definovat funkci vracející typ `Either<byte[], IOError>`, ve které se bude nacházet buď obsah souboru jako pole bajtů, nebo struktura `IOError` s informacemi o tom proč čtení souboru neuspělo. Programátor pak musí počítat s oběma možnostmi.
  - Generický typ `Option<T>` se používá jako náhrada `null` hodnoty — struktura buď obsahuje data typu `T`, nebo neobsahuje nic, což opět nutí programátora aby výsledek funkce explicitně ošetřil.

  Výhodou, a to proč těmto typům říkáme monády, je možnost kompozice operací které transformují její obsah — pro zjednodušení uvedu příklad s dvěma funkcemi `read: String -> Option<byte[]>` a `decode: byte[] -> Option<Image>`, kde chci přečíst soubor a poté jeho obsah dekódovat jako obrázek. V obou operacích může nastat chyba. Monáda definuje funkci `compose`, kterou použiju takto: `read("a.txt").compose(decode)`. Pokud čtení souboru uspěje, pak `compose` zavolá funkci `decode` nad přečtenými bajty, a jejím výsledkem je buď `Image` nebo `null`. Pokud čtení souboru neuspěje, tak už na začátku je výsledkem `null` a `compose` na tom nic nezmění. Tím pádem můžu zřetězit libovolný počet transformačních operací, a až na konci zjišťovat zda celý řetězec operací uspěl nebo ne.

  Samozřejmě reálně bychom chtěli vědět kde přesně se něco pokazilo, ale určitě není problém domyslet si podobný proces pro typ `Either` který namísto propagace `null` bude propagovat první chybu která nastala.

- Mnoho moderních jazyků podporuje *výjimky*. Pokud nastane chyba (výjimečná situace), pak funkce vyhodí výjimku což okamžitě ukončí provádění funkce, a kontrola je předána kódu který výjimku zachycuje — tím je tzv. `try/catch` blok který se většinou nachází v nějaké nadřazené funkci, takže se program vrací zpět volanými funkcemi až narazí na první `try/catch` blok který výjimku zachytí, a odtamtud pokračuje.

  Jednou z nevýhod je, že vyhození výjimky je náročnější na výkon než jednoduché vrácení chybové hodnoty nebo monadického typu, proto není dobré je používat v místech kde očekáváme velké množství chyb. Například v `C#` existuje pro převod textu na číslo `int.Parse` a `int.TryParse`, z nichž první vyhodí výjimku a druhá používá návratovou hodnotu pro oznámení úspěchu/neúspěchu, a `out` parametr pro předání převedeného čísla.

Datový proud je sekvence dat která postupně pramení z nějakého zdroje, nebo naopak cílem do kterého jsou data postupně odesílána — zdrojem i cílem může být např. soubor nebo síťový prvek, ale existují i nekonečné zdroje (např. `/dev/urandom`, nebo funkce generující prvočísla) a nekonečné cíle (např. `/dev/null`). V některých případech, například u normálních souborů na disku, proudy podporují skok na danou pozici v proudu.

Většinou se setkáme s bajtovými proudy, kde data čteme po celých bajtech (8 bitů), ale v některých případech jako je audio, video, a obecně jakákoliv komprimovaná data, vyžaduje zpracování po jednotlivých bitech. Rozdíl je pouze v tom jak na data nahlížíme, moderní procesory pracují s bajtem jako nejmenší jednotkou paměti, takže pokud zpracováváme data po bitech tak stejně je musíme číst po bajtech a poté použít bitové operace pro extrakci jednotlivých bitů. Na druhou stranu, pokud pracujeme s textem v určitém kódování, tak většina programovacích jazyků má knihovny které interně pracují s bajtovými proudy, ale převádí mezi skupinami bajtů a datovým typem reprezentujícím buď jednotlivé znaky nebo řetězce.

Moderní operační systémy podporují proudy i v rámci vstupu a výstupu programu. Typickým případem užití jsou Unixové konzolové programy pro které je možnost propojování výstupu jednoho programu se vstupem jiného programu klíčovou částí jejich designu. Přestože to funguje všude, tedy i na Windows, to že v Unixu se s hardwarem, sokety, dokonce i informacemi o systému, zachází jako se soubory (přesněji souborovými deskriptory) ze kterých lze proudově číst nebo do kterých lze proudově zapisovat, to výrazně zjednodušuje práci programátorům — pokud umím pracovat s obecným proudem dat, tak můžu programově komunikovat téměř s čímkoliv bez toho, abych se zabýval tím, odkud ta data přicházejí nebo kam směřují.

Jazyk UML – typy diagramů a jejich využití v rámci vývoje.

Unified Modeling Language je jazyk pro modelování a vizualizaci návrhu softwarového systému. Zahrnuje řadu diagramů které dělíme na *strukturální diagramy* a *diagramy chování* (ty zahrnují i podskupinu *diagramů interakcí*). Jejich využití ve fázích vývoje:

- Specifikace požadavků — diagram případů užití, diagram aktivit, sekvenční diagram (popř. diagram komunikace)
- Návrh a implementace — diagram tříd, diagram objektů, sekvenční diagram (popř. diagram komunikace), stavový diagram, diagram komponent, diagram kompozitní struktury, diagram nasazení

Strukturální diagramy

Diagram tříd zobrazuje třídy objektově orientovaného programování, jaké atributy a metody každá třída obsahuje, a jak jsou třídy propojeny. V každé třídě definuje atributy a metody, jejich typy, a

viditelnost:

+ Public

- Private

Protected

~ Package

Statické atributy a metody jsou podtrženy. Atributy v moderních verzích UML mohou mít dodatečné informace:

`+attr: int [0..1]` označuje multiplicitu vazby (v tomto případě může být `null`)

`+attr: int {readOnly}` označuje atribut pouze pro čtení

`+attr: int = 5` označuje počáteční hodnotu atributu

Mezi třídami jsou definovány vztahy:

- Závislost (přerušovaná čára, jednoduchá šipka, používá se pokud třída vytváří nebo má v parametru metody třídu jinou)
- Dědičnost (plná čára, prázdná trojúhelníková šipka vede z potomka do rodiče)
- Realizace (přerušovaná čára, prázdná trojúhelníková šipka vede z třídy do rozhraní které daná třída implementuje)
- Agregace (plná čára, prázdný diamant je na straně třídy která obsahuje ale nevlastní instance druhé třídy — jsou jí někde předány)
- Kompozice (plná čára, plný diamant je na straně třídy která obsahuje a vlastní instance druhé třídy — při zániku vlastníka zanikne i instance druhé třídy)

Další informace: `https://visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/`.

Diagram objektů zobrazuje instance tříd v jeden konkrétní moment. Podobá se třídnímu diagramu, ale namísto obecných tříd zobrazuje konkrétní instance tříd, a hodnoty jejich atributů.

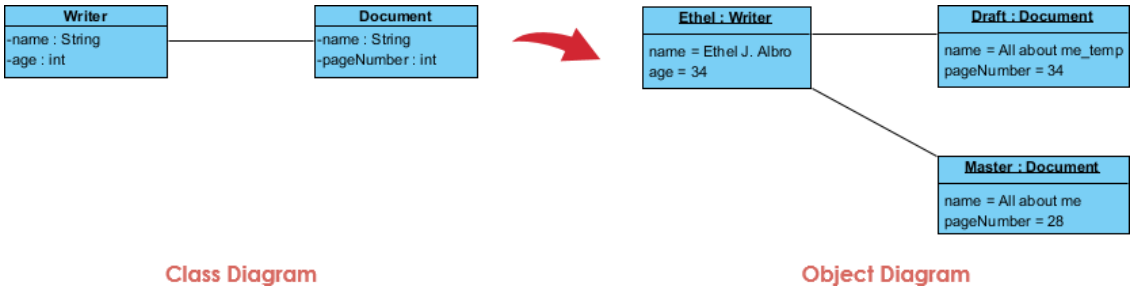


Diagram komponent zobrazuje komponenty celého systému (a jejich složení z menších podkomponent), závislosti mezi komponentami, a rozhraní pomocí kterých komponenty mezi sebou komunikují.

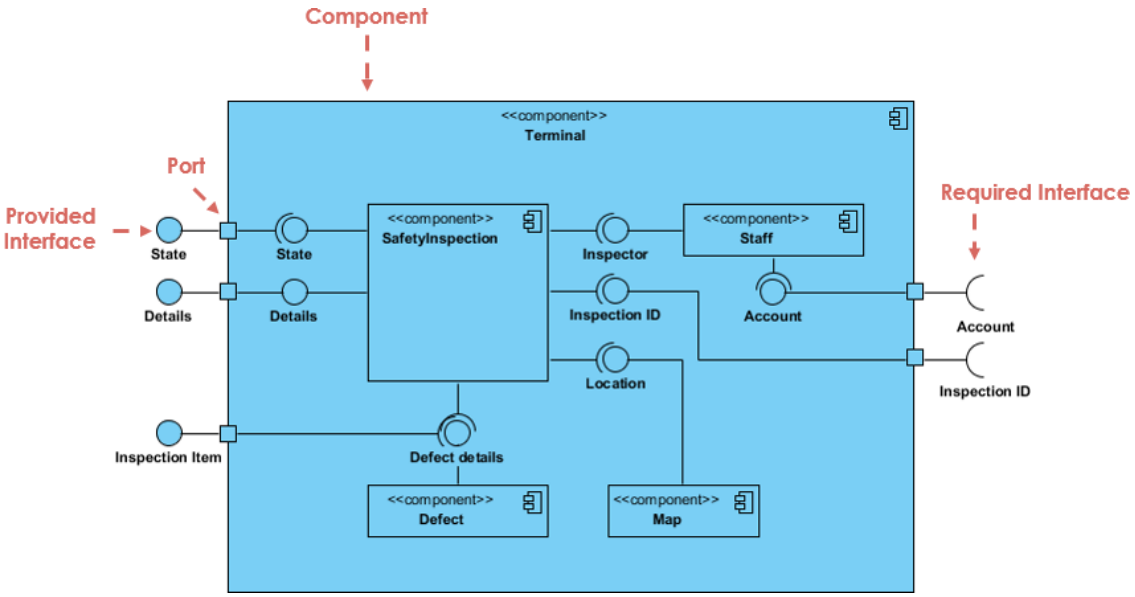


Diagram kompozitní struktury zobrazuje součásti konkrétní komponenty a jak jsou propojeny.

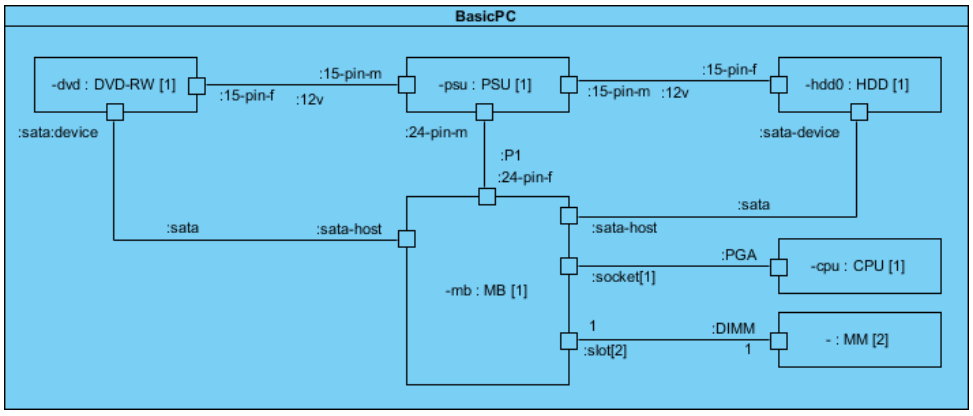


Diagram nasazení zobrazuje fyzickou strukturu komponent. Například u webů by diagram zobrazil rozložení a propojení webových serverů, databázových serverů, API serverů, atd.

Diagram balíčků a diagram profilů asi nejsou až tak důležité.

