Státnice

Složitost a vyčíslitelnost (10/10)

Metody tvorby algoritmů: rozděl a panuj, dynamické programování, hladový algoritmus. - ADS, PG I & II

Rozděl a panuj

- **Dekompozice shora dolů**
- **Problém rozdělíme na podproblémy menšího rozsahu, po jejichž vyřešení jsme schopni triválně vyřešit mateřský problém**
- **Pokud jsou podproblémy stejného typu, můžeme přirozeně realizovat rekurzí**
- **Aby byla metoda použitelná, měly by dílčí úlohy být převážně nezávislé**
- **Příklady**
  - **Hanojské věže**
  - **Quicksort, Mergesort**
  - **Binární vyhledávání**
  - **Prakticky cokoliv**
- Master Theorem: T(n) = aT(n/b) + O(n^d), potom T(n) je:
  - O(n^d), pokud a < b^d
  - O(n^d*log n), pokud a = b^d
  - O(n^log a), pokud a > b^d
  - Vysvětlení: a udává, na kolik se nám úloha štěpí, b udává, kolikrát je rozštěpená úloha menší, než ta původní a d je složitost zpracování jedné části
  - Příklad na Mergesortu: a = 2 (dělíme na dva kusy), b = 2 (kusy jsou poloviční), d = 1 (slití je lineární) => O(n log n)
  - Příklad na Binárním vyhledávání: a = 1 (dělíme na dva kusy, ale jeden zahazujem), b = 2 (kusy jsou poloviční), d = 0 (posloupnost je setříděná, rozdělení je konstantní) => O(log n)

Dynamika

- **Komplementární postup k rozděl a panuj**
- **Zdola nahoru**
- **Nepotřebuje nezávislé subúlohy (v tom je kouzlo dynamiky)**
- **Stavy dynamického programování**
  - **Acyklický orientovaný graf**
- **Řešíme nejdříve triviální problémy na nejnižší úrovni, a pak se po levelech dostáváme nahoru pomocí spojování již vyřešeného**
  - **Optimální řešení lze zkonstruovat z optimálních řešení podproblémů**
- Postupy

    - Top-down - problém se rekurzivně dělí na podproblémy a když se nějaký podproblém vyřeší, hodnota se zapamatuje, aby se stejný podproblém nemusel řešit znova (chápu to jako že když třeba počítám rekurzivně fib, tak si "kešuju" ty fiby, abych je nepočítal víckrát a neměl z toho exponencielu)
    - Bottom-up
- **Příklady**
    - **Fibonacciho čísla**
    - Různé textové úlohy, např. palindromy
    - Batoh (s omezeným maximem hmotností a kapacity)
        - Naplňujeme všechny kapacity fiktivních batohů s kapacitou < cílová kapacita za použití prvních i předmětů
        - Iterujeme tedy přes inkluzi nového předmětu
        - Více viz aproximace...
    - Floyd-Warshall
        - Graf, mohou být záporné hrany, bez záporných cyklů!
        - Nejkratší vzdálenosti mezi všemi dvojicemi vrcholů
        - Máme matici Mi představující nejkratší cesty mezi vrcholy za použití prvních i vrcholů
        - Při iteraci vždy zkoušíme jestli má smysl použít cestu přes nový vrchol

Hladové algoritmy

- **Sežerou to nejlepší na co v daný okamžik přijdou (vyberou podle nějaké primitivní metriky)**
- **Lokální optimalizace**
    - **Greedy přístup (Hill climbing) nám poměrně spolehlivě najde lokální extrém, ale už tam zůstane**
    - **Dá se trošku vylepšit aby měl větší šanci najít skutečné extrémy**
        - **Iterace**
        - **Simulated annealing - s rostoucím časem hledání snižujeme pravděpodobnost, že algoritmus půjde prozkoumat i zhoršující výsledky**
- **Typicky malá složitost, ale typicky taky nefungujou**
- **Příklady**
    - **Minimální kostra (Kruskal) - přidávej postupně hrany, které nevytváří kružnici, od nejkratší**

Amortizovaná složitost. - ADS2, ZSV?, DS?

- Měření nejhorším případem může někdy dávat pesimistické výsledky
- Zkusme tedy místo složitosti jedné operace měřit složitost celé posloupnosti operací (soustředíme se tedy spíše na průměrný případ)
- Příklad: Binární čítač
    - Při inkrementaci n-bitového binárního čísla o 1 můžeme potřebovat převrátit až O(n) bitů

    - Zkusíme-li amortizovat n po sobě jdoucích přičtení jedničky (začínáme-li od nuly), vyjde nám složitost O(1)
    - Důkaz agregační metodou:
        - V průběhu n přičtení se poslední bit překlopí n-krát, předposlední n/2-krát, obecně tedy i-tý bit od konce n/2^i-krát
        - V součtu tedy n + n/2 + n/4 + … = 2n operací
        - 2n / n = O(1)
    - Důkaz účetní metodou:
        - Řekněme, že každá změna bitové hodnoty stojí korunu
        - Za každé přičtení jedničky budeme vybírat 2 koruny - nyní jsou dvě možnosti:
            - Poslední cifra je 0, pak spotřebujeme jednu korunu na přetočení cifry na jedničku a druhou korunu uložíme k té jedničce do zásoby
            - Poslední cifra je 1, pak ji přetočíme na 0 a zaplatíme to z té koruny, kterou jsme tam uložili do zásoby, stále tedy máme dvě koruny; musíme však přenést bit do vyššího řádu, tak opakujeme stejnou akci s předposlední cifrou
        - Protože za každou akci vybíráme 2 koruny, při n operacích zaplatíme 2n / n => O(1)
- Příklad 2: Implementace C# List.Add:
    - Pokud je alokované pole moc malé, alokujeme dvojnásobnou velikost - to dává nejhorší případ O(n) počtu kopírování položek
    - Pokud začneme od prázdného pole a budeme postupně přidávat n prvků, vyjde amortizovaná složitost O(1)
        - Při každém vložení vyberu 3 koruny, na vložení, na překopírování právě vloženého prvku při příštím vložení a na příští překopírování odpovídajícího prvku v levé polovině pole.
    - Dá se dělat i potenciálovou metodou
        - Potenciálová metoda je velmi podobná jako účetní, rozdíl je v tom že místo abychom sledovali přínosy a ceny operací a jejich částí sledujeme “potenciální energii” celkového stavu
        - Potenciálová funkce musí být nenulová, a musí platit:
            - $T_{AMORT} = T_{ACTUAL} + (\phi_{AFTER} - \phi_{BEFORE}) * C$
        - V případě pole tedy potenciál ϕ = 2n - N, kde n je počet prvků v poli a N jeho velikost
        - Další použití: FibHeap, splay trees (what the FUCK is a splay tree??)...

Úplné problémy pro třídu NP, Cook-Levinova věta. - ZSV

Formální definice problémů

- Rozlišujeme problémy a úlohy
- Rozhodovací problém má výstup ano/ne, vstupu říkáme instance, můžeme definovat jako test na náležení do jazyka L obsahujícího právě ty instance, které splňují požadovanou vlastnost

- Úloha hledá pro danou instanci nějaký konkrétní výstup, vstup je instance úlohy, formálně binární relace mezi vstupem a výstupy
    - U optimalizační úlohy hledáme výstup, který je maximální nebo minimální mezi všemi možnými výstupy pro daný vstup
- Vstupy a výstupy kódujeme do abecedy {0, 1}* (to můžem, převody mezi soustavama jsou polynomiální - kromě unární soustavy)

Třídy jazyků

- Definujeme vždy pro nějakou přirozenou funkci f, jazyk L představující binární kódování problému
- Jazyk L patří do třídy DTIME(f(n)), pokud existuje TS M tž. L = L(M) a výpočet pro slovo délky n na něm končí do O(f(n)) kroků
- Jazyk L patří do třídy DSPACE(f(n)), pokud existuje TS M tž. L = L(M) a výpočet pro slovo délky n na něm zabere na pásce max O(f(n)) buněk
- Relace R patří do třídy DTIMEF(f(n)), pokud existuje TS M, který přijme slovo x právě tehdy když existuje y tž. R(x, y), a páska obsahuje slovo y
- DTIME(f) je vždy podmnožina DSPACE(f) - nestihnem popsat víc buněk než máme času
- P, PSPACE a PF definujeme pomocí výše zmíněných definic a polynomů, tedy např.
    - P = sjednocení všech DTIME(n^i)
- NP varianty jde intuitivně chápat dvěma způsoby - Jednak můžeme říct, že když dostaneme řešení NP úlohy, dokážeme ho v polynomiálním čase ověřit. Druhá interpretace jde přes nedeterministický TS, který takový problém dává v polynomiálním čase
    - Formálně je také potřeba, aby velikost výstupu šla omezit polynomem závislým na vstupu
    - L je z NP ⇔ existuje jazyk B z P a polynom p, že platí: x z L právě když existuje y s polynomiální velikostí vůči x tž. (x, y) z B
    - NP = sjednocení NTIME(n^i)
        - L patří do NTIME(f(n)), pokud existuje nedeterministický TS M, který pracuje v čase O(f(n)) a L = L(M).