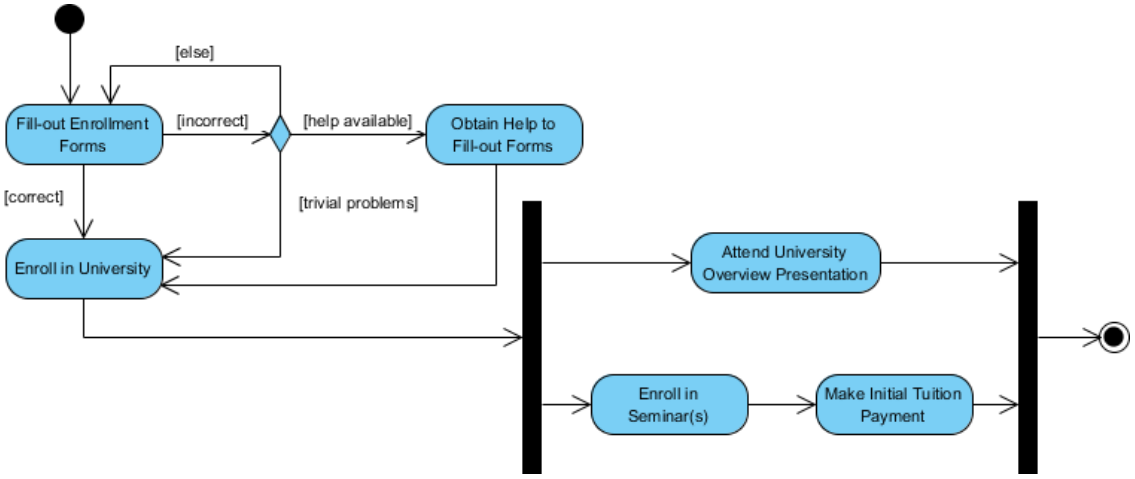
Diagramy chování

Diagram užití zobrazuje aktéry (uživatele s určitou rolí, externí systémy), a případy užití což jsou akce které aktéři provádí s různými systémy. Mezi akcemi můžou existovat 3 speciální vztahy:

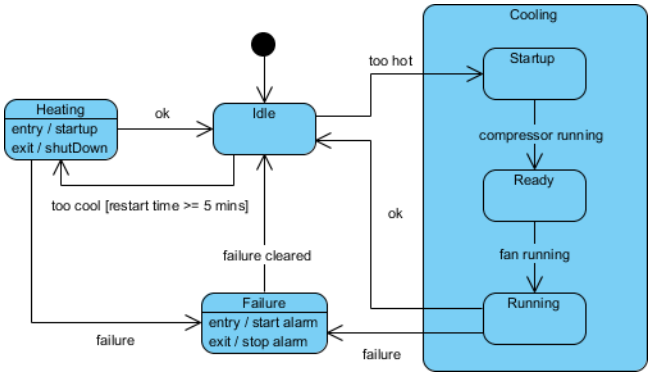
- `<<include>>` říká, že jeden případ užití zahrnuje druhý. Takto lze definovat akce které jsou součástí několika větších akcí. Značíme přerušovanou šipkou označenou `<<include>>` ve směru od větší akce do zahrnuté akce.
- `<<extend>>` přidává k případu užití další, volitelné případy užití. Značíme přerušovanou šipkou označenou `<<extend>>` ve směru od volitelné akce k hlavní akci.
- Generalizace popisuje v podstatě dědičnost dvou případů užití, tedy že jeden případ užití je speciálním případem druhého. Značí se plnou čarou se šipkou vedoucí z potomka do rodiče.

Diagram aktivit popisuje řídicí tok procesu. Aktivita začíná v inicializačním bodu, a prochází šipkami. Ty mohou vést do elips označující akce (konkrétní úloha), nebo diamantů označující rozhodnutí ze kterého se řídicí tok větví (popř. spojují více větví do jedné). Používají se také černé obdélníky které rozdělují řídicí tok na více větví (*fork*), které jsou provedeny paralelně, a poté opět spojeny dohromady

(*join*). *Swimlane diagram* rozšiřuje diagram aktivit o pásy, které určují který aktér provádí každou akci.



Stavový diagram popisuje objekt podobný konečnému automatu. V diagramu je seznam stavů ve kterých se objekt může nacházet, přechody které spojují dva stavy a události které způsobí přechod z jednoho stavu do druhého, a mohou se v něm nacházet i akce které nastanou na začátku nebo konci přechodu. Jedním zajímavým rozdílem oproti klasickým diagramům konečných automatů jsou vnořené stavy, což může být jednoduchá sekvence stavů kterou lze např. použít ve více diagramech, nebo i několik automatů které jsou spuštěny souběžně — to umožňuje popsat velmi složitou logiku.



Diagramy interakcí

Sekvenční diagram zobrazuje komunikaci mezi objekty se zdůrazněním pořadí předávaných zpráv. Horizontálně jsou rozloženi aktéři a objekty. Vyplněný obdélník pod aktérem nebo objektem zdůrazňuje aktivitu. Plná čára s šipkou znázorňuje předání zprávy, přerušovaný čára s šipkou návratovou hodnotu. Pokud přerušovaná čára vede do objektu, jedná se o konstrukci. Plná čára vedoucí z objektu do stejného objektu je předání zprávy do sebe sama (např. volání pomocné funkce nebo cyklus); pokud navíc vede do menšího obdélníku, jedná se o rekurzivní funkci.

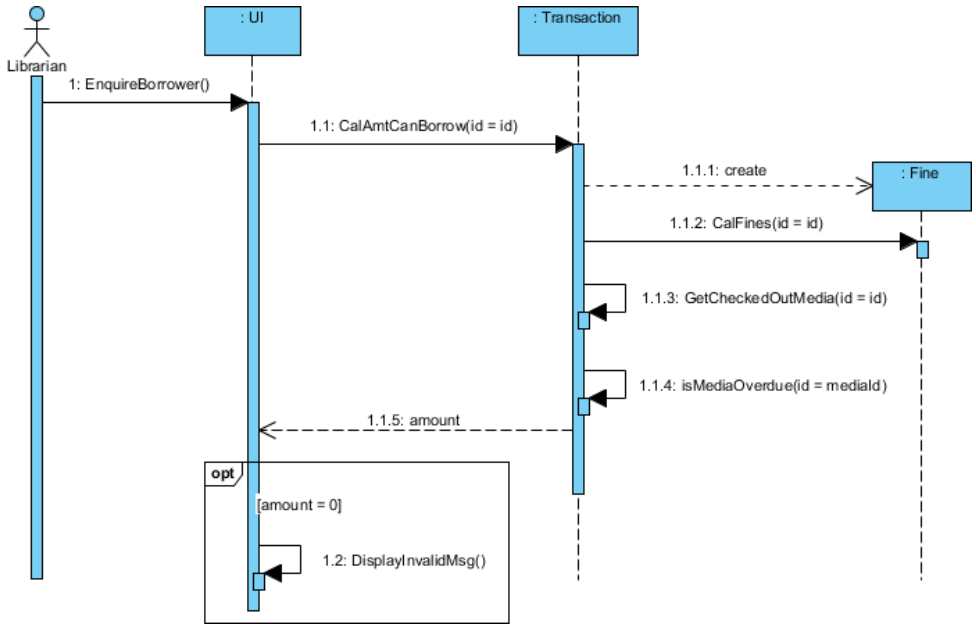


Diagram komunikace obsahuje stejné informace jako sekvenční diagram, jen je zobrazuje jiným způsobem — rozložení sekvenčního diagramu zdůrazňuje pořadí předávání zpráv za cenu menší přehlednosti objektů mezi kterými komunikace probíhá, zatímco u diagramu komunikace to je přesně naopak (zdůrazňují objekty a komunikační cesty, za ceny menší přehlednosti pořadí zpráv).

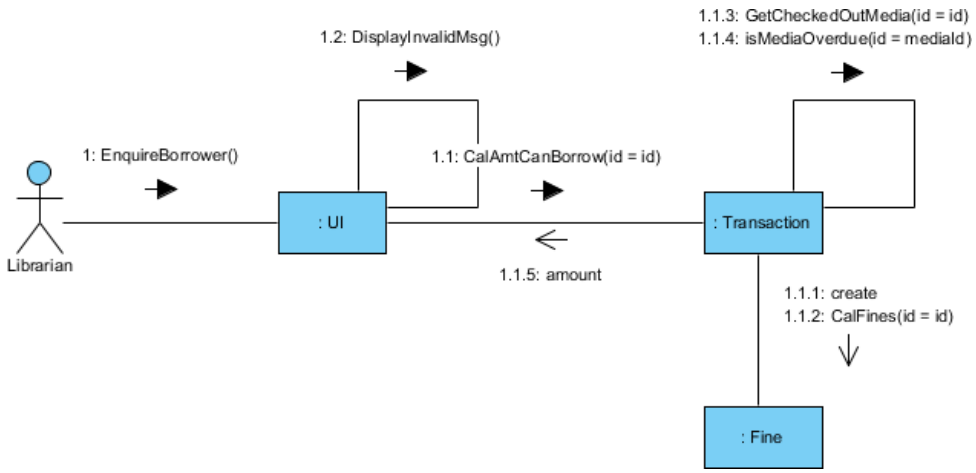
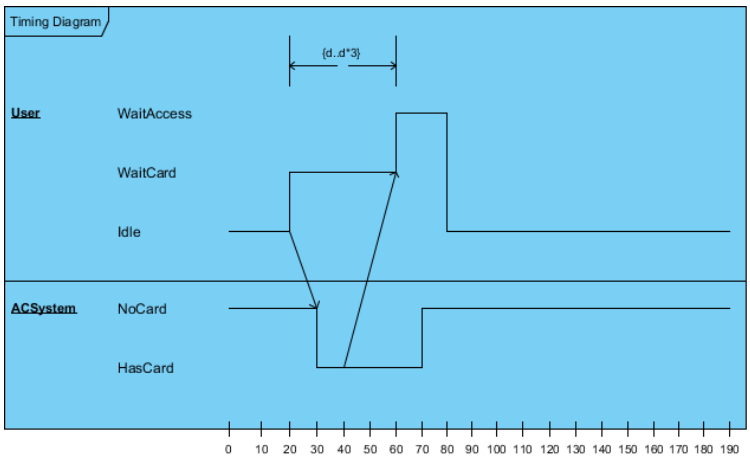
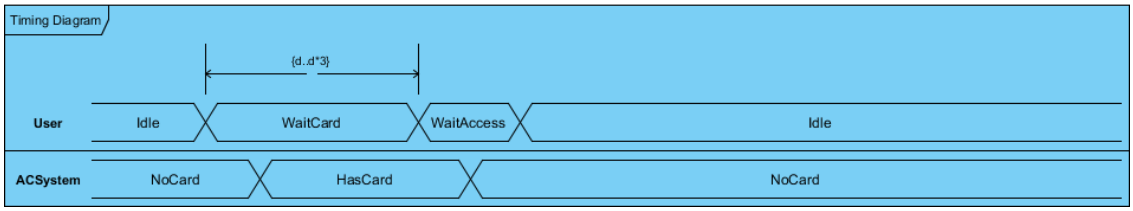


Diagram přehledu interakcí je podobný diagramu aktivit, ale aktivitami jsou v tomto případě diagramy interakcí (mohou být nakresleny celé, nebo jen jako odkazy).

Diagram časování je podobný sekvenčnímu diagramu, ale čas proudí zleva doprava a cílem diagramu je znázornit interakce objektů v daném časovém úseku.


Timing Diagram
{d..d*3}
User
WaitAccess
WaitCard
Idle
ACSystem
NoCard
HasCard
0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170 180 190


Timing Diagram
{d..d*3}
User
Idle
WaitCard
WaitAccess
Idle
ACSystem
NoCard
HasCard
NoCard

Databázové a informační systémy

Modelování databázových systémů, konceptuální modelování, datová analýza, funkční analýza; nástroje a modely.

VIZ VOLITELNÝ PŘEDMĚT.

Relační datový model, SQL; funkční závislosti, dekompozice a normální formy.

VIZ VOLITELNÝ PŘEDMĚT.

Transakce, zotavení, log, ACID, operace COMMIT a ROLLBACK; problémy souběhu, řízení souběhu: zamykání, úroveň izolace v SQL.

VIZ VOLITELNÝ PŘEDMĚT.

Procedurální rozšíření SQL, PL/SQL, T-SQL, triggery, funkce, procedury, kurzory, hromadné operace.

VIZ VOLITELNÝ PŘEDMĚT.

Základní fyzická implementace databázových systémů: tabulky a indexy; plán vykonávání dotazů.

VIZ VOLITELNÝ PŘEDMĚT.

Objektově-relační datový model a XML datový model: principy, dotazovací jazyky.

VIZ VOLITELNÝ PŘEDMĚT.

Datová vrstva informačního systému; existující API, rámce a implementace, bezpečnost; objektově-relační mapování.

VIZ VOLITELNÝ PŘEDMĚT.

Distribuované SŘBD, fragmentace a replikace.

VIZ VOLITELNÝ PŘEDMĚT.

Počítače a sítě

Architektura univerzálních procesorů. Principy urychlování činnosti procesorů.

Architektura procesoru zahrnuje:

- Organizace a propojení komponent jako jsou registry, paměť, sběrnice, cache, výpočetní jednotky a koprocesory
- Typ instrukční sady (RISC — Reduced Instruction Set Computer, CISC — Complex Instruction Set Computer)
- Počet bitů se kterými přirozeně pracuje jako se základní datovou jednotkou (16-bit, 32-bit, 64-bit, dříve i např. 12-bit)
- Endianita (little-endian, big-endian)

Sběrnice je vodič pro přenos dat mezi komponentami.

Registr je část pracovní paměti. Nachází se velice blízko samotné logické jednotky, takže přístup k nim je velice rychlý, a některé registry mají v procesoru speciální význam pro některé instrukce.

Von Neumannovo schéma je velmi starou koncepcí jednoduchého univerzálního počítače. Podle něj se počítač skládá z:

- Řídicí jednotka která zpracovává instrukce a řídí všechny součásti
- Aritmeticko-logická jednotka provádějící výpočty
- Operační paměť obsahující instrukce programu i operační data
- Vstupní a výstupní jednotka

Dnešní počítače kombinují řídicí a aritmeticko-logické jednotky, a další koprocesory do jediné centrální jednotky (CPU). Dále Von Neumannovo schéma nepočítá s multitaskingem, a další spoustou funkcí a optimalizací které moderní procesory podporují, ale je to velmi jednoduchý model který se hodí k popisu naprostého základu funkce procesoru.

Hardvarská architektura se od Von Neumanna moc neliší — jednou ze změn je oddělení paměti pro instrukce a pro data, takže je možné použít různé typy pamětí (např. RWM pro operační paměť, ROM pro programovou paměť). Počítače vycházející z tohoto typu architektury existují, ale jedná se spíš o specializované zařízení a mikropočítače.

Urychlování procesorů

Základní možnosti urychlení:

- Zvýšení frekvence. To většinou potřebuje větší voltáž, a tím pádem větší příkon což generuje více tepla — je potřeba buď lepší chlazení, nebo zmenšit tranzistory což se děje každých pár let.

- Zvýšení počtu jader. To taktéž potřebuje větší příkon, takže se zvyšujícím se počtem jader se často snižuje frekvence. Samozřejmě je potřeba aby program který na procesoru běží uměl všechna jádra využít.

Cache (mezipaměť) je kopie dat z operační paměti, která je fyzicky mnohem blíže procesoru než operační paměť, takže je k ní přístup mnohem rychlejší. Typicky tvoří hierarchii dvou nebo tří cachí, kdy L1 je nejblíže individuálnímu jádru a má nejmenší kapacitu, a L3 je nejvzdálenější, největší, a navíc sdílená mezi jádry. Často se používá *prefetching*, který kromě konkrétní adresy operační paměti, ke které bylo přistoupeno, navíc načte a uloží do cache paměť která je poblíž, protože je pravděpodobné že se k ní bude v blízké době také přistupovat.

Pipelining využívá toho, že zpracování instrukce je rozděleno do několika fází, a často je možné — pokud nejsou instrukce na sobě závislé — aby se fáze překrývaly. Fázemi mohou být (1) načtení instrukce, (2) dekódování instrukce, (3) provedení instrukce, (4) zápis do paměti. V takovém případě by pipelining po načtení instrukce a přestupu do fáze dekódování okamžitě začal načítat druhá instrukce; po přestupu první instrukce do fáze provedení by druhá instrukce přešla do fáze dekódování, a začala by se okamžitě načítat třetí instrukce; atd. Kromě možné závislosti instrukcí, např. pokud přistupují ke stejné paměti, nastává problém u skokových instrukcí.

Predikce skoku se snaží odhadnout, která instrukce bude následovat po (podmíněné) skokové instrukci, a od toho bodu bude pokračovat pipelining. Pokud se predikce netrefí, pak musí procesor celý proces pipeliningu od tohoto bodu zahodit, a začít znova. Predikce může být statická (tj. vždy se rozhodne určitým způsobem), náhodná, nebo adaptivní (procesor si pamatuje jeden nebo i více předchozích výsledků podmínky, a rozhodne se podle historie každé skokové instrukce).

Speciální SIMD instrukce provedou jednu operaci nad větším množstvím dat najednou. Příkladem jsou SSE nebo AVX instrukce v architektuře x86, kdy např. AVX-512 umí pracovat s 512 bity najednou jako se skupinami rozdělenými po 8, 16, 32, nebo 64 bitech, a obsahuje instrukce které tyto skupiny interpretují buď jako celá čísla nebo čísla s pohyblivou řadovou čárkou, a provedou na nich aritmetické nebo bitové operace.

Základní vlastnosti monolitických počítačů a jejich typické integrované periférie. Možnosti použití.

Monolitický počítač (také mikropočítač, jednočipový počítač) je kompaktní integrovaný obvod obsahující celý mikropočítač. Využívá se např. ve vestavěných systémech, malých přenosných zařízeních jako jsou kalkulačky nebo dálkové ovladače, mohli bychom tam zařadit i třeba Raspberry Pi který je založen na ARM architektuře takže na něm může běžet operační systém založený na Linuxu nebo speciální ARM verzi Windows.

Často jsou založeny na Harvardské architektuře (viz předchozí otázka) a instrukční sadě typu RISC, ale nemusí tomu tak být vždy. Pokud jsou založeny na Harvardské architektuře, skládají se z:

- Procesor + oscilátor generující hodinový signál (určí taktovací frekvenci)
- Operační paměť (RWM, např. RAM)
- Programová paměť (ROM, např. EPROM, EEPROM, Flash)
- Vstupní a výstupní porty (sériový port, GPIO piny, USB)

Na události externích periferií můžeme reagovat pomocí pravidelných kontrol stavu periferie (polling), nebo tak že periferii dovolíme dočasně přerušit vykonávání programu a okamžitě reagovat na událost (přerušení/interrupt).

Typické periferie

Časovače které umožní provádět události v pevných intervalech které se řídí hodinovým signálem.

Watchdog je časovač který resetuje systém nebo jinak převezme řízení pokud přestane hlavní program odpovídat (např. při zacyklení). Je hodně důležitý pro zařízení ke kterým nemá člověk fyzicky přístup, např. cokoliv co jsme odeslali do vesmíru, kde softwarová chyba nebo vesmírné záření by mohlo způsobit nekonečný cyklus.

Převodníky analogového a digitálního signálu se používají v jakémkoliv zařízení které má senzory na vnější vlivy které jsou analogové (teploměr, mikrofon, kamera, radar) a je potřeba je převést na digitální bity, nebo naopak používají digitální signál pro ovlivnění analogových součástek (LED světla, reproduktory).

Struktura OS a jeho návaznost na technické vybavení počítače.

Operační systém je software, který je načten při startu systému a spravuje celý počítač — komunikuje s hardwarem pomocí *ovladačů*, spravuje veškerý software který na něm běží, a většinou implementuje základní služby jako jsou uživatelské účty, aktualizace, aplikace související s typickým využitím počítače, atd.

Jádro (kernel) je základem operačního systému. Při startu počítače se typicky načte boot loader z disku, který obsahuje informace o nainstalovaných operačních systémech, a předá řízení kernelu operačního systému který chceme spustit. Moderní operační systémy jsou často rozděleny do tzv. privilegovaných režimů, kde kernel má nejvíce privilegií a je oddělen od prostoru uživatelských programů které mají méně privilegií, a tím pádem menší šanci narušit stabilitu a bezpečnost systému. Rozlišujeme několik typů architektury kernelu:

- Monolitické — všechny systémové služby běží jako součást kernelu, takže mají stejná privilegia a limituje se počet přechodů mezi uživatelským a kernelovým prostorem, ale pokud nastane chyba v jedné části kernelu tak může shodit celý systém, a mohou náročnější na údržbu pokud jsou příliš velké. Příklad monolitického kernelu je ten Linuxový, kde např. všechny ovladače jsou součástí kernelu a jsou načteny podle potřeby.
- Mikrokernely — implementuje jen základní funkcionalitu, a poskytuje rozhraní pro implementaci systémových služeb a ovladačů odděleně jako uživatelské procesy. Nevýhodou pak je, že tento přístup vyžaduje mnohem více přechodů mezi uživatelským procesem a jádrem, což má vliv na výkon systému.
- Hybridní — je někde mezi monolitickým a mikrokernelem. Příkladem jsou moderní Windows a Mac kernely. Systémové služby a ovladače jsou implementovány odděleně, ale narozdíl od mikrokernelů mohou běžet ve stejném privilegovaném režimu jako jádro.

Ovladač je program který propojuje hardware a software. Operační systém definuje rozhraní které ovladač implementuje, a uživatelská aplikace nebo systémová služba využije pro provádění konkrétních úkonů:

- Windows Display Driver Model (WDDM) je rozhraní pro ovladače grafické karty na Windows
- Advanced Linux Sound Architecture (ALSA) je rozhraní pro ovladače zvukové karty na Linuxu
- a mnoho dalších...

Správa procesů a jejich zdrojů je důležitou funkcí operačního systému. Při spuštění aplikace musí systém vytvořit proces, přiřadit mu paměť (což u moderních systémů znamená vymezení virtuálního adresového prostoru pro proces aby byl každý proces izolován), načíst dynamické knihovny, pak za běhu procesu se stará o multitasking (přepíná mezi procesy tak rychle, že to vypadá jako by běžely všechny

procesy současně) a přiřazování zdrojů (např. souborových deskriptorů), a při ukončení procesu uvolní všechny zdroje které využil.

Souborový systém je součást operačního systému, která definuje jak jsou organizována data na disku, stará se o metadata (např. název souboru, datum posledního přístupu, přístupová práva, ikony). Operační systém definuje rozhraní pomocí kterého programy můžou přistupovat k souborům nezávisle na tom, který souborový systém je právě používán.

Síťové systémy jsou dnes dost důležité, takže ovladače a nástroje pro správu připojení k internetu, sdílení souborů, vzdálený přístup, apod. jsou často součástí základní instalace operačního systému.

Protokolová rodina TCP/IP.

OSI model byl první snahou globálně standardizovat komunikaci v sítích. Definuje 7 vrstev, kde každá vrstva plní jednu konkrétní funkci, a nižší vrstva zapouzdřuje data z vrstvy vyšší:

- *Aplikační vrstva* je nejvyšší vrstva která zahrnuje protokoly konkrétní aplikace (např. webový prohlížeč, databázový klient).
- *Prezentační vrstva* se stará o kódování dat z aplikační vrstvy, což může zahrnovat kódování textu, šifrování, a/nebo kompresi.
- *Relační vrstva* spravuje relaci, tedy udržování a synchronizace komunikace, ale může i spravovat přihlášení uživatele.
- *Transportní vrstva* rozděluje komunikaci do segmentů (a na druhé straně ji potom opět spojuje) a stará se o kontrolu chyb přenesených dat a případné požadavky na přeposlání segmentu.
- *Síťová vrstva* se stará o směrování komunikace v síti, tj. mezi dvěma síťovými uzly mezi kterými neexistuje přímé propojení.
- *Linková vrstva* se stará o přenos dat mezi dvěma přímo propojenými zařízeními.
- *Fyzická vrstva* definuje fyzické parametry zařízení a konverzi digitálních dat na signály které lze fyzicky přenést (např. jako elektrické, optické, nebo rádiové signály).