Savičova věta říká mimo jiné že NSPACE(f(n)) je podmnožinou DSPACE(f(n)^2)

NP-Úplné problémy

- A polynomiálně převoditelný na B ($A \leq^p_m B$) pokud existuje fce f (převádění, spočitatelná v poly čase) pro kterou platí x z A ⇔ f(x) z B
    - V případě převodů v P má smysl definovat převoditelnost v logaritmickém prostoru
- PP je reflexivní a transitivní, A <= B, B z P (NP) implikuje A z P (NP)
- Jazyk A je NP těžký pokud pro každý jazyk B z NP platí že B <= A. Pokud je zároveň z NP, je NP-úplný
- A <= B, pokud A je NPU a A,B z NP, pak i B je NPU (logicky musí být aspoň tak těžký)
- Problém existence certifikátů (CERT)

    - Obecný problém pro všechny NP úlohy
    - Instancí je zakódování DTS M, vstupní slovo x a řetězec 1^t (unární kódování konstanty omezující počet kroků)
    - Řešíme zda existuje y kratší než t pro který platí že M(x, y) přijme v t krocích)
    - CERT je NPU
    - Dk.?
    - CERT je ne uplně praktický na převody - je moc abstraktní
- Cook-Levin: Kachlíkování je NPU
    - Standardně SAT, můžeme ale brát tak že dokážeme KACHL a pak převedeme na SAT :)
    - KACHL: množina barev, čtvercová síť určité velikosti, kachlíky s obarvenejma hranama, kachlíky nejde otáčet
    - KACHL je očividně NP (dostanem řešení, není problém ověřit)
    - Na KACHL se dá převést libovolný jiný NP problém A
        - Budeme předpokládat že pro A existuje NTS M který splňuje
            - Jediný přijímací stav (různý od počátečního)
            - Z příjímacího stavu neexistuje definovaný přechod
            - Hlava na prvním symbolu vstupního slova x, značí začátek vymezeného prostoru délky p(|x|)
            - Hlava se při výpočtu nepohne nalevo od počátku (mimo vymezený prostor)
            - Přijímací konfigurace jednoznačná - prázdná páska a hlava nejvíc vlevo
        - Základní idea: kódujeme postup výpočtu NTS M do kachlíků
            - Horizontální barvy mezi dvěma řadami kachlíků reprezentují konfigurace
            - Horní okraj = počáteční konfig, dolní okraj = přijímající konfig, boční okraj = lambda
            - Barvy $B = \Sigma \cup Q \times \Sigma \cup \{q_l, q_r \mid q \in Q\}$
            - Velikost s = p(|x|)
            - Kachlíky kódují možné přechody mezi konfigy pomocí barev
                - Buď přímo symbol na odpovídajícím místě pásky (horizontální hrany)
                - Nebo stav + symbol (tam kde máme hlavu) na horizontálních hranách
                - Nebo posun hlavy (párujeme přes q_l a q_r na vertikálních hranách)
            - Kachlíky různých typů
                - Kopírující (tam kde není hlava)
                - Přechodové bez posunu
                - Přechodové s posunem (dvojice spojená přes q_L a q_R)
                - Finalizační - kopíruje koncový stav a pozici hlavy

        - Pro uplně formální důkaz je ještě potřeba dodělat ekvivalenci obou reprezentací, ale to snad nikdo chtít nebude
- <TODO? převody problémů>
- Kachl se převádí na SAT
    - Uděláme formule s s^2*|K| proměnnými $x_{i,j,k}$ které budou mít ten význam, že na pozici (i,j) umístíme k-tý kachlík
    - Zákl. typy
        - “Na i,j něco je”
        - “Na i,j nejsou různé věci”
        - “Na sousedních hranách jsou matchující kachle”
            - + pro každý okraj
- SAT na 3SAT
    - Každou klauzuli ve tvaru (A | B), tž. |A| + |B| > 3 přepíšeme na (A | x) & (B | ~x)
- 3SAT na Vrcholové pokrytí
    - Pro každou proměnnou zavedeme grafík obsahující negativní a pozitivní literál, spojíme hranou - to zaručí ohodnocení všech proměnných
    - Pro každou klauzuli přidáme úplný grafík, spojíme s odpovídajícími literály
    - k = 2m + n, kde m je počet klauzulí a n počet proměnných
- VP na Hamiltona (kružnice vedoucí přes všechny vrcholy)?
- SAT na 3D párování
- 3DM na Loupežníky - batoh

Pseudopolynomiální algoritmy, silná NP-úplnost.

- Má smysl rozlišovat mezi “těžkostí” i různých NPU
- Batoh - B velikost batohu, váhy předmětů menší než B, ceny předmětů, maximalizujeme cenu batohu, ozn. V := suma cen
    - Dynamika
    - Máme tabulku (n + 1) x (V + 1), v každém slotu [j, c] je souhrnná váha a množina předmětů celkové ceny přesně c s minimální velikostí za použití prvních j předmětů, pokud neexistuje tak prázdná množina a B + 1
    - Iterujeme přes předměty a ceny, vždy zvážíme zda nám předmět pomůže (tzn. jestli s ním danou cenu získáme s nižší velikostí)
- Batoh má složitost O(nV)
    - Což je sice polynomiální vůči V, ale V samotné může být exponenciální vůči vstupu (kvůli binárnímu kódování)
    - Velikost vstupu je $O(n \log_2(B + V))$
    - Kdyby existoval polynom p tž. (B + V) <= p(n), byl by batoh poly
- Nechť A je libovolný rozhodovací problém a I jeho instance. len(I) označíme délku zakódování instance pomocí binárního kódování, max(I) hodnotu největšího číselného parametru v instanci. Řekneme že A je číselný problém pokud pro každý polynom p exje instance I tž. max(I) > p(len(I))

    - Lidsky: v zadání se můžou vyskytovat čísla exponencielně velká vzhledem k délce vstupu (?)
- Algoritmus je pseudopoly, pokud můžeme jeho časovou složitost omezit polynomem nad max(I) a len(I)
    - Lidsky: pseudopoly algoritmus pracuje v polynomiálním čase k velikosti vstupu, anebo alespoň vzhledem k hodnotám na vstupu
        - Plyne z toho, že každý poly algoritmus je i pseudopoly?
    - Kdybychom našli pseudopoly algoritmus pro nečíselný NPU problém, bylo by P = NP
- Nechť p polynom a A z NP. A(p) označíme restrikci problému na rozumné instance (max(I) <= p(len(I))). A je silně NPU pokud i tato restrikce je NPU pro NĚJAKÝ polynom p, jinak slabě NPU
    - Čili silně NP-úplné problémy nemají pseudopoly alg., zatímco slabě NPU ano
    - Nečíselné problémy automaticky silně NPU
- TSP je příklad silně NP-úplného číselného problému
- Existence pseudopoly algoritmu často implikuje existenci dobré aproximace optimálního řešení pro velké (exponenciální) vstupy

Aproximační algoritmy a schémata. - ZSV, ADS2

- Intuitivně: Snažíme se pro NPU úlohy najít řešení které bude nějak “rozumně” horší než to optimální, ale poběží v poly čase
- Optimalizační úloha A je buď maximalizační nebo minimalizační, a má tři části
    - Množina instancí D_A
    - Pro každou instanci máme množinu přípustných řešení $S_A(I)$
    - Funkce $\mu_A(I, \sigma)$ která přípustnému řešení σ přiřazuje kladné racionální číslo (hodnotu řešení)
    - Maximálně nebo minimálně ohodnocené řešení nazveme optimální
        - Hodnota optimálního řešení pro instanci I: OPT(I)
- Aproximační algoritmus je algoritmus který pro každou instanci vrátí nějaké přípustné řešení; označíme ALG(I). Aproximační poměr takového algoritmu je ε pokud je pro každou instanci nalezené řešení nejvýše ε-krát horší než optimální:
    - OPT(I) <= ε * ALG(I) (pro maximalizančí úlohy)
    - ALG(I) <= ε * OPT(I) (minimalizační)
- Vybrané aproximace:
    - Bin Packing
        - Množina předmětů a jejich velikostí (racionalní 0-1), chceme rozdělit do disjunktních množin kde suma velikostí v množině nepřekročí 1
        - Aproximace hladová - pro každý předmět dycky najdeme první koš množinu kam se dá ještě nacpat
            - Poměr 2 - plyne z toho, že v řešení bude existovat maximálně jeden koš (množina) který je zaplněn míň jak polovičně (jinak by ho nebylo potřeba zakládat)
    - TSP

        - Jen pro konkrétní podmínky (Trojúhelníková nerovnost, úplný graf, eukleidovský prostor)
        - Aproximační poměr 2
        - Kombinace hladového algoritmu a obcházení minimální kostry grafu
    - Batoh
        - Založeno na tom, že omezíme celkovou cenu předmětů jejich vydělením a zaokrouhlením na základě aproximačního poměru
        - Aproximační poměr je tedy vstupem algoritmu -> aproximační schéma
        - Přeškálování cen se dělá pomocí
            - m index předmětu s maximální cenou
            - t = dolní celá část ($\log_2(\varepsilon$ * c[m] / n)) - 1
            - c' = c / 2^t (přeškálovaná cena)
        - Složitost O(1/ε n^3)
        - Aproximační poměr: 1 + ε
- Algoritmus ALG řešící A nazveme aproximačním schématem úlohy A pokud na vstupu přijímá instanci I a racionální číslo ε > 0 a vydá řešení odpovídající aproximačnímu poměru (1 + ε).
- Poly aproximační schéma je AS kde časová složitost ALGε je polynomiální v délce instance (len(I))
- Úplně poly aproximační schéma je pokud ALG pracuje v čase polynomiálním v len(I) a 1/ε
- Rozdíl mezi poly a úplně poly je že u PAS může být 1/ε i v exponentu, u UPAS ne
- Případ kdy můžeme pseudopoly zobecnit na schéma je docela častý
- Máme opt. úlohu A s nezáporným celočíselným ohodnocením a polynomem q, tž. každá instance I splňuje OPT(I) < q(len(I), max(I)), pak pokud exje UPAS pro A, exje i pseudopoly (není mi uplně jasný co tahle věta vlastně chce říct)
    - Důsledkem prý je že pokud A je silně NPU opt. úloha splňující předchozí předpoklady a P není NP, neexje UPAS.
- Pokud by existoval poly aproximační algoritmus pro TSP s konstantním aprox poměrem ε > 1, pak P = NP (skrze hamiltonovskou kružnici)

Algoritmicky vyčíslitelné funkce, jejich vlastnosti, ekvivalence jejich různých matematických definic. - ZSV

- Co je algoritmus
    - Intuitivně: posloupnost instrukcí (+ potřebujeme prostředek, kde je vykonáme)
    - Formálně: Turingův stroj ⇔ ČRF ⇔ program pro RAM ⇔ program v C / Pascalu…
- Church-Turing (-Kleene): Pro každý algoritmus v intuitivním smyslu (lol) existuje TS který ho implementuje.
- Algoritmicky vyčíslitelné fce jsou tedy fce pro něž dokážeme sestavit TS, C# program, RAM program atp. který je řeší, tj. pro definovaný vstup nám vydá správnou výstupní hodnotu (tipuju)

- Turingovsky vyčíslitelná funkce: TS M počítá (částečnou) funkci f, pokud pro x z N stroj M, který obdrží $(x)_B$ na pásce, skončí výpočet v konfiguraci se slovem $(y)_B$ na pásce ⇔ f(x) ↓= y. Pokud f(x) není definována, M neskončí, nebo vyplivne nesmysl
- TS je ekvivalentní s dalšíma variantama co do výpočetní síly, zejména se jedná o
    - k-páskový TS (převod kartézským součinem stavů a znaků)
    - Jednostranně omezená páska (převod "přehnutím" pásky)
    - Více hlav víc ví (ve skutečnosti neví, také ekvivalentní TS)
- Universální TS - schopen simulovat práci libovolného TS
    - Používáme binární kódování všeho, tj. abecedy, přechodové fce atp.
    - Pro přepis používáme oddělovače, které rovněž kódujeme binárně
    - Instrukce oddělujeme #, jednotlivé části popisující instrukci (znak na pásce, stav) oddělujeme |
    - Kódujeme i binární reprezentaci identifikátorů stavů a znaků
    - Fixní délka bloku, stačí 3 bity (0, 1, L, N, R, #, |, ;)
- Kód pro UTS před který přidáme 1 (kvůli leading zeros) se dá interpretovat jako číslo, které nám jednoznačně popisuje co se bude dít, tedy nějaký TS - Gödelovo číslo
    - Vstupem UTS je dvojice w; x, kde w je kód simulovaného TS a x vstup
    - UTS konstruujeme se 4ma páskama (IO, pracovní, pomocná s délkou bloku v unárním kódování, stavová)
- Jazyky a rozhodnutelnost
    - Jazyk je rekurzivně spočetný pokud existuje TS který ho přijímá, tj. který se zastaví a přijme právě na slovech z jazyka, slova mimo jazyk buď odmítne nebo nezastaví (částečně rozhodnutelný)
    - Jazyk je rekurzivní pokud je RS a TS se vždy zastaví, také označován jako rozhodnutelný (TS přijímá vs. TS rozhoduje)
    - POST - jazyk je rekurzivní když on i jeho doplněk je RS
        - DK => trivi
        - DK <= paralelní spuštění obou strojů (po instrukcích), čekáme kdo se zastaví
    - Jsou i neRS jazyky - RS je spočetně mnoho, všech jazyků nespočetně mnoho
    - L_{DIAG} - diagonální jazyk, tedy jazyk který přijímá právě slova, která nepřijímá jim odpovídající jazyk (přesněji slovu odpovídající TS)
        - Je tedy nerozhodnutelný a taky není vůbec RS
- Ekvivalence TS a ČRF
    - => sestrojíme TS pro odvozující operace
    - <= zkoušíme minimalizací hledat kód výpočtu stroje pomocí predikátů T_n, když najdeme, extrahujeme výsledek pomocí funkce U
        - $T_n(e,x_{1..n},y)$ je PRP který je splněn ⇔ w_y kóduje konvergující výpočet M_e nad vstupem $x_{1..n}$ a po skončení nechá na pásce nějaké binární číslo

Částečně rekurzivní funkce. - ZSV

- f je PRF, pokud existuje její odvození ze základních funkcí pomocí substituce a primitivní rekurze:
  - Konstantní nulová: $o(x) = 0$
  - Následník: $s(x) = x+1$
  - Projekce $I_n^j(x_{1..n}) = x_j$
  - Substituce $S_n^m(f, g_{1..m}) = f(g_1(x_{1..n}), g_2 \dots , g_m)$
    - čili f je funkce m proměnných a g jsou funkce n proměnných
    - Substituce je funkce n proměnných
    - Odpovídá volání funkce
  - Primitivní rekurze: $R_n(f, g) = h(x_{1..n})$ ve tvaru (i) nebo (ii)
    - (i) $f(x_{2..n})$, pro $x_1 = 0$
    - (ii) $g(x_1 - 1, h(x_1 - 1, x_{2..n}), x_{2..n})$, pro $x_1 > 0$
    - čili f je funkce n-1 proměnných a g je funkce n+1 proměnných
    - Je to funkce n >= 2 proměnných
    - Odpovídá for cyklu
- f je ČRF, pokud ji lze odvodit ze základních funkcí pomocí substituce, primitivní rekurze a minimalizace:
  - Minimalizace: $M_n(f) = \min\{y \mid f(x_{1..n}, y) \downarrow = 0$ a (for-all $z <= y)[f(x_{1..n}, z)\downarrow]\}$
    - Lidsky: nejmenší y, pro které je f definovaná a vrací 0; navíc musí být f definována pro všechna menší y (ale nevracet nulu)
    - čili f je funkce n+1 proměnných
    - Je to funkce n proměnných
- f je ORF, pokud je ČRF a zároveň je definovaná na všech vstupech
- PRF jsou podmnožinou ORF a ty jsou podmnožinou ČRF
  - dokonce ostře (Ackerman)
- Predikát/relace R = množina n-tic (z N^n), $x_{1..n}$ žije v R $\Leftrightarrow R(x_{1..n})$
  - Charakteristická funkce χ_R - vrací 1, pokud $x_{1..n}$ žije v R, jinak 0
  - Pro RSP nerozhodují hodnoty char. funkce, ale jestli je definovaná nebo ne
  - Rekurzivní predikát = jeho char. funkce je PRF nebo ORF
- Pozorování: podmínku v minimalizaci můžeme nahradit libovolnou ČRF použitím její char. fce.