Ukázalo se ovšem, že model není praktický, takže namísto něj se používá model *TCP/IP* který 7 vrstev zjednodušuje na 4.

Aplikační vrstva spojuje vrchní 3 vrstvy OSI (aplikační, prezentační, relační). Patří tam např. HTTP, FTP, IMAP, SMTP.

- Když se podíváme např. na HTTP, tak vidíme že celý protokol definuje jak bude prováděna komprese a kódování dat pomocí hlaviček, a definuje *cookies* které se používají mj. pro správu uživatelské relace — to by právě bylo v modelu OSI rozděleno do více vrstev, ale v reálném světě má každý aplikační protokol trochu jiné potřeby tak TCP/IP říká že si tyto věci každý protokol definuje individuálně.
- Na druhou stranu, existují i protokoly které zajišťují jednu z těchto funkcí, a zabalují data jiného aplikačního protokolu — například TLS protokol který zajišťuje zabezpečení a integritu dat, a lze jej použít pro webové protokoly, emailové protokoly, Voice over IP, atd.

Transportní vrstva vychází z transportní vrstvy OSI. Patří tam např. TCP, UDP, a několik dalších.

- TCP vytváří kanál pro spolehlivou obousměrnou komunikaci. Rozděluje data do paketů, každý označí pořadovým číslem aby je mohl v cílovém zařízení správně seřadit, a opakovaně odesílá pakety které nedojdou do cíle nebo jsou poškozeny. Nevýhodou je vysoká režie která zpomaluje přenos, a zastavení celého přenosu i když se ztratí nedůležitý paket.

- UDP je alternativou pro aplikace které nevyžadují spolehlivost protokolu TCP — např. online hry, služby pro streamování audia a videa, VoIP. Pokud je u některých paketů potřeba spolehlivost, pak se o ni stará aplikační vrstva.

Internetová vrstva vychází ze síťové vrstvy OSI. Patří tam hlavně IP protokol, ale také rozšíření IP protokolu jako je IPsec, který přidává zabezpečení (takto lze zabezpečit aplikace které nedefinují žádný proces pro zabezpečení).

- IP protokol definuje adresy pro identifikaci zařízení v síti, zabaluje paket transportní vrstvy a přidává metadata o cílové adrese která je použita pro směrování paketu do cíle. Zahrnuje dvě verze (IPv4 a IPv6), z nichž adresní prostor IPv4 je již vyčerpán takže internet se pomalu, už přes 20 let, přesunuje na IPv6...

Linková vrstva (vrstva síťového rozhraní) vychází z linkové vrstvy OSI, s tím že fyzickou vrstvu si zajišťuje každé konkrétní zařízení a není specifikovaná přímo v TCP/IP modelu. Patří tam např. MAC vrstva, kterou využívá Ethernet, Wi-Fi, mobilní sítě jako LTE, atd., a související ARP protokol který slouží k identifikaci MAC zařízení v síti.

Metody sdíleného přístupu ke společnému kanálu.

Dělení metod sdíleného přístupu:

- Bezkolizní (nenastávají kolize; patří k nim např. předávání tokenu v síti Token Ring, a metody založeny na rozdělení přenosového kanálu jako jakou např. FDMA a TDMA)
- Kolizní s vyhýbáním (snaha se kolizím vyhnout, ale kolize můžou stále nastat; patří k nim např. CSMA/CA dnes používaná ve Wi-Fi komunikaci)
- Kolizní se zotavením (snaha rychlého zotavení při detekci kolize; patří k nim např. ALOHA která je starší, nebo CSMA/CD která byla dříve používaná v polo-duplexním Ethernetu, dnes Ethernet s kroucenou dvojlinkou propojuje pouze dva síťové prvky a je plně duplexní takže ke kolizím nedochází)

Token Ring slouží k sestavení LAN sítě v kruhové topologii. Metoda předávání tokenu funguje tak, že kruhem putuje speciální prázdný rámec. V momentu, kdy doputuje do stanice která chce vysílat data, tato stanice rámec přemění na datový rámec a pošle opět do sítě. Poté co datový rámec doputuje do cílové stanice je v rámci zaznačena speciální hodnota, kterou se oznámí odesílateli že rámec byl přijat; až tento finální rámec doputuje do odesílatele, je rámec vyprázdněn a začíná se odznova. Výhodou je že nemohou nastat kolize, nevýhodou je vyšší latence protože stanice musí čekat než do ní doputuje prázdný rámec než může odeslat data.

ALOHA je jedním z nejstarších protokolů pro sdílený přístup, ze kterého později vyšlo CSMA. Funguje tak, že každá stanice vyšle svá data kdykoliv chce, a pokud během přenosu zjistí že jiná stanice už vysílá data, tak musí chvíli počkat než data zkusí odeslat znova. Princip je velmi jednoduchý, ale zkolabuje při velkém zatížení sítě. Zlepšením je ALOHA se sloty, která definuje časový interval a povoluje odeslání dat pouze na začátku časového intervalu; výsledkem je dvojnásobná efektivita protože kolize může nastat pouze na začátku, zatímco u klasické ALOHY může nastat kdekoliv.

CSMA (Carrier-Sense Multiple Access) je rodina protokolů, které oproti protokolu ALOHA sledují aktivitu v síti ještě předtím, než se pokusí odeslat své vlastní data. Základní verze pouze zkontroluje aktivitu těsně před odesláním, takže kolize může nastat pokud dvě stanice začnou vysílat v přibližně stejný čas. Pokud kolize nastane, základní verze CSMA to už nezjistí a pokračuje ve vysílání.

CSMA/CA rozšiřuje CSMA o to, že každá stanice vyšle krátkou zprávu předtím než začne vysílat velké množství dat, aby se snížil risk kolize. Možným řešením *hidden node* problému je možnost (kterou však neimplementuje každé CSMA/CA zařízení) vyžadovat povolení ke komunikaci s cílovým

zařízením, které cílové zařízení v jeden moment přiřadí jen jednomu zařízení v síti a ostatní musí počkat.

CSMA/CD rozšiřuje CSMA o to, že vysílací stanice je schopna zjistit že došlo ke kolizi, a v takovém případě okamžitě přestane vysílat a začne čekat náhodnou dobu než zkusí vysílat znovu.

FDMA (Frequency-Division Multiple Access) je jednou z metod založenou na rozdělení přenosového kanálu. Rozděluje frekvenční pásmo do několik částí tak, aby každý mohl přenášet data na své vlastní frekvenci.

TDMA (Time-Division Multiple Access) funguje podobně, jenom namísto frekvenčního pásma rozděluje čas do slotů tak, aby každý přenášel data v deterministicky určených časových intervalech.

Problémy směrování v počítačových sítích. Adresování v IP, překlad adres (NAT).

Směrování je proces určení cesty v síti (respektive neurčuje se celá cesta, protože síť může být velmi komplikovaná, ale pouze jeden krok cesty mezi dvěma přímo propojenými zařízeními). Každý router (směrovač) v síti má *směrovací tabulku*, což je tabulka pravidel obsahující:

- Cílovou adresu a masku sítě (při příchozím paketu směrovač porovná zda se cílová adresa paketu shoduje s cílovou adresou nebo podsítí pravidla)
- Adresa zařízení do kterého bude paket odeslán, a rozhraní které je fyzicky propojuje
- Metrika, podle které se směrovač rozhoduje které pravidlo je nejlepší, pokud pro daný paket existuje pravidel více

Statické směrování znamená, že pravidla v tabulce jsou spravovány manuálně. Je vhodné např. pro menší sítě, nebo lze využít vedle dynamického směrování pro definici záložních cest pro případy kdy dynamické směrování selže. Výhodou je plná kontrola nad cestami v síti, nevýhodou je samozřejmě nemožnost automatické adaptace na změny a výpadky v síti.

Dynamické směrování se automaticky adaptuje na změny v síti. Kvůli obrovské velikosti internetové sítě je síť rozdělena do tzv. *autonomních systémů* které spravují poskytovatelé internetu. Směrování pak probíhá ve dvou vrstvách — pomocí jednoho z protokolu se směrují pakety mezi dvěma autonomními systémy, a pomocí dalšího protokolu se směruje v rámci jednoho autonomního systému.

BGP (Border Gateway Protocol) je dnes nejpoužívanější protokol pro směrování mezi autonomními systémy.

RIP (Routing Information Protocol) je jeden z nejstarších směrovacích protokolů; u internetových sítí je považován za interní protokol (tj. směruje uvnitř autonomního systému). Funguje pomocí *distance-vector algoritmu*. Router periodicky ohlašuje sousedům svou směrovací tabulku, kde jedinou metrikou je počet skoků na cestě mezi routerem a cílovou adresou nebo podsítí. Router tak nezná celou topologii sítě, ale pouze vzdálenost ke každému cíli. Nevýhodou protokolu RIP je velká prodleva mezi aktualizacemi tabulek, a periodicky velké vytížení sítě pokud se směrovací tabulka rozroste.

OSPF (Open Shortest Path First) je o něco modernější interní směrovací protokol. Oproti RIPu zná každý směrovač celou topologii sítě, a namísto periodického odesílání celých směrovacích tabulek jsou odesílány pouze změny topologie, a ty jsou odesílány okamžitě v momentu kdy nastanou. Pro vyhledání nejkratší cesty se používá Dijsktrův algoritmus, kde každá hrana (cesta mezi dvěma zařízeními) má nastavenou cenu podle její přenosové rychlosti.

IP adresa slouží k identifikaci zařízení v IP síti. Dnes existují dvě verze adres — původní IPv4, což je 32bitové číslo zapsané jako čtveřice bajtů, a novější IPv6 což je 128bitové číslo zapsané jako osmice hexadecimálních číslic, poskytující mnohem větší adresní prostor než IPv4 které už je úplně vyčerpané.

NAT (Network Address Translation) umožňuje aby pod jednou IPv4 adresou bylo více počítačů najednou, čímž se mimojiné obchází fakt že všechny IPv4 adresy již byly vyčerpány a přechod na IPv6 pravděpodobně nebude dokončen po zbytek století. V základu jde o přepis IP adresy a případně portu na síťovém rozhraní. Většinou se přepisuje mezi privátní a veřejnou IP adresou:

- Domácí routery používají NAT pro přepis mezi privátní IP adresou uvnitř sítě a veřejnou IP adresou kterou přiřadil poskytovatel internetu.
- Poskytovatelé internetu mají vyhrazené privátní IP adresy za které může skrýt zákazníky kteří budou sdílet veřejnou IP adresu. Jeden problém NATu je, že nastávaly problémy v případech kdy poskytovatel a domácí síť měli stejné privátní IP adresy, takže byla nutnost pro poskytovatele vyhradit úplně oddělený adresní prostor. Další problém NATu je např. nemožnost hostovat veřejné služby jako webhosting.

Bezpečnost počítačových sítí s TCP/IP: útoky, paketové filtry, stavový firewall. Šifrování a autentizace, virtuální privátní sítě.

Problém TCP/IP sítí je, že typicky jsou všechna data přenášena nešifrovaně, což způsobuje problém — každý router přes který paket projde může data nejen přečíst, ale i upravit, takže není zaručeno soukromí, integrita, a autenticita dat. Bezpečnost v TCP/IP se většinou řeší na úrovni aplikační vrstvy pomocí protokolů jako TLS, ale existuje i na internetové vrstvě ve formě protokolu IPsec.

TLS (Transport Layer Security), a starší SSL který TLS nahrazuje, jsou protokoly které zabalují data aplikační vrstvy TCP/IP. Tím zabezpečuje přenos nešifrovaných protokolů jako jsou HTTP, FTP, SMTP, a další. Při zahájení připojení se musí server a klient dohodnout jaký šifrovací algoritmus použít (např. AES který má dnes např. speciální procesorové instrukce pro zrychlení), a také si musí bezpečně předat dešifrovací klíč — k tomu slouží algoritmy jako RSA nebo Diffie-Hellman.

IPsec je soubor protokolů které zabezpečují přenos paketů na internetové vrstvě. Zahrnuje *Authentication Headers* který zajišťuje integritu dat a také chrání proti opakovanému zaslání stejného paketu (*replay attack*), a *Encapsulating Security Payloads* který samotná data šifruje.

Šifrování dělíme na symetrické a asymetrické. U symetrického je pro šifrování i dešifrování dat použit stejný klíč. U asymetrického jsou pro šifrování a dešifrování použity dva různé klíče, které jsou nějak matematicky propojeny — princip je založen na operacích které lze rychle spočítat, ale je velmi obtížné jít opačným směrem (např. součin prvočísel je vcelku rychlý, ale faktorizace výsledku zpět na prvočísla je velmi náročný. U TLS asymetrické šifrování použito pro předání dešifrovacího klíčem, a symetrické šifrování pro samotná data.

Integritu a autenticitu dat ověříme pomocí jedné z Message Authentication Code metod. Zpráva je označena tagem který je spočítán z obsahu zprávy a klíče který je mezi oběma stranami bezpečně předán předem. Příjemce z obsahu zprávy a klíče spočítá tag na své straně, a pokud se shoduje s tím který je odeslán společně se zprávou, tak víme že se zprávou nikdo nemanipuloval a že opravdu přišla od strany se kterou jsme bezpečně vyjednali klíč.

Útok MITM (Man-in-the-middle) je útok kde někdo sedí mezi dvěma stranami, a odposlouchává a případně mění zprávy které mezi nimi chodí. TLS toto řeší tak, že při připojení na cílový server je provedena výměna certifikátu, který je podepsán certifikační autoritou která je registrovaná v prohlížeči (popř. FTP klientovi, emailovém klientovi) nebo samotném operačním systému. Třetí strana nemůže dešifrovat obsah zprávy protože nemá dešifrovací klíč, a pokud by chtěla podstrčit falešný certifikát ke kterému klíč má, tak by jej příjemce zamítnul protože by nebyl podepsán certifikační autoritou.

Útok DoS (Denial-of-Service) je útok jehož cílem je znepřístupnit přístup k zařízení, respektive ke službě která na daném zařízení běží. Lze provést zatížením služby velkým množstvím požadavků které nestíhá zpracovat, nebo využitím chyby (např. nedostatku v rozdělování zdrojů, kdy vytvořím

požadavek který vytíží procesor nebo paměť, nebo interní chyby která způsobí pád celé služby). Také existují varianty DoS, např. distribuovaný DoS kdy botnet složený z mnoha počítačů útočí na jeden cíl, nebo amplifikovaný DoS kdy útočník nějaké externí službě zašle malé množství dat a ta vrátí (na zfalšovanou návratovou IP adresu) několikanásobné množství dat (např. NTP servery, které slouží k hlášení času obrovskému množství zařízení, měly příkaz který způsobil přes 500násobnou amplifikaci).

Útok ARP spoofing je jednou z možností jak provést MITM nebo DoS útoky. ARP (Address Resolution Protocol) slouží k asociaci fyzické MAC adresy s IP adresou v síti. Principem útoku je rozeslání falešné ARP zprávy do sítě tak, aby ostatní zařízení si mysleli že útočník je např. router, a začali na toto zařízení odesílat zprávy namísto aby je odesílali do opravdového routeru. Poté může útočník provést MITM (číst a upravit zprávy) nebo DoS (zprávy zahodit).

Paketový filtr (základní typ firewallu) je jednou z možných mitigací DoS útoku, a obecně analýzou paketů může zabránit i jiným pokusům o útok. Jedná se o sadu pravidel, podle kterých se síťové zařízení rozhodne zda paket pustit dál nebo jej zahodit. Pravidla můžou zahrnovat kontrolu zdrojových a cílových IP adres a portů, inteligentnější zařízení mohou provádět i hlubokou inspekci která analyzuje i obsah paketu. Stavové filtry nenahlíží pouze na individuální pakety, ale sledují celé relace probíhající přes TCP kde je jasné navázání a ukončení relace, ale i UDP kde filtr většinou předpokládá že první UDP datagram jakoby započne relaci, a po určitém čase kdy neprojdou žádné UDP datagramy je relace ukončena.

Virtuální privátní síť slouží k propojení počítačů do privátní sítě prostřednictvím veřejné sítě (např. připojení do intranetu ve škole nebo zaměstnání prostřednictvím Internetu). K zabezpečení lze využít TLS a IPsec, ale i třeba DTLS což je TLS adaptované pro UDP.

Volitelný předmět — DAIS

Modelování databázových systémů, konceptuální modelování, datová analýza, funkční analýza (+ nástroje a modely z otázek IT).

Modelování databázových systémů se zabývá převodem požadavků na strukturu dat informačního systému do modelu, pomocí kterého bude navržena konkrétní databáze. Výsledkem této (datové) analýzy je konceptuální model.

Konceptuální model popisuje logickou strukturu databáze bez ohledu na konkrétní databázový systém. Určuje **co** je obsahem databáze:

- Entitní typ — typ objektu definován názvem a atributy které bude mít každá konkrétní...
- Entita — instance daného entitního typu
- Klíč — podmnožina atributů která jednoznačně identifikuje entitu
- Integritní omezení — říká zda je atribut volitelný, zda musí být hodnota unikátní, a případně omezení hodnot kterých může nabývat
- Vztah mezi dvěma entitními typy, s možnými kardinalitami:
  - 1:1 (vztah *je ředitel školy* — jeden člověk řídí jednu školu)
  - 1:N (vztah *pobývá v kanceláři* — profesor má jednu kancelář, ale v kanceláři může pobývat více profesorů najednou)
  - M:N (vztah *učí studenty* — profesor učí mnoho studentů, a každý student má několik profesorů na různé předměty)
  - Můžeme specifikovat i volitelnost (0:1, 0:N)

Zápis konceptuálního modelu:

- Lineární notace
  - `Student(id, first_name, last_name)`

    Student je název entitního typu, v závorce jsou atributy, podtržené *id* je klíč.
  - `TEACHES(Teacher: (0:M), Subject: (1:N), years)`

    TEACHES je název vztahu, ve vnější závorce jsou 2 entitní typy a 1 dodatečný atribut, ve vnitřních závorkách u entitních typů je kardinalita.
- E-R diagram (Entity-Relationship diagram)
  - Grafické znázornění, existuje mnoho variant
- Datový slovník
  - Tabulkové znázornění zahrnující detailní informace o atributech, včetně datového typu a jeho velikosti, volitelnosti (not null), a integritních omezení

Funkční analýza se zaměřuje na funkce systému, tedy jací jsou aktéři a jaké funkce se systémem provádí (podobně jako u use case diagramu).

Diagram datových toků je nástrojem pro modelování úložišť, procesů (funkce transformující vstupy na výstupy), a konkrétních datových toků v informačním systému (přesun informace z jedné části systému do druhé). Je organizován hierarchicky, kdy první úroveň definuje hlavní systémy a také propojení systémů s vnějším světem (např. evidence objednávek), další úrovně podrobněji popisují každý systém a podsystém (např. vytvoření/změna objednávky), až do nejspodnější úrovně ze které odvodíme *minispecifikace* což je popis logiky konkrétní funkce (jak transformuje vstupní datové toky na výstupní) přirozeným jazykem (nebo míchanicí přirozeného jazyka a pseudokódu).

Relační datový model, SQL; funkční závislosti, dekompozice a normální formy.

Relační datový model je druhou fází modelování databázového systému, kdy se konceptuální model převede do modelu vhodného pro relační databázové systémy — ty jsou založeny na tabulkové reprezentaci dat, kde tabulka reprezentuje entitní typ, sloupce jsou atributy, a řádky jsou entity (záznamy).

Primární klíč je zde podmnožina atributů která jednoznačně identifikuje entitu.

Vazba 1:1 je v relačním modelu definována jako atribut v jedné tabulce, odkazující na primární klíč ve druhé tabulce (*cizí klíč*).

Vazba 1:N a M:N jsou v relačním modelu reprezentovány další tabulkou obsahující *cizí klíče*, které odkazují na primární klíče záznamů v tabulkách mezi kterými je vazba definována.

SQL (Structured Query Language) je deklarativní jazyk pro provádění dotazů nad relačními databázovými systémy. Cílem jazyka je specifikovat **co** udělat, ne **jak** to udělat. Zahrnuje příkazy které můžeme rozdělit do 4 kategorií:

- Data Query Language (dotazování, založeno na relační algebře — SELECT)
- Data Manipulation Language (úprava dat — INSERT, UPDATE, DELETE)
- Data Definition Language (práce s relačními schématy — CREATE, ALTER, DROP)
- Data Control Language (práce s oprávněním a transakcemi — GRANT, REVOKE, BEGIN/-COMMIT/REVOKE TRANSACTION)

Relační algebra definuje operace nad celými relacemi/tabulkami:

- Projekce $\Pi_X(R)$ je výběr atributů X z relace R (`SELECT <X> FROM <R>`)
- Selekce $\sigma_X(R)$ je výběr řádků dle kritéria X (je možno použít logické spojky) z relace R (`SELECT * FROM <R> WHERE <X>`)
- Spojení $R \bowtie S$ je výběr kombinací záznamů dvou relací které se shodují na společných atributech (`SELECT * FROM <R> NATURAL JOIN <S>`)
  - Další variace spojení kde se narozdíl od natural joinu musí deklarovat konkrétní atributy na kterých chceme spojovat, a forma joinu (inner join, full outer join, left outer join, right outer join)
- Agregace ${}_AG_{F(B)}(R)$ provede seskupení záznamů relace R podle atributu A, a pro každou skupinu je aplikována funkce F na atribut B (např. ${}_{country}G_{AVG(age)}(People)$ pro průměrný věk každé země; `SELECT *, <F(B)> FROM <R> GROUP BY <A>`).

Funkční závislost je závislost mezi dvěma množinami atributů $X \to Y$ pokud platí, že mají-li dva prvky **každé možné** relace stejné hodnoty atributů X, pak mají stejné hodnoty atributů Y.

- Poznámka o „každé možné“ relaci jen říká, že pro konkrétní relaci může vypadat že existuje funkční závislost (např. závislost *jméno* $\to$ *příjmení* pokud v relaci zrovna nejsou 2 osoby se stejným jménem), ale aby se jednalo o závislost tak by to muselo platit pro všechny možné relace s validními daty.

Uzávěr množiny atributů X značíme $X+$ a je to množina všech atributů které podle atributů X můžeme dohledat (patří tam i samotné atributy z X). Uzávěr určíme pomocí množiny všech funkčních závislostí.

Armstrongovy axiomy jsou pravidla odvozování funkčních závislostí:

- Dekompozice (převod na elementární závislosti — $X \to YZ \Rightarrow X \to Y; X \to Z$)
- Sjednocení (opak dekompozice — $X \to Y; X \to Z \Rightarrow X \to YZ$)
- Augmentace (pokud platí $X \to Y$, pak platí $XZ \to YZ$)
- Tranzitivita (pokud platí $X \to Y$ a $Y \to Z$, pak platí $X \to Z$)
- (a několik dalších)

Klíč je minimální množina atributů která jednoznačně identifikuje jinou množinu atributů.

Normální forma je řada pravidel, které když databáze splňuje, tak můžeme říct že je dobře navržena. Každá úroveň zahrnuje tu předchozí.

- 1. normální forma vyžaduje atomické (nedělitelné) hodnoty každého atributu. Pokud např. máme celou adresu v jednom atributu, nebo několik telefonních čísel v jednom atributu, musíme je rozdělit (adresu do více atributů, telefonní čísla do oddělené tabulky s 1:N vazbou).
- 2. normální forma vyžaduje, aby neklíčové atributy byly závislé na celém klíči a ne jen jeho části (pokud mám klíč na spojení dvou atributů ale můžu jednoznačně určit hodnotu atributu pouze s využitím jedné části klíče, pak musím tabulku rozdělit na dvě; jinak řečeno, pokud $AB \to XY$ pak nesmí platit $B \to X$).
- 3. normální forma vyžaduje, aby neexistovala tranzitivní závislost neklíčového atributu (pokud $A \to XY$, pak nesmí platit $X \to Y$).
- Boyce-Coddova normální forma vyžaduje, aby každá funkční závislost měla na levé straně klíč (tedy pokud $AB \to XY$, pak nesmí platit $X \to A$).
- Další normální formy asi nejsou až tolik důležité...