Vlastnosti

- U ČRF nejsme schopni říct, jestli se na vstupu x zastaví.
- Konečný součet a součin: Mějme f(i,x) PRF, pak i následující funkce jsou PRF
  - Konečná suma f(i,x)
  - Konečný produkt f(i,x)
  - Důkaz: prostě to vyjádříme for cyklem
- Podmíněný příkaz: $g_{1..n}(x)$ PRF, $R_{1..n}(x)$ primitivně rekurzivní predikáty, z nichž vždy právě jeden platí, potom následující funkce je PRF: $f(x) = g_i(x) \Leftrightarrow R_i(x)$

    - Důkaz: funkci zapíšeme jako sumu součinů g_i × char funkce R_i
- Oboje platí i pro ORF a ČRF
- Omezená kvantifikace nad PRP je taky primitivní, obecná už ale ne
    - Důkaz: omezená kvantifikace se dá udělat for cyklem, na obecnou třeba while
- Primitivně rekurzivní predikáty uzavřeny na konjunkci, disjunkci a negaci
- Obecné pozorování dokazování vlastností: Je dobré na to jít přes charfunkce
- $\varphi_e^{(n)}$ je ČRF n proměnných počítaná strojem M_e
- Kleene: Pro každé přirozené e, n existuje PRF U(y) a primitivně rekurzivní predikát $T(e,x_{1..n},y)$: $\varphi_e^{(n)} = U(\mu y[T_n(e,x_{1..n},y)])$
    - $T_n(e,x_{1..n},y)$ je PRP který je splněn ⇔ w_y kóduje konvergující výpočet M_e nad vstupem $x_{1..n}$ a po skončení nechá na pásce nějaké binární číslo
    - Dalším důsledkem je že můžeme libovolný RSP zapsat jako nějaký PRP a existenční kvantifikátor
- Univerzální funkce $\varphi_z^{(n+1)}(e, x_{1..n}) = \varphi_e^{(n)}(x_{1..n})$
    - Existuje (z Klíneho): $\varphi_z^{(n+1)}(e, x_{1..n}) = U(\mu y[T_n(e,x_{1..n},y)])$
- s-m-n věta - ta je tak hnusná, že sem ani nejde zapsat .. ale bez tý se snad dá žít, ne?
    - s-m-n funkce je prostá PRF
    - počítá číslo funkce s n parametry, která vznikla tak, že jsme do nějaké jiné funkce s (m + n) parametry zafixovali prvních m parametrů
    - s-m-n má m + 1 parametrů - číslo původní fce a hodnoty fixovaných parametrů

Rekurzivní a rekurzivně spočetné množiny a jejich vlastnosti. - ZSV

- Množina (podmn. přirozených čísel) A je rekurzivní, je-li její charakteristická funkce ORF
- Množina A je rekurzivně spočetná, je-li definičním oborem nějaké ČRF
    - W_e = dom φ_e = $\{x \mid \varphi_e(x)\downarrow\}$
- Množina A odpovídá jazyku $L = \{w_i \mid i \text{ žije v } A\}$ (plyne z ekvivalence ČRF a TS)
    - **Všechno o rekurzivních / rekurzivně spočetných jazycích platí i pro množiny**
    - **Jazyk L je rekurzivně spočetný ⇔ exje TS M: L = L(M)**
    - **Jazyk L je rekurzivní ⇔ exje TS M, který se vždy zastaví: L = L(M)**
        - **Pozn.: L(M) je jazyk (množina slov) přijímaný TS M, je to taková množina binárních řetězců, na kterých se TS zastaví a skončí v přijímajícím stavu**
    - **Je-li L rekurzivní => doplněk L je také rekurzivní**
    - **Postova věta: L je rekurzivní ⇔ L i doplněk L jsou rekurzivně spočetné**
- Konečná aproximace: $\varphi_{e,s}$ je konečnou aproximací φ_e definovanou následovně
    - $\varphi_{e,s} = \varphi_e(x_{1..n})$ pokud exje y < s: $T_n(e, x_{1..n}, y)$, kde T je rekurzivní predikát použitý výše u Kleeneho věty
    - $\varphi_{e,s}\uparrow$ jinak
    - Můžeme použít pro konečnou aproximaci množin, kterou označíme analogicky $W_{e,s}$ - stojí za to poznamenat že taková množina je konečná
    - Lidsky: s je omezení (třeba na počet kroků), do kterých musí výpočet T_n skončit

Rekurzivní množiny a jejich vlastnosti

- Rekurzivní množiny jsou ty, u nichž jsme schopni efektivně zodpovědět otázku náležení. Jsme také schopni je efektivně a systematicky generovat (tzn. vypsat jejich prvky v rostoucím pořadí).
- Pro rekurzivní množiny platí, že jsou oborem hodnot nějaké rostoucí úsekové ČRF
    - Úseková fce - její definiční obor tvoří počáteční přirozených čísel, tj. pokud je definovaná pro nějaké x, je definovaná i pro všechna y menší než x (může být celé N)
    - Logicky z toho plyne, že pokud je rekurzivní množina nekonečná, musí být ta ČRF jíž je definičním oborem obecná
    - Taky z toho plyne ona vlastnost sekvenčního vypsání
- Chovají se hezky k běžným množinovým operacím:
    - Uzavřené na sjednocení, průnik a doplněk
    - Důkaz plyne z toho, že rekurzivní predikáty jsou uzavřené na konjunkci, disjunkci a negaci (to ale dokázat neumim)
    - Mimochodem rekurzivní predikáty nejsou uzavřené na kvantifikaci, přidáním existenčního kvantifikátoru z toho uděláme částečně rekurzivní predikát
- Rekurzivní množinu lze reprezentovat buď výčtem hodnot (odpovídá úsekové funkci), nebo bitovým polem (odpovídá hodnotám char. funkce) - potenciálně nekonečné

Rekurzivně spočetné množiny a jejich vlastnosti

- U rekurzivně spočetných množin jsme schopni efektivně ověřit, že daný prvek do množiny patří, pokud nám někdo dá důkaz tohoto faktu (?). Rekurzivně spočetné množiny jsme schopni efektivně generovat, ale obecně nikoli systematicky, tj. můžeme vypsat prvky, ale bez řazení a můžou se opakovat
- Uzavřené na existenční kvantifikaci (dokonce efektivně)
    - Díky tomu že RSP jsou uzavřeny na exist (důkaz přes zakódování n-tic => jedna kvantifikace)
- Uzavřené na sjednocení a průnik
    - Zase z toho, že ty predikáty jsou uzavřený na konjunkci a disjunkci
    - Nejsou nicméně uzavřený na doplněk
- A množina, pak NTJE
    - A je RS
    - A je prázdná, nebo je oborem hodnot nějaké ORF (! tedy máme způsob jak množinu generovat)
        - Generování dělám tak, že dostávám jako parametr dvojici <x, s> a zkouším konečnou aproximaci do s kroků, jestli přijme x, vypíšu ho, jestli ne, vypíšu jiný prvek o kterém vím že do množiny patří (což můžu protože není prázdná)
    - A je oborem hodnot nějaké ČRF
    - A je oborem hodnot rostoucí ČRF
- Platí Postova věta - tady by se možná i dala dokázat, idea je že vezmeme ty komplementární RS množiny a děláme minimalizaci s testem náležení do konečných aproximací (daných řídící proměnnou) těch množin, což je sice brutálně neefektivní, ale

časem se musíme dostat k nějaké mezi, kdy nám hodnota buď do jedné, nebo do druhé aproximace padne => rekurzivní

Algoritmicky nerozhodnutelné problémy (halting problem). - ZSV, ADS2

- Obecně tedy algoritmicky rozhodnutelné problémy odpovídají rekurzivním množinám, u rekurzivně spočetných nemáme zaručeno, že se TS zastaví

Halting problem na Turingově stroji

- Instance = (M, x), kde M je TS a x vstupní slovo, ptáme se, jestli $M(x)\downarrow$
- Zakódováno do jazyka: $L_{HALT} = \{w; x \mid w \text{ kóduje TS M a } M(x)\downarrow\}$
    - Je rekurzivně spočetný, ale ne rekurzivní
        - Předpokládáme že L' = je dopněk jazyka L_{HALT} RS, a že ho přijímá nějaký TS M
        - Sestrojíme stroj M_e který přijímá diagonálu L', tj takové stroje které se nezastaví sami na sobě
            - M_e můžeme sestrojit pomocí M
            - M_e přijme vstup právě když se zastaví
        - Zkoumáme w_e, tedy jestli se stroj zastaví sám na sobě
            - Pokud w_e z L, pak ho musí M_e přijmout, což je spor s definicí
            - Pokud w_e není z L, pak by ho měl M_e odmítnout, ale tím pádem by zas měl patřit do jazyka

Halting problém v ČRF

- Problém zastavení: $K_0 = \{\langle x,y\rangle \mid \varphi_x(y)\downarrow\} = \{\langle x,y\rangle \mid y \text{ žije ve } W_x\}$
- Diagonála problému zastavení: $K = \{x \mid \varphi_x(x)\downarrow\} = \{x \mid x \text{ žije ve } W_x\}$
- Množiny K a K_0 jsou rekurzivně spočetné, ale ne rekurzivní
    - Dk (jsou rekurzivně spočetné): Z definice, $K = \text{dom } \varphi_z(x, x)$, $K_0 = \text{dom } \varphi_z(x, y)$
        - Pozn.: $\varphi_z(e, x)$ je univerzální ČRF
    - Dk (K není rekurzivní): Sporem, předpokládáme, že K je rekurzivní => doplněk K (označme ~K, jinak se píše s pruhem) je rekurzivně spočetná
        - Nechť φ_e je ČRF, pro níž $\sim K = \text{dom } \varphi_e$, ptáme se, zda e žije v ~K
        - Z definice K: e žije v ~K $\Leftrightarrow \varphi_e(e)\uparrow$
        - Z toho, že $\sim K = \text{dom } \varphi_e$: e žije v ~K $\Leftrightarrow \varphi_e(e)\downarrow$
    - Dk (K_0 není rekurzivní): Pro spor zase nechť K_0 rekurzivní => má charakteristickou funkci $\chi_0(\langle x,y\rangle)$
        - Definujeme $\chi(x) = \chi_0(\langle x,x\rangle)$, pak by χ byla charakteristickou funkcí K, což by znamenalo, že K je rekurzivní, což ale není
        - Dá se to pojmout i lidštěji, K_0 je obecnější problém, než K, takže musí být aspoň tak těžký, jako K a ten je rekurzivně spočetný

Převoditelnost

- A je m-převoditelná na B ($A \leq_m B$), pokud exje ORF f: x žije v A $\Leftrightarrow$ f(x) žije v B
- A je 1-převoditelná na B ($A \leq_1 B$), pokud je m-převoditelná a f je prostá
- Množina je m/1-úplná pro třídu rekurzivně spočetných množin, pokud je rekurzivně spočetná a každá jiná rekurzivně spočetná množina je na ni m/1-převoditelná
    - Když A <= B a B je rekurzivní/rek. spoč., pak i A je rekurzivní/rek. spoč.
    - Relace <= je reflexivní a tranzitivní
    - A m-úplná, B RS, A <= B, pak B je m-úplná

- A <= B implikuje A' <= B
- Množiny K_0 a K jsou 1-úplné
  - Důkaz

Věty o rekurzi a jejich aplikace, Riceova věta.- ZSV

- Pozn.: Rice je speciálním případem věty o rekurzi

Riceova věta

- Znění: C libovolná třída ČRF => A_C = {e | φ_e žije v C} je rekurzivní ⇔ C je prázdná, nebo C obsahuje všechny ČRF
- Důkaz: Pokud je C prázdná, pak je i A_C prázdná, tedy rekurzivní. Pokud C obsahuje všechny ČRF, pak C = N (přirozená čísla), tedy opět rekurzivní. Zbývá tedy případ, kdy C neprázdná i neplná, ukážeme, že lze K 1-převést na A_C nebo na $\sim A_C$, z čehož bude plynout, že A_C nerekurzivní, protože K je nerekurzivní
  - def: a: $\varphi_a\uparrow$, předpokládejme, že φ_a nežije v C, tedy a žije v $\sim A_C$; ukážeme, že K $\leq_1 A_C$ … // kdyby φ_a žilo v C, obrátíme role A_C a $\sim A_C$
  - def: b: φ_b žije v C, tedy b žije v A_C, b ≠ a
  - def $\varphi_c(x, y) = \varphi_b(y)$, pokud x žije v K, jinak ↑
  - def f(x) = s(c, x) //s-m-n věta// => $\varphi_{f(x)}(y) = \varphi_c(x, y)$, z toho dostaneme:
  - buď x žije v K => $\varphi_{f(x)} = \varphi_b$ => $\varphi_{f(x)}$ žije v C => f(x) žije v A_C
  - nebo x nežije v K => $\varphi_{f(x)} = \varphi_a$ => $\varphi_{f(x)}$ nežije v C => f(x) žije v $\sim A_C$
  - => tedy x žije v K ⇔ f(x) žije v A_C
  - no a protože f(x) je prostá ORF => z definice K $\leq_1 A_C$
- Důsledky: Netriviální dotazy o algoritmech nejsou algoritmicky rozhodnutelné
  - To zahrnuje například dotaz, jestli je algoritmus konečný

Věta o rekurzi

- Znění: Pro libovolnou ORF jedné proměnné f(x) exje n: $\varphi_n = \varphi_{f(n)}$
  - n se nazývá pevný bod
  - Lidsky: f je nějaká transformace algoritmu, věta říká, že pro každou transformaci existuje nějaký program (n), který i po provedení transformace počítá stejnou funkci
- Důkaz:
  - Diagonalizace, uděláme si matici funkcí :-): $(r, u) = \varphi_{\varphi r(u)}$
    - Tzn. na diagonále jsou funkce $\varphi_{\varphi u(u)}$
    - Pokud $\varphi_r(u)\uparrow$ => na (r, u) je nedefinovaná funkce
  - def a: $\varphi_a(u, x) = \varphi_{\varphi u(u)}(x)$
  - def $\varphi_e(u) = d(u) = s(a, u)$ //s-m-n věta//, jde o prostou PRF
  - platí: $\varphi_{\varphi e(u)}(x) = \varphi_{d(u)}(x) = \varphi_{s(a,u)}(x) = \varphi_a(u,x) = \varphi_{\varphi u(u)}(x)$
  - tedy prvky matice (e, u) = (u, u), pro všechna u, tzn. e-tý řádek se rovná diagonále
    - obsahují prvky $D = R_e = \{\varphi_{\varphi e(u)} = \varphi_{d(u)}\}$
  - ORF f přemapuje řádek R_e na R_v, kde $\varphi_v = f \cdot d$
    - nyní tedy $D = R_v = \{\varphi_{\varphi v(x)} = \varphi_{f(d(x))}\}$, pro x z N

    - protože řádek R_e byl shodný s diagonálou => prvek na (e, v) = prvku na (v, v) … => $\varphi_{d(v)} = \varphi_{f(d(v))}$
    - položíme tedy n = d(v) a máme pevný bod funkce f
- Důsledky:
    - Existuje prostá PRF g, která ke Gödelovu číslu funkce f určí její pevný bod
        - Plyne z důkazu VoR, ta funkce g = f · d
    - Každá ORF f má nekonečně mnoho pevných bodů
        - Dk: Funkce f · d má stejně jako všechny ČRF nekonečně mnoho Gödelových čísel, z každého z nich dosazením do d dostaneme pevný bod funkce f, protože d je prostá => pevných bodů je nekonečno
    - Existuje n: $W_n = \{n\}$
    - Existuje n: $\varphi_n(x) = n$
        - Lidsky: existuje program, který vypisuje sám sebe
        - Dk: def. $\varphi_e(x, y) = x$
            - def: f(x) = s(e, x), dostaneme $\varphi_{f(x)}(y) = \varphi_e(x, y) = x$
            - S použitím VoR nalezneme n: $\varphi_n(y) = \varphi_{f(n)}(y) = n$
    - Taky lze pomocí VoR dokázat Rice
        - Vezmeme a z A_C a b co není z A_C
        - Definujeme f(x) jako
            - a, pokud x není z A_C
            - b, pokud x je z A_C
        - Pak pro n které je pevný bod f ukážeme spor s náležením do C

Datové struktury (11/11)

Binární stromy a jejich vyvažování - PG II, ADS

- Uspořádaný slovníkový problém: Máme totálně uspořádané universum U, chceme reprezentovat jeho podmnožinu S a co nejefektivněji navrhnout operace MEMBER, INSERT, DELETE, MIN, MAX. Očekáváme, že se uspořádání mění jen minimálně.