Dekompozice je proces rozkladu velké tabulky na několik menších, abychom splnili podmínky normálních forem. Můžeme začít s tabulkou obsahující všechny atributy, a postupně odebírat funkční závislosti a přesunovat je do nových tabulek. Příklad $\{A, B, C, D, E\}$ kde platí:

- $AB \to C$
- $C \to DE$
- $D \to E$

Provedení rozkladu s počáteční tabulkou $\{A, B, C, D, E\}$:

1. $C \to DE \Rightarrow \{A, B, C\}; \{C, D, E\}$
2. $D \to E \Rightarrow \{A, B, C\}; \{C, D\}; \{D, E\}$
3. $AB \to C \Rightarrow \{A, B, C\}; \{C, D\}; \{D, E\}$

Transakce, zotavení, log, ACID, operace COMMIT a ROLLBACK; problémy souběhu, řízení souběhu: zamykání, verzování, úrovně izolace v SQL.

Transakce je sekvence příkazů která převede databázi z jednoho konzistentního stavu do druhého konzistentního stavu. Laicky řečeno, buď jsou provedeny všechny příkazy najednou, nebo není proveden žádný.

Databáze s transakcemi musí splňovat vlastnosti kterým souhrnně říkáme *ACID*:

- Atomicity — Atomičnost (operace se provede buď celá nebo vůbec)
- Consistency — Konzistence (databáze musí být na konci transakce v konzistentním stavu — všechna integritní omezení musí platit, všechny triggery reagující na změny musí být spuštěny)
- Isolation — Izolovanost (dvě transakce probíhající najednou jsou od sebe izolovány — nesmí se navzájem ovlivňovat)
- Durability — Trvalost (po dokončení transakce je stav databáze trvale uložen)

Pro zahájení transakce použijeme příkaz `BEGIN TRANSACTION`, po kterém následuje sekvence příkazů (u některých databázových systémů tam můžeme zahrnout i např. try/catch bloky nebo podmínky). Pokud jakýkoliv příkaz selže, nebo pokud manuálně spustíme příkaz `ROLLBACK TRANSACTION`, databáze se navrátí do stavu ve kterém byla před transakcí. Použitím příkazu `COMMIT TRANSACTION` potvrdíme úspěšné dokončení transakce.

V některých databázových systémech můžeme použít i tzv. savepointy, které uloží stav databáze uvnitř transakce, a příkazem `ROLLBACK` se můžeme vrátit ke konkrétnímu savepointu a od něj pokračovat v transakci dál, místo toho abychom se vrátili na úplný začátek a transakci celou zrušili.

Zotavení je proces návratu do konzistentního stavu databáze po kritické chybě. Pokud systém spadne, je potřeba buď navrátit transakci která probíhala v době pádu (UNDO), nebo zapsat na disk transakci která byla dokončena ale pád nastal v době zápisu do permanentního úložiště (REDO). K obojímu se používá *log* (také nazývaný *žurnál*), typicky databáze umí provést oba způsoby zotavení najednou ale je jednodušší je vysvětlit odděleně:

Okamžitá aktualizace (UNDO/NO-REDO) provádí návrat transakce po pádu. Každá změna je okamžitě uložena na disk, ale ještě těsně před tím je do logu zapsán stav záznamu před změnou. Pokud nastane chyba, tak databázový systém po restartu projde log a obnoví původní stav všech záznamů.

Odložená aktualizace (NO-UNDO/REDO) nejdříve všechny změny zapíše do logu, a až poté začne zapisovat na disk. Pokud nastane při zápisu na disk chyba, tak databázový systém po restartu projde log a zopakuje zápis na disk.

Řízení souběhu je potřeba pokud k databázi přistupuje více uživatelů najednou, a tím pádem může najednou běžet několik transakcí. V takovém případě nastávají 3 typy problémů:

- *Ztráta aktualizace* nastane, když dvě transakce společně přečtou nějakou hodnotu, a poté ji společně přepíšou. Pokud by např. každá transakce zdvojnásobila hodnotu, pak by po průběhu dvou transakcí měla být hodnota zčtyřnásobena, ale pokud nastane tento problém tak jedno zdvojnásobení se „ztratí“.
- *Špinavé čtení* nastane, když jedna transakce zapíše hodnotu, druhá transakce ji přečte, a poté první transakce provede rollback. Druhá transakce pak pracuje s hodnotou která není validní.
- *Nekonzistentní analýza* nastane, když jedna transakce pracuje s daty které později změní a potvrdí druhá transakce.

Zámky jsou jedním z možných řešení problémů souběhu, kdy databázový systém zamkne část dat ke které transakce přistoupí.

- Pokud transakce chce číst data, požádá o *sdílený zámek*. V takovém případě může data číst více transakcí najednou, ale nikdo je nesmí měnit kvůli zachování konzistence.
- Pokud transakce chce měnit data, požádá o *exkluzivní zámek*. V takovém případě má k datům plný přístup daná transakce, a žádná jiná nemůže data měnit ani číst.
- Deadlock nastane, když dvě transakce každá čeká až ta druhá uvolní svůj zámek. Možná řešení jsou např. vyžádat si všechny zámky už na začátku transakce, nebo po určitém čase automaticky jednu transakci zrušit a spustit ji znova později.

Verzování je další způsob řešení problémů souběhu, kdy databázový systém vytváří kopie dat pro každou transakci, čímž zaručí že první transakce neuvidí změny které provedla druhá transakce dokud je druhá transakce nepotvrdí.

Úroveň izolace určuje jak jsou od sebe transakce izolovány, neboli kdy se změny jedné transakce projeví uvnitř druhé transakce. Každý databázový systém si je implementuje trochu (nebo úplně) jinak, ale obecně se jedná o balanc mezi výkonem a problémy které mohou nastat. Pokud využíváme zamykání, tak určuje přesné chování zámků — např. jaká část dat je zamknutá, a zda jsou zámky uvolněny okamžitě po skončení příkazu nebo až na konci transakce (pokud by např. byl sdílený zámek uvolněn okamžitě po provedení příkazu SELECT, tak by dva identické SELECTy mohly vracet různé výsledky pokud by mezi nimi jiná transakce přidala nový záznam, čemuž říkáme *výskyt fantoma*).

Procedurální rozšíření SQL, PL/SQL, T-SQL, triggery, funkce, procedury, kurzory, hromadné operace.

Procedurální rozšíření jsou rozšíření SQL jazyka o typické prvky procedurálních jazyků — procedury, funkce, podmínky, cykly, události, atd. Téměř každý databázový systém na to šel trochu jinak — PL/SQL byl vyvinut firmou Oracle pro jejich databázový systém Oracle, T-SQL byl vyvinut Microsoftem pro Microsoft SQL Server.

V PL/SQL je základní jednotkou *blok*, který zahrnuje volitelné deklarace lokálních proměnných (`DECLARE`), povinný počátek části s příkazy (`BEGIN`), volitelný blok pro záchyt výjimek (`EXCEPTION`) a povinné ukončení bloku (`END`). Takovýto blok je *anonymní procedura*, ale můžeme deklarovat i pojmenované procedury které pak můžeme volat z jiných procedur, a pojmenované funkce které oproti procedurám explicitně vrací hodnotu (existují i `OUT` parametry pomocí kterých umí vracet hodnotu i procedury, ale ty např. nelze použít uvnitř SELECT dotazu).

V T-SQL neexistuje přímo ohraničený blok pro deklarace, proměnné si můžu definovat kdekoliv. Blok kódu je ohraničen `BEGIN` a `END`, zachytávání výjimek lze provést pomocí try/catch bloků (`BEGIN TRY`, `END TRY BEGIN CATCH`, `END CATCH`). Stejně jako v PL/SQL existují i pojmenované procedury a funkce.

Trigger je procedura která je spuštěna při určité události, např. při vložení (`INSERT`), úpravě (`UPDATE`), nebo smazání (`DELETE`) záznamu. Můžeme je použít čistě jako reakci na událost (např. logování události, replikace dat), nebo pro zrušení události.

Podmínky:

- PL/SQL:

```
IF x THEN ...
ELSIF x THEN ...
ELSE ...
END IF;
```

- T-SQL:

```
IF x
BEGIN ... END
ELSE IF x
BEGIN ... END
ELSE
BEGIN ... END
```

Cykly:

- PL/SQL:

  (A) `LOOP ... END LOOP;`
  (B) `WHILE x LOOP ... END LOOP;`
  (C) `FOR x IN y LOOP ... END LOOP;`

- T-SQL:

```
WHILE x
BEGIN ... END
```

Kurzor je ukazatel na konkrétní záznam který navštívíme v průběhu `SELECT` dotazu:

- PL/SQL (A):

```
CURSOR c IS SELECT ... (uvnitř DECLARE)
OPEN c;
LOOP
  FETCH c INTO x;
  EXIT WHEN c%NOTFOUND;
  ...
END LOOP;
CLOSE c;
```

- PL/SQL (B):

```
CURSOR c IS SELECT ... (uvnitř DECLARE)
FOR x IN c
LOOP
  ...
END LOOP;
```

- T-SQL:

```
DECLARE c CURSOR FOR SELECT ... (uvnitř DECLARE)
OPEN c
FETCH NEXT FROM c INTO x
WHILE @@FETCH_STATUS = 0
BEGIN
  ...
  FETCH NEXT FROM c INTO x
END
CLOSE c
DEALLOCATE c
```

Hromadná operace je speciální příkaz který databázovému systému řekne, že bude prováděno velké množství operací jednoho typu. V PL/SQL si můžu například načíst velké množství záznamů do lokálních proměnných pomocí:

```
SELECT x, y BULK COLLECT INTO list_of_x, list_of_y
```

...poté s nimi něco provést, a hromadně je změnit pomocí:

```
FORALL index IN 1 .. list_of_x.COUNT [INSERT/UPDATE/DELETE]
```

T-SQL přesný ekvivalent těchto operací nemá.

Dynamické SQL je potřeba pokud neznáme celou formu příkazu který chceme volat v proceduře, nebo používáme příkaz který jinak není podporován (např. `GRANT` nebo `CREATE`). V takovém případě uložíme příkaz do textového řetězce a spustíme příkazem `EXECUTE IMMEDIATE` v PL/SQL nebo `sp_executesql` v T-SQL.

Musíme dávat pozor na *SQL injection* pokud zpracováváme uživatelská data a vytváříme z nich dynamické SQL příkazy — pokud může útočník kontrolovat obsah textového řetězce který bude spuštěn jako SQL příkaz, může způsobit velkou škodu. Pro ochranu můžeme použít *vázané proměnné* které budou poté nahrazeny opravdovou hodnotou, ale nelze to tak udělat všude (např. nelze tak nahradit jméno tabulky v `CREATE TABLE x`).

(Základní fyzická implementace databázových systémů: tabulky a indexy; plán vykonávání dotazů.)

OTÁZKA Z HLAVNÍHO PŘEDMĚTU, ZAHRNUJE 2 NÁSLEDUJÍCÍ OTÁZKY.

Fyzická implementace databázových systémů: tabulky (halda, shlukovaná tabulka, hashovaná tabulka) a indexy (B-strom, bitmapový index), materializované pohledy, rozdělení dat, stránkování, řádkové a sloupcové uložení dat.

Fyzická implementace definuje datové struktury pro objekty databázového systému, a řeší uložení dat.

Stránka je nejmenší blok dat se kterým databázový systém pracuje. Většinou je uložena na disku (tím pádem je velikost stránky nějakým násobkem alokační jednotky souborového systému), ale protože je přístup k disku mnohem pomalejší než přístup k paměti RAM, používá se *cache buffer* který uchovává některé stránky v hlavní paměti.

Implementace tabulek:

- *Halda* je základní typ tabulky ve které nejsou záznamy nijak uspořádány (připomíná pole které lze na konci rozšiřovat).

  Smazání záznamu pouze označí záznam jako smazaný — byl by velice špatný nápad přesouvat všechny záznamy o jedno místo doleva při každém smazání. Vložení záznamu pak vyhledá první smazaný záznam který nahradí, nebo pokud žádný neexistuje tak je záznam vložen na konec pole (pokud je potřeba, je vytvořena nová stránka).