Binární vyhledávací stromy (BVS)

- Binární vyhledávací strom T reprezentující množinu S je binární strom (každý vrchol má max 2 syny), ve kterém platí:
  - Každý vrchol reprezentuje prvek z S (a každý prvek z S má odpovídající vrchol)
  - Všechny vrcholy podstromu levého syna jsou <= v
  - Všechny vrcholy podstromu pravého syna jsou > v
  - Někdy se přidává navíc jeden level s listy, které reprezentují intervaly (třeba Koubek to tam má, ale Koubek je magor)
- Základní implementace:
  - Pomocná funkce FIND(x) začne v kořeni a postupuje stromem, když najde x, vrátí příslušný vrchol, jinak tam vrchol není
  - MEMBER(x): FIND(x) nám prozradí, jestli je hodnota přítomna
  - INSERT(x): s FIND(x) najdeme vrchol v, pod který ten náš patří, pokud už x ve stromu je, neuděláme nic, jinak přidáme synátora
  - DELETE(x): v := FIND(x),
    - pokud je v list, prostě ho smažeme
    - pokud má v jednoho syna, smažeme a přepojíme
    - jinak dáme na jeho místo vrchol s nejmenší hodnotou z pravého podstromu a původní v prostě smažeme (analogicky můžeme použít největší hodnotu z levého podstromu)
  - MIN(): nejlevější prvek ve stromě
  - MAX(): nejpravější prvek ve stromě
  - SUCC(x), PRED(x)
- Všechny operace mají složitost O(hloubka stromu), což může při ošklivé degradaci být až O(n)

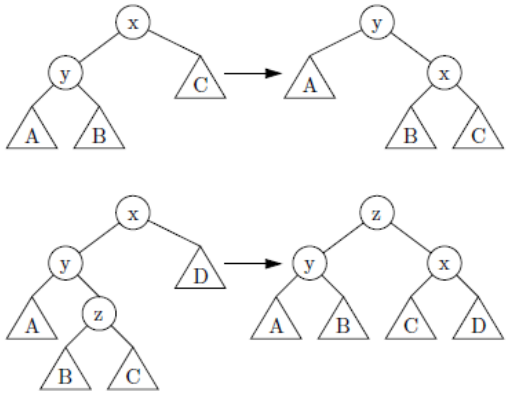
Vyvažování BVS

- Dokonalé vyvážení: pro všechny vrcholy platí, že rozdíl počtu vrcholů levého a pravého podstromu je maximálně 1. Těžké udržet.
- Hloubkové vyvážení: pro všechny vrcholy platí, že rozdíl hloubky pravého a levého podstromu je maximálně 1. Slabší, ale mnohem jednodušší udržet.

AVL strom

- Hloubkově vyvážený BVS.
- (1) AVL strom o n vrcholech má hloubku O(log n).
  - Plyne ze vztahu “a_K >= 2^(k/2)”, kde a_K = min. # vrcholů AVL stromu hloubky k
  - Dokazujeme indukcí od kořene:
    - $a_1 = 1$, $a_2 = 2$, $a_3 = 4$, $a_K = 1 + a_{K-1} + a_{K-2}$

        - $a_K = 1 + a_{K-1} + a_{K-2} > 2^{((k-1)/2)} + 2^{((k-2)/2)} >= 2^{(k/2)}$
- Operace identické jako BVS, až na vyvažování: jednoduchá a dvojitá rotace (viz obrázky níže). Jednoduchá rotace je navzdory šipce obousměrná. Druhý směr pro dvojitou rotaci analogicky.



- INSERT, DELETE: při průchodu dolů si děláme značky
    - +... hlubší je pravý podstrom
    - -... hlubší je levý podstrom
    - o… oba podstromy jsou vyvážené
- INSERT, DELETE: při zpětném průchodu (směrem nahoru) upravujeme uložené značky. Porovnáváme výsledek operace (zajímá nás, jestli se hloubka inkriminovaného podstromu snížila, zvýšila či zůstala stejná) nebo značku vrcholu, který jsme právě opustili, se značkou aktuálně zkoumaného vrcholu (ještě neupravenou).
    - Pro vyvážení stromu po INSERT stačí jediná rotace, i kdyby to bylo až v kořeni.
    - Pro vyvážení stromu po DELETE už může být potřeba mnoho rotací.
    - Jsou-li zkoumané značky (-, -) či (+, +) a naše cesta jim odpovídá, provedeme jednoduchou rotaci. Jsou-li zkoumané značky (-, +) či (+, -) a naše cesta jim odpovídá, provedeme dvojitou rotaci. Rozmysleme si, proč je vůbec potřeba rotaci provádět (proč musí být aktuálně zkoumaný vrchol nevyvážený).
- Složitost operací O(log n)... plyne z (1)

Červeno-černé stromy

- ČČ strom je BVS s následujícími vlastnostmi:
    - Každý uzel je buď červený nebo černý
    - List (vždy virtuální, může reprezentovat interval) je černý
    - Červený vrchol má oba syny černé
    - Všechny cesty od daného vrcholu k listům obsahují stejný počet černých uzlů
- (2) Díky definici platí pro každé dvě cesty z daného vrcholu do jeho listů, že jedna může být max. dvakrát delší než druhá. Strom je tedy “intuitivně” hrubě vyvážen.
    - bh(x)... počet černých uzlů na cestě z vrcholu x do listů
    - Indukce od listů: podstrom vrcholu x obsahuje >= 2^bh(x) - 1 vnitřních uzlů
    - Pro kořen už přímo plyne logaritmická hloubka - maximálně 2 . log(n+1)
- Jiný pohled na důkaz výšky: převodem na (2,4)-strom, o němž víme, že má logaritmickou hloubku. Převod se dělá tak, že červené vrcholy posadíme na úroveň

jejich černých otců. Jelikož původní ČČ strom byl max. dvakrát vyšší, jeho hloubka je O(2 log n) = O(log n)

- INSERT: cheatsheet. Nový vrchol červený, listy černé a neobsahují hodnotu. Pokud je otec nového vrcholu červený, máme průser => buď přebarvíme otce, strejdu a dědu, anebo nejprve rotujeme a potom přebarvujeme).
- DELETE: cheatsheet. Prostě guláš.

Haldy - ADS2

- Uspořádaný slovníkový problém: Máme totálně uspořádané universum U, chceme reprezentovat jeho podmnožinu S a co nejefektivněji navrhnout operace INSERT, MIN, DELETEMIN, DECREASE a INCREASE. Očekáváme tedy, že se uspořádání může hojně měnit.

Halda je strom, v němž platí:

- Každý vrchol reprezentuje nějaký prvek z S
- Každý vrchol reprezentuje menší prvek než všichni jeho synové (lokální podmínka haldy)

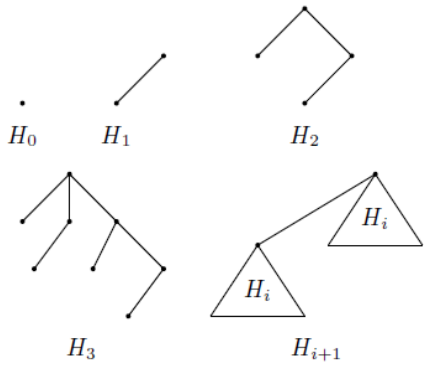
D-regulární halda (d > 1)

- Když očíslujeme vrcholy průchodem do šířky (zleva-doprava, odshora-dolů), platí:
  - Každý vrchol má nejvýše d synů
  - Když vrchol není list, pak všechny vrcholy s menším číslem mají právě d synů
  - Když má vrchol méně než d synů, pak všechny vrcholy s větším číslem jsou listy
- Důsledky definice:
  - Hloubka $O(\log_d n)$
  - Existuje max. 1 vrchol s 1 až d-1 syny - ostatní jich mají buď d anebo 0
- Snadná implementace a orientace prostřednictvím pole. Při číslování z definice a začínáme-li nulou, pak pro vrchol s indexem k jsou:
  - indexy jeho synů: (d . k + 1) až (d . k + d)
  - index otce: (k - 1) div d
- Pomocné operace/značení:
  - UP(v): dokud není splněna podmínka haldy, měníme menšího syna za většího otce
  - DOWN(v): dokud není splněna podmínka haldy, měníme většího otce za nejmenšího syna
- INSERT(v): v se stane posledním listem, na nějž zavoláme UP
- MIN: vrátí hodnotu kořene
- DELETEMIN: kořen nahradíme posledním listem, načež na něj zavoláme DOWN
- DELETE(v): v nahradíme posledním listem, načež na něj zavoláme UP či DOWN
- INCREASE/DECREASE(v): upravíme v, načež na něj zavoláme DOWN/UP
- MAKEHEAP(S): jak dostaneme S, tak postavíme strom (s polem lineárně) a postupně na každý vrchol počínaje posledním ne-listem voláme DOWN (pořadí určuje číslování z definice)
- Složitosti: MIN konstantní, MAKEHEAP O(d |S|), UP O(log |S|), DOWN O(d log |S|)

    - MAKEHEAP je tedy lineární a pro větší S efektivnější než opakovaný INSERT
- Poznámka: DELETE, INCREASE a DECREASE nepřijímají hodnoty, nýbrž reference na vrcholy (jinak bez speciálních datových struktur nezachováme logaritmickou složitost).
- Aplikace: Heapsort, Dijkstra.

Binomiální halda

- Binomiální strom H_i: H_0 je jednoprvkový a H_{i+1} vznikne propojením dvou H_i (kořen jednoho se stane krajním synem kořene druhého). Jednoduchou indukcí dostaneme:
    - (1): H_i má 2^i vrcholů
    - (2): Kořen H_i má i synů, které jsou jeden po druhém kořeny H_0, ... , H_{i-1}
    - (3): Výška H_i (nejdelší cesta z kořene do listu) je i



- Binomiální halda reprezentující |S| je seznam stromů T_i takový, že existuje bijekce mezi S a sjednocením vrcholů všech T_i, přičemž každý T_i je isomorfní s nějakým H_i a platí pro něj podmínka haldy. Vzájemně T_i isomorfní nejsou.
    - Díky (1) plyne existence binomiální haldy pro S z existence binárního zápisu a:
    - (4): Max. počet T_i je tedy řádově O(log |S|)
- Všechny operace jsou založeny na MERGE (resp. na binárním sčítání s přenosem)
    - SPOJ(H_i, H_i): ten s vyšším klíčem v kořeni se stane krajním synem kořene druhého. SPOJ je konstantní a MERGE má tedy O(log |S|).
- INSERT(v): MERGE s jednoprvkovou binomiální haldou
- MIN: projde kořeny a vypíše nejnižší
- DELETEMIN: odebere nejmenší kořen a všechny jeho syny MERGUJE
- INCREASE a DECREASE: jako u regulárních hald
- MAKEHEAP: jednoduše opakujeme INSERT (nic efektivnějšího nás nenapadá)
- Složitost INSERT/MERGE/MIN je zřejmě O(log |S|). Složitost DECREASE je díky (3) opět O(log |S|). Složitost INCREASE už je O(log^2 |S|) - plyne z (2) až (4). DELETEMIN je znovu O(log |S|), ale je třeba si rozmyslet, jak na sebe v tomto speciálním případě jednotlivá volání MERGE působí (intuitivně to člověka svádí k O(log^2 |S|)).
- LÍNÁ IMPLEMENTACE: rušíme POUZE předpoklad o isomorfismu T_i s H_i, nikoli lokální podmínku haldy. Korekci (MERGE) děláme až na poslední chvíli (DELETEMIN a MIN).
    - INSERT/MERGE pak jsou konstantní - prostá konkatenace seznamů.
    - MIN/DELETEMIN v nejhorším O(|S|) - samé jednoprvkové stromy. Amortizovaně však O(log |S|).

Fibonacciho haldy

- Fibonacciho halda reprezentující |S| je seznam stromů T_i takový, že existuje bijekce mezi S a sjednocením vrcholů všech T_i, přičemž pro každý strom platí podmínka haldy. Dále pro každé dva různé T_i a T_j platí: rank(T_i) != rank(T_j). Rank(T) = počet synů kořene. Vrchol stromu je označený ⇔ není kořen a byl mu odříznut alespoň jeden syn.
- Oproti binomiální haldě záměrně nespecifikujeme strukturu stromů (kromě synů kořene), což vede na flexibilní a efektivní DECREASE/INCREASE, přičemž amortizovaně zachováváme DELETEMIN. Při líné implementaci se FH hodí na průběžné počítání (např. Dijkstru), z nějž si postupně odebíráme minima (ideálně jen několik po sobě). Empiricky bylo potvrzeno zlepšení oproti regulárním haldám pro velké řídké grafy.
- Přejímá od binomiální haldy operaci SPOJ(T_i, T_i) operující nad dvěma stromy, kde kořeny mají stejný počet synů. Vyvažování pomocí operací:
    - VYVAZ1: dostane dva seznamy stromů a volá SPOJ, dokud je to potřeba
    - VYVAZ2(v): dokud mezi v a jeho předky nalézá označené vrcholy, odtrhává je společně s jejích podstromy a vkládá do haldy. První neoznačený vrchol označí (pokud to není kořen) a končí. Každému vrcholu tedy odebere právě dva syny, než z něj udělá kořen jiného stromu.
- MERGE a INSERT stejné jako u líné binomiální haldy (konkatenace seznamů)
- MIN: vybereme nejmenší kořen a zavoláme VYVAZ1
- DELETEMIN: vybereme a smažeme nejmenší kořen, podstromy vložíme do haldy a zavoláme VYVAZ1
- DECREASE(v, k): je-li v kořen, snadné. Jinak odtrhne v, vloží do haldy, sníží hodnotu a volá VYVAZ2(předek v)
- INCREASE(v, k): zvýší hodnotu, podstromy v vkládá do haldy a volá VYVAZ2(v)
- Poznámka: schválně u DECREASE a INCREASE děláme naprosté minimum. Až dosud byly haldy “operačně vyvážené” - tahle je spíše “specializovaná” pro změny uspořádání.
- DELETE(v): DECREASE(v, -nekonečno) a DELETEMIN
    - může být poměrně náročná operace, takže DELETE nechceme používat často
- Složitosti (líná varianta):
    - MERGE a INSERT: O(1)
    - MIN/DELETEMIN: amortizovaně O(log |S|), max. O(|S|)
    - DECREASE: O(1 + # odtrhnutých předků)
    - INCREASE a DELETE: O(1 + # odtrhnutých předků + # synů v)
- Problém je s dokazováním složitostí, ale stačí si pamatovat několik vlastností:
    - (1) Indukcí od listů: size(T) ≥ $F_{rank(T)+2}$
        - Jinými slovy: “má-li vrchol v i synů, potom počet vrcholů podstromu s kořenem ve v má alespoň F_{i+2} vrcholů (fibonnaciho číslo)”.
        - Při indukci používáme indukční předpoklad a rank vrcholů odvodíme z pomocného lemmatu: i-tý syn vrcholu v má alespoň (i - 2) synů, pokud vrchol s i = 1 je nejstarší
            - plyne ze SPOJ a VYVAZ2
    - (2) Max. rank stromu ve FH je O(log |S|)
        - Plyne z (1) pro případ, kdy size(T) = |S|.