  Každý záznam je jednoznačně identifikovatelný pomocí *ROWID*, což umožňuje využít indexy odkazující na konkrétní záznamy. Procházení bez indexu probíhá sekvenčně, což je velmi rychlé v případech kdy chci aby mi dotaz vrátil nebo zpracoval větší část dat (na přesném % záleží, ale okolo 75 % by měl být sekvenční průchod rychlejší než procházení přes index).

- *Shlukovaná/clusterovaná tabulka* je databázovým systémem vytvořena v případě kdy je primární klíč tabulky clusterovaný. Je založena na stromové struktuře (typicky B+ strom). Záznamy jsou seřazeny podle tohoto klíče, a uloženy přímo ve stromové struktuře (narozdíl od klasického indexu který obsahuje jen odkaz na záznam — ROWID u haldy, primární klíč u clusterované tabulky).

  Vyhledávání podle klíče (i např. rozmezí hodnot klíče) je velice rychlé, za předpokladu že výstup dotazu je poměrně malou částí celé tabulky (sourozenci ve stromu mohou být fyzicky uloženy na různých místech, což způsobuje náhodný přístup k disku).

  Vkládání je o něco pomalejší kvůli nutnosti zatřizování záznamu na správné místo — pro vložení je potřeba projít strom od kořene k listu do kterého je záznam vložen, a pokud by vložení překročilo maximální velikost listu tak je uzel rozdělen (a může to způsobit i kaskádové rozdělení kořenů).

- *Hash tabulka* ukládá záznamy se stejnou hodnotou hashe blízko u sebe. Typicky jsou hash tabulky vytvářeny interně v databázovém systému, např. pro účely indexace nebo v MSSQL je hash tabulka automaticky vytvořena pro dočasné tabulky. Vyhledání záznamu může být rychlejší než u tabulky založené na stromové struktuře, na druhou stranu to vyžaduje minimum kolizí a tím pádem plýtvání místa.

Implementace stromových struktur:

- *B-strom* je strom který se podobá binárnímu stromu, ale každý uzel obsahuje proměnný počet potomků (strom řádu n má limit n prvků v jednom potomkovi; po rozdělení má každá strana $\frac{n}{2}$ potomků, díky čemuž je vyvážený).
- *B+ strom* je modifikací B-stromu, kde jsou data obsaženy pouze v listových uzlech, a nelistové uzly obsahují pouze klíče které vedou do potomků. Listové uzly také mohou mít referenci na dalšího potomka pro zrychlení sekvenčního přístupu.

Dělení indexů:

- Podle množství atributů:
  - Jednoduchý index (obsahuje pouze 1 atribut)
  - Složený index (obsahuje 2 nebo více atributů; je implementován jako B+ strom kde seřazení uzlů závisí na pořadí atributů ve složeném indexu, což znamená že index `(a,b)` lze použít pouze pokud hledáme podle `(a)` nebo `(a,b)`, ne pokud hledáme pouze podle `(b)`)
- Podle implementace:
  - *B+ strom* (snad jasné)
  - *Bitmapový index* je vhodný pro kategoriální atributy. Pro každou hodnotu atributu je vytvořen řetězec bitů. Počet bitů je roven počtu záznamů, a 1 označuje záznamy pro které je atribut roven požadované hodnotě. Pomocí bitových operací pak lze např. zkombinovat dotazy na dané atributy, a z pozic jedniček poté získat všechny záznamy.
  - *Prostorový index* se používá pro prostorová data (implementace např. quadtree strukturou).
  - *Full-text index* se používá pro indexaci textových dat pro urychlení vyhledávání částí textu.

Dotazy které referencují pouze atributy které jsou pokryty klíčem indexu nemusí vůbec vstupovat do hlavní datové struktury a celý dotaz lze provést pouze v rámci struktury indexu — u takových indexů říkáme že *pokrývají* daný dotaz.

Indexy jsou obvykle vytvářeny automaticky pro klíče (primární, unikátní, cizí). Nevýhodou je, že čím více máme indexů, tím více jich pak musíme aktualizovat při změnách v tabulce, což zpomaluje CUD operace.

Pohled (View) je databázový objekt který se tváří jako tabulka, ale neobsahuje žádná data — pouze říká jakým způsobem má databázový systém data získat a zpřístupnit je ve formě tabulky.

Zhmotněný pohled (Materialized view) je reálná tabulka která obsahuje výsledky `SELECT` operace (slouží jako cache u komplikovaných SELECTů). Některé databázové systémy aktualizují tyto výsledky automaticky při změně nějaké hodnoty v odkazovaných tabulkách a atributech, některé je aktualizují periodicky takže výsledky nemusí být vždy aktuální.

Řádkové uložení dat je typický způsob uložení dat tabulek, kdy jsou ukládány jednotlivé řádky za sebou. V heap tabulce je navíc každý řádek jednoznačně identifikovatelný pomocí ROWID.

Sloupcové uložení dat namísto po sobě jdoucích řádků ukládá po sobě jdoucí hodnoty jednoho sloupce, tj. každý sloupec je uložen odděleně. Výhodou je lepší komprese, protože se nemíchají různé typy dat. V některých databázových systémech můžeme pro tabulku zvolit sloupcové uložení dat (např. v MSSQL se tomuto říká *COLUMNSTORE*).

Plán vykonávání dotazů, logické a fyzické operace, náhodné a sekvenční operace, ladění vykonávání dotazů, algoritmy spojení.

Vykonávání dotazu je provedeno v několika fázích:

1. Parsování dotazu (převod do interní reprezentace a zpracování pohledů/views)
2. Výběr logického plánu (přepis dotazu na sémanticky ekvivalentní dotaz který poběží rychleji; např. změnou pořadí některých operací)
3. Výběr fyzického plánu (výběr algoritmů které provedou logický plán, rozhoduje se podle typu tabulky, existujících indexů, atd.; většinou vygeneruje několik plánů které ocení podle odhadovaného využití procesoru a disku, a vybere ten s nejnižší cenou)

Plán vykonávání dotazu je tedy sekvence kroků které databázový systém provede v rámci konkrétního dotazu.

Query optimizer je komponenta databázového systému která provádí výběr plánu.

Plan cache je cache obsahující nedávno vygenerované plány, což umožňuje přeskočit fázi generování plánů pokud je proveden stejný dotaz vícekrát.

Logické operace jsou operace relační algebry.

- Selekce
- Projekce
- Join
- Sort
- a další...

Fyzické operace implementují logické operace (obecně vše co je potřeba pro provedení CRUD operací a procedurálních rozšíření, tedy i např. cykly, volání funkcí, vkládání a mazání záznamů, aktualizace indexů, načtení a aktualizace vzdálených objektů v distribuovaných systémech).

Ladění vykonávání dotazů (ve smyslu zvýšení výkonu) můžeme provést mnoha způsoby — přidat nebo upgradovat hardware, přidat indexy, optimalizovat konkrétní dotazy, použít cache (např. materializované pohledy), snížit přenos dat a počet databázových operací v aplikační vrstvě (neoptimalizované ORM), atd.

ZBYTEK V DALŠÍ OTÁZCE...

Operátory plánu vykonávání dotazů; statistiky hodnot v databázových systémech; cenová optimalizace.

Operátory pro přístup k datům dělíme podle typu struktury. Při výpočtu ceny u nich sledujeme počet logických přístupů (kolikrát bude k objektu přistoupeno poté co je načten do paměti RAM) a počet fyzických přístupů (kolik bude přečteno stránek z disku).

- Sekvenční průchod (prochází celou datovou strukturu jakéhokoliv typu, a hledá požadované záznamy)
  - Table Scan (halda)
  - Index Scan (clusterovaná tabulka nebo index)
- Přímý přístup (prochází stromovou strukturu od kořene do listu s požadovanými daty)
  - Index Seek (clusterovaná tabulka nebo index; buď pro 1 hodnotu nebo pro rozmezí hodnot)

Operátory agregace provádí sumarizaci dat (`GROUP BY` s využitím např. `SUM`, `COUNT`, `MAX`).

- Stream Aggregate (vyžaduje setřízená data podle seskupovacího klíče, pak může postupně agregovat do jedné hodnoty v paměti)
- Hash Aggregate (vytváří v paměti hash tabulku která mapuje seskupovací klíče na agregované hodnoty)

Operátory spojení provádí joiny.

- Nested Loop Join
  - Pro každý záznam vnější tabulky hledá záznam vnitřní tabulky.
  - Kvůli vnořenému cyklu je algoritmus vhodný jen pokud je vnější tabulka malá, a vnitřní tabulka má index na spojovacím klíči.
  - Varianta *Indexed NLJ* pro vnitřní tabulku používá Index Seek, pokud je to možné.
  - Je jediným spojovacím operátorem který umí provést spojení s nerovností.
- Merge Join
  - Sekvenčně prochází obě tabulky posouváním dvou kurzorů.
  - Vyžaduje aby obě tabulky byly setřízeny podle spojovacího klíče (setřízení je možno provést i v paměti pokud to je pro optimalizátor výhodné).
  - Na začátku je kurzor umístěn na první pozici v obou tabulkách, poté se vždy posune kurzor jehož hodnota je v pořadí „nižší“. Když se hodnoty pod kurzory shodují, je spojení záznamů přidáno k výsledkům.
  - Pokud se v tabulkách vyskytují duplicitní hodnoty (*many-to-many merge join*), je vytvořena pomocná tabulka *worktable*. Pravý kurzor se začne posunovat a uloží všechny duplicity do *worktable*. Poté se začne posunovat levý kurzor, a pro každou shodu je do výsledků vložen řádek levé tabulky spojen se všemi řádky ve *worktable*. V momentu kdy se levý kurzor přestane shodovat je *worktable* vyprázdněna.
- Hash Join
  - Vytvoří hash tabulku pro menší tabulku, poté prochází větší tabulku a hledá odpovídající záznamy v hash tabulce.

Statistiky jsou metadata tabulek a indexů, které optimalizátor používá pro odhad cen operátorů. Většinou jsou jen přibližné, protože by bylo příliš náročné aktualizovat je po každém dotazu a pro všechna data v tabulce, takže jsou aktualizovány průběžně a většinou vzorkováním částí dat. Uchovávají např.:

- Kardinalitu tabulky (počet záznamů), používá se např. u Hash Join který na začátku srovná velikosti tabulek
- Hustota sloupce (1 ÷ počet záznamů s odlišnými hodnotami), používá se např. u agregačních operátorů
- Histogram hodnot sloupce (počet řádků které spadají do daného intervalu hodnotu), používá se např. pro určení selektivity operátoru

Selektivita operátoru odhaduje poměr dat tabulky které operátor vrací (např. selektivita 0.1 říká že dotaz vrátí 10 % celé tabulky).

Cenová optimalizace přiřazuje každé operaci „cenu“ procesorového výpočtu a diskových přístupů v závislosti na statistikách (a např. na odhadu mohutnosti dat které jsou vstupem a výstupem sousedních operátorů).

Při generování plánů často není možné vygenerovat všechny možné plány, protože by jejich generování zabralo více času než třeba provedení dotazu nejhorším plánem, takže databázový systém většinou vygeneruje několik *kandidátů* a z nich vybere ten s nejnižší celkovou cenou všech operátorů.

Fyzická implementace datových struktur a algoritmů vykonávání dotazů, optimalizace přístupu do hlavní paměti a k disku, návrh a implementace cache buffer.

Datové struktury, algoritmy, stránky viz. předchozí otázky.

Cache buffer je lze implementovat pomocí algoritmu LRU (Least Recently Used). Jedná se o frontu stránek, kde při přístupu ke stránce je stránka vložena nebo přesunuta na začátek fronty, a v případě vložení je nejstarší stránka z fronty odebrána. Tento přístup je založen na předpokladu, že když se přistoupí ke stránce, je pravděpodobné že se k ní v blízké době přistoupí znova.

V cache bufferu mohou být i nedávno aktualizovaná data která ještě nebyla zapsána na disk, a v takovém případě se odebráním stránky z fronty provede zápis na disk. V případě že databázový systém spadne před zápisem na disk je provedeno zotavení pomocí logu/žurnálu.

Objektově-relační datový model a XML datový model: principy, dotazovací jazyky.

Objektově orientovaný databázový systém kombinuje vlastnosti relačních a objektových databázových systémů.

- V relačním databázovém systému jsou data reprezentována tabulkami, kde sloupce jsou atributy a řádky jsou záznamy (entity).
- V objektovém databázovém systému jsou data reprezentována objekty, které stejně jako v OOP vychází ze tříd (`CREATE TYPE x`) s atributy a metodami, a možností kompozice, dědičnosti, atd. což umožňuje reprezentovat velmi komplexní vztahy formou, která je pro člověka možná přirozenější než čisté tabulky. Některé rysy najdeme dnes i v klasických relačních systémech (např. procedury a funkce které přenáší část logiky na stranu databázového systému namísto aplikace).

- V objektově-relačním datovém modelu jsou vlastnosti obou přístupů zkombinovány. Pro reprezentaci se používají tabulky, které mohou mít atributy uživatelsky definovaných typů, což mohou být komplexní objekty stejně jako u čistě objektového databázového systému.

Tabulky objektově orientovaného databázového systému dělíme na *objektové* a *relační*. Relační tabulky mohou obsahovat jak klasické atributy které najdeme v relačních databázových systémech, tak objektové atributy. Objektové tabulky obsahují pouze objekty; výhodou je, že pro objekty v objektových tabulkách je automaticky generován identifikátor, který lze použít jako referenci uloženou ve více tabulkách najednou:

```
CREATE TABLE x (instance REF trida)

SELECT DEREF(instance) FROM x
```

Kolekce jsou datové typy které v sobě obsahují jiné datové typy. V relačních databázových systémech jsou typicky pouze lokální tabulky s jednoduchými atributy, zatímco u objektových databází mohou kolekce obsahovat objekty i další kolekce.

- Pole (kolekce s pevnou délkou, respektive limitovanou kapacitou)
- Asociativní pole (mapování klíčů na hodnoty)
- Vnořená tabulka (celá tabulka která může obsahovat další objekty a tabulky a tvořit komplexní struktury)

XML (Extensible Markup Language) je standardizovaný značkovací jazyk. XML dokument se skládá z hierarchie *tagů*, které mohou mít atributy s hodnotami a mohou mít také textový obsah. Pracujeme-li s XML dokumentem jako databází, budeme chtít definovat jak má struktura dokumentu vypadat.

XML schéma je formální popis struktury XML dokumentu — z čeho se skládá každý element, jaké jsou validní hodnoty atributů nebo textového obsahu každého elementu, kde v hierarchii se elementy nachází, atd.

XPath je dotazovací jazyk který vrátí část XML dokumentu. Dotaz má tvar cesty ve stromové struktuře dokumentu, pomocí které lze vybrat jeden konkrétní element, nebo celý strom elementů:

- `/aa` vybere elementy `aa` které jsou přímí potomci aktuálního kontextu (lze pomocí něj vybrat kořen dokumentu)
- `//aa` vybere element který je přímým nebo nepřímým potomkem aktuálního kontextu
- `[/aa]` vybere element který má přímého potomka `aa`
- `[@attr=5]` vybere element který má atribut `attr` s hodnotou 5

XQuery je jazyk pro dotazování a transformaci dat v XML dokumentu. Používá se v nativních XML databázových systémech.

- `let $doc := doc(x)` načte dokument s názvem `x` do proměnné `$doc`
- Nad XML proměnnou lze provést XPath dotaz (např. `$doc/a/b/c`, `$doc//a/b[attr]`)
- V některých ohledech je podobný SQL, např.:

```
for $x in doc(...)//a/b

where $x/aaa > 1

order by $x/bbb

return $x/ccc
```

Datová vrstva informačního systému; existující API, rámce a implementace, bezpečnost; objektově - relační mapování.

Datová vrstva je jednou z krajních vrstev třívrstvého informačního systému:

- Prezentační vrstva (uživatelské rozhraní a interakce)
- Aplikační/business vrstva (logika a funkce aplikace)
- Datová vrstva (přístup k datům které jsou uloženy v databázi, souborovém systému, apod.)

Pro přístup k databázi můžeme využít několika přístupů:

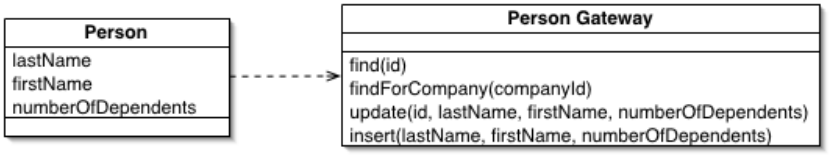
- Využít API pro konkrétní databázi a provádět přes něj dotazy (např. MySQLi v PHP)
- Využít API které umí pracovat s více typy databází a slouží jako vrstva abstrakce mezi programem a konkrétní databází (např. PDO v PHP, ODBC v Javě, ADO.NET v .NETu)
- Využít nějaký ORM framework (objektově-relační mapování) který zpřístupní entity v databázi jako objekty v objektově orientovaném jazyce (např. Hibernate v Javě, Entity Framework v .NETu)

Pokud přistupujeme k databázi přímo a vykonáváme na ní dotazy, musíme dávat velký pozor na bezpečnost - SQL injection (použít parametrizované dotazy), nastavení práv databázových uživatelů (pokud se do databáze někdo dostane tak ať nenapáchá velkou škodu).

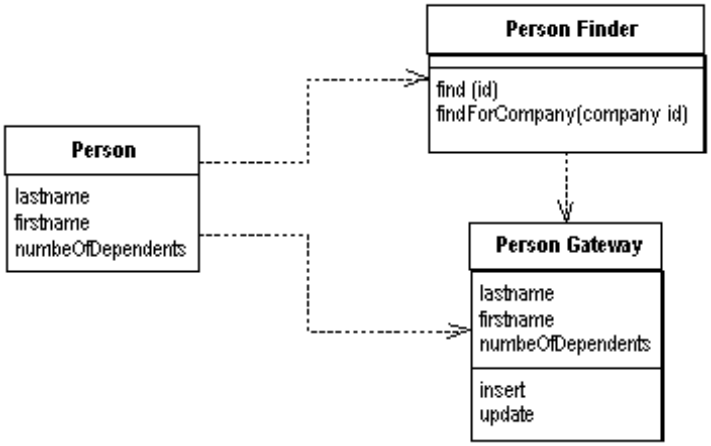
Objektově-relační mapování (ORM) se mohou zaměřovat jak na jeden konkrétní databázový systém, tak využít vrstvu abstrakce a fungovat s jakýmkoliv systémem (abstrakce a generalizace jsou flexibilní ale na druhou stranu zhoršují výkon a nemusí dosáhnout maximálního potenciálu konkrétního databázového systému).

Vzory:

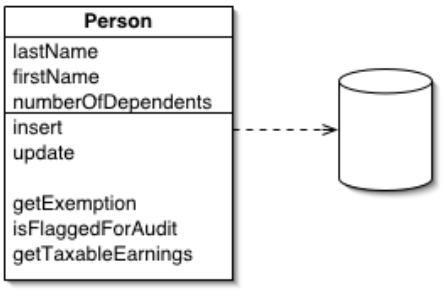
- Table Data Gateway (gateway objekt je brána k celé tabulce; obsahuje metody pro CUD operace jejichž parametry jsou samotné atributy tabulky, a metody pro R operace jejichž výstupem jsou seznamy řádků)



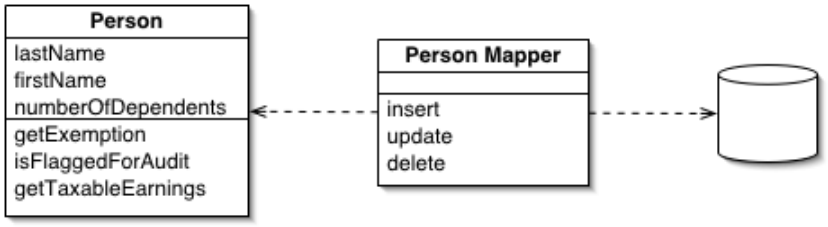
- Row Data Gateway (gateway objekt je brána k jednomu řádku tabulky která ale neobsahuje doménovou logiku, obsahuje data a CUD operace; dále existují někde jinde funkce pro R operaci které vrací gateway objekty)



- Active Record (podobný Row Data Gateway, ale neoddělují se data a logika)



- Data Mapper (mapper objekt převádí řádky na objekty a naopak, bez toho aby samotné objekty byly závislé na konkrétním relačním modelu)



Distribuované SŘBD, fragmentace a replikace.

Distribuovaný databázový systém je databázový systém kde jsou data rozprostřena ve více fyzických lokacích.

Heterogenní DSŘBD může používat různá databázová schémata a software, což velice situaci komplikuje a musí existovat nějaká vrstva translace. Typicky se s tímto nesetkáme, ale existují exotická řešení jako Informix Flexible Grid.

Homogenní DSŘBD je přirozeně opakem heterogenního. Cílem dobrého distribuovaného systému je tento fakt skrýt; navenek (pro software pracující s databázovým systémem) by se měl chovat stejně jako nedistribuovaný.

Replikace je jedním ze dvou přístupů tvorby distribuovaného databázového systému. Při replikaci jsou všechna data uložena ve více lokacích (*uzlech*), což umožňuje paralelní provádění dotazů a zabrání nedostupnosti systému v momentu kdy jeden uzel zrovna nefunguje (pád, údržba). Mezi nevýhody patří mj. že každá změna musí být propagována do všech uzlů, a využití více místa. Pokud je jeden

uzel nedostupný, nastává problém který řeší *primární kopie*:

- Jedna z kopií konkrétního objektu je *primární*, všechny ostatní jsou *sekundární* (pokud by primární kopie všech objektů byly na jednom uzlu, existoval by jediný bod selhání).
- Pro úspěšnou aktualizaci stačí, aby byla aktualizována primární kopie. Uzel poté zodpovídá na šíření primární kopie všem uzlům se sekundární kopií.

Fragmentace je druhým přístupem. Rozděluje data na *fragmenty*, které jsou uloženy v lokaci (uzlu) kde budou fragmenty využívány nejčastěji. Dělíme na dva způsoby:

- *Horizontální fragmentace* rozděluje množinu celých řádků. Pokud se řádek nachází v lokálním uzlu, pak k němu může přistupovat okamžitě; v opačném případě musí komunikovat s uzlem na kterém se řádek nachází.
- *Vertikální fragmentace* rozděluje množinu sloupců. Musí se dávat pozor na to, aby každý řádek ve vertikálním fragmentu bylo možno jednoznačně identifikovat (k tomu se používá *tuple ID*, což je klíč který jednoznačně identifikuje řádek a je ukryt jako sloupec každého vertikálního fragmentu).

Hybridní přístup kombinuje fragmentaci a replikaci, kdy je databáze fragmentovaná a některé fragmenty jsou dále replikovány do všech uzlů (např. replikuji jen ty nejdůležitější fragmenty).

Katalog obsahuje obecně metadata o databázovém systému (tabulky, uživatelé, indexy, atd.) a v distribuovaných systémech také informace o uzlech, fragmentech, apod. Katalog může být uložen centralizovaně, replikován na všech uzlech, nebo fragmentován (každý uzel si spravuje svůj katalog, spojením všech katalogů vzniká kompletní katalog).

Transakce (tam kde je potřeba komunikace s více uzly) musí být koordinovány tak, aby byly potvrzeny/navráceny v každém uzlu najednou. Koordinátor (typicky uzel na kterém byla transakce spuštěna) provádí transakci ve dvou fázích:

- *Fáze přípravy* rozesílá všem uzlům příkaz pro provedení transakce. Poté sesbírá odpovědi všech uzlů zda se transakce dokončila úspěšně.
- *Fáze potvrzení* zkontroluje odpovědi všech transakcí. Pokud všechny uspěly, pak odešle uzlům informaci že můžou lokálně provést COMMIT; pokud jeden nebo více uzlů neuspělo, pak všem odešle příkaz ROLLBACK.