        - Indukcí si dokážeme pomocné lemma: $F_n \geq ((1 + \sqrt{5}) \; div \; 2)^n$; n >= 3
            - tedy dolní odhad fibonnaciho čísel
        - Potom: $|S| \geq ((1 + \sqrt{5}) \; div \; 2)^{rank(T) + 2}$; rank(T) >= 1
            - po zlogaritmování obou stran jasně vidíme kýžený výsledek
    - (3) Počet stromů ve vyvážené FH je O(log |S|)
        - Plyne přímo z definice a (2)
    - (4) Důkaz amortizované složitosti… TODO

Trie - ADS2, Wiki

- Alias: prefixový strom (prefix tree). Mezi varianty se řadí také digital tree nebo radix tree.
- Buď Σ konečná a uspořádaná abeceda velikosti K a U všechna slova nad Σ. Chceme reprezentovat množinu S ⊂ U a efektivní operace MEMBER, INSERT a DELETE. Díky předpokladům můžeme MEMBER snadno použít i na prefixy prvků S.
- Hrany trie (příp. vrcholy) jsou ohodnoceny jedním prvkem z abecedy. Klíč je pak dán hranami na cestě od kořene. U každého vrcholu si musíme pamatovat, jestli představuje reprezentované slovo. V podstatě se tedy jedná o konečný automat.
- Operace:
    - MEMBER: jde po stromě a vyhlásí přítomnost slova
    - INSERT: jde po stromě a příp. připojí nové vrcholy
    - DELETE: najde vrchol reprezentující dané slovo, smaže ho a všechny jeho listové předchůdce, až dokud nenarazí na první ne-list nebo vrchol reprezentující jiné slovo.
- Složitosti:
    - MEMBER(s): O(|s|)
    - INSERT(s) a DELETE(s): záleží na implementaci…
        - Přihrádkové třídění (je-li abeceda navíc indexovaná): O(|s|)
        - Binární vyhledávání (neindexovaná pole): O(log K . |s|)
        - Nejhůře každopádně: O(K . |s|)
- Aplikace:
    - Algoritmus (Aho-Corasick) - vyhledávání množiny slov ve fulltextu
    - Třídění slov (lexikografické uspořádání vypíšeme pomocí preorder průchodu)
    - Slovníky pro detekci překlepů nebo predikci psaní (autocomplete)
- Aho-Corasick (kdyby zkoušející požadoval):
    - Vyhledáváme větší počet slov (jinak si vystačíme třeba s Rabin-Karpem)
    - Trie obohacena o zpětné hrany, které vedou z daného vrcholu (slova S) do jiného vrcholu - ten reprezentuje nejdelší sufix S takový, který je zároveň prefixem (byť třeba vlastním) některého z hledaných slov.
        - Např. slovník {barbara, barman}. Jsme ve stavu {barbar} a další písmeno je “m”. Nemůžeme se vrátit do počátečního stavu, protože bychom ztratili přečtený suffix “bar”. Z {barbar} nám tedy vede zpětná hrana do {bar}, kde můžeme pokračovat znakem “m”, až do {barman}.
    - Konstrukce zpětných hran induktivně po hladinách (až po vytvoření trie):

        - z(u)... zpětná hrana z vrcholu u do z(u); implicitně kořen
        - Q… fronta vrcholů (nastaveno na syny kořene)
        - Opakuj: vyzvedni z Q vrchol u a pro každého jeho syna v zjisti, kam bychom se mohli dostat ze z(u) přečtením písmene mezi u a v. To bude nové z(v), načež v přidáme do Q.
    - Zpětné hrany můžeme přeskakovat ve zkratkových hranách, abychom usnadnili vypisování namatchovaných slov (jedno je sufixem druhého). Hodí se snad jen pro malé abecedy a velké S.
- Srovnání s hašováním:
    - + nemusíme se zabývat výběrem vhodné hashovací funkce, resp. očekávanou doménou slov
    - + pro krátká slova rychlejší
    - - pro dlouhá slova pomalejší (převládnou postihy za cache miss)
    - - trie vyžaduje mnohem více náhodného přístupu, takže při pomalém médiu rychle převáží spíše hašování (s tím se v praxi snad ani nesetkáme)
    - + nemusíme alokovat záložní prostor (hašování se točí kolem pojmu naplnění), ale zase musíme efektivně implementovat přechody a velká abeceda = velká daň
- Bitwise trie - speciální varianta, kde Σ je {0,1}. Při reprezentaci uzlů trie v poli (viz regulární haldy) je výhodou existence speciální CPU instrukce, resp. makra, které pro dané binární slovo velmi rychle (na bázi bitových operací) najdou index prvního nenastaveného bitu v poli. Možná bychom ale museli předpokládat slova omezené délky. Obecně lze téhle varianty využít pro libovolnou abecedu (klíče převádíme do binárního formátu).
    - Vede to na lepší využití paměti, cache hit a paralelizaci (máme vyhrazeno místo pro každý uzel trie).
    - Prý dokonce často poráží ČČ stromy, přestože by měly být papírově lepší… to je pravděpodobně ta cache, no :).
- Komprese
    - Motivace: obyčejná trie potřebuje O(K . součet délek slov + počet slov) paměti
    - Komprese je vhodná jen pro víceméně statické trie.
    - Metoda #1: cheatsheet. Zkrátka slučujeme uzly a hrany, které nepřipouštějí žádné větvení. Uzly pak nejsou adresovány jedním znakem, nýbrž řetězcem proměnlivé délky a ukládání klíčů není zcela příjemné. Nakonec je často překládáme na řetězce pevné délky, např. takto:
        - Řetězce proměnlivé délky uložíme do pole.
        - V uzlech ukládáme trojice: {index do pole, index prvního znaku, index posledního znaku}
    - Ideální užití pro malý počet dlouhých a předem známých slov s velkou abecedou.
    - Metoda #2: odstranění cyklů, např.:
        - Mějme slova “top”, “tap”. Obyčejná trie: vrcholy “t”, “to”, “ta”, “top”, “tap” - kdybychom spojili “to” a “ta”, můžeme spojit “top” a “tap” (odstranit cyklus). S trochou dodatečné komplexe by ještě šlo aplikovat metodu #1.

- Nevýhoda: výsledná trie netuší, jak se do uzlu dostala - ví jen, že aktuálně přijímá slovo z S

B-stromy a jejich varianty - OZD

- B-strom řádu m:
    - Kořen je buďto listem, anebo má minimálně dva potomky
    - Všechny listy jsou na stejné hladině
    - Každý vnitřní uzel má ceil(m/2) až m potomků a o 1 méně klíčů
    - Klíče jsou v uzlu setříděně a každá jejich dvojice (nebo jen ty krajní) definuje podstrom (potomka), jehož hodnoty jsou omezeny intervalem dané dvojice klíčů
- Vlastnosti:
    - Vnitřní uzly B-stromu jsou alespoň z poloviny plné
    - B-strom řádu m je speciální případ (a,b)-stromu, resp. (ceil(m/2), m)-strom
- Vhodný pro velké datasety (databáze) - velký počet synů v každém uzlu minimalizuje IO operace (uzel ideálně odpovídá bloku paměti či logické stránce). Převážně se očekává blokové (sekvenční) čtení. Zápis může být klidně náhodný (vyvažovat nemusíme často).
- MEMBER: triviální. Správný interval (podstrom) nalezneme binárním vyhledáváním.
- INSERT: najdeme správný list, vložíme nový klíč a hodnotu. Je-li list přeplněný (m klíčů), prostřední klíč posuneme o hladinu výš a uzel rozštěpíme na dva, které nyní budou zaplněné minimálně: floor((m - 1) div 2). Když si to člověk rozmyslí, vychází to jak pro sudá tak i lichá m.
    - Pokud musíme, rekurzivně štěpíme předky. Nejhůře zvýšíme výšku stromu.
    - Bloky předků ale mohou již být odstraněné z cache - můžeme preventivně štěpit naplněné uzly (m - 1 klíčů) při MEMBER a INSERT, za cenu snížení limitu minimálního naplnění.
- DELETE:
    - Najdeme vrchol s klíčem a společně s hodnotou smažeme. Pokud nemáme dostatek klíčů a jeden ze sourozenců má minimální počet klíčů, můžeme je spojit dohromady. V opačném případě si můžeme od jednoho ze sourozenců půjčit (rotovat klíče přes předka).
    - Kdybychom spojením z prvního bodu způsobili nedostatek klíčů v předkovi, opakujeme na něj první bod a postupně takhle rekurzíme dále.
- Označíme-li N počet listů/prvků, pak je hloubka B-stromu $O(\log_m N)$
    - Kdyby někdo chtěl důkaz něčeho tak zřejmého, mějme B-strom hloubky K:
        - $N \in \langle 2 \cdot ceil(m/2)^{(K-2)}; 2 \cdot m^{(K-2)} \rangle$. Po zlogaritmování se (K-2), ½ a 2 přemění v multiplikativní konstantu. Toť vše :).
- Složitost operací je $O(\log_m N)$, ale skrývá se tam i multiplikativní konstanta log M a maximální velikost klíče

Varianty B-stromů

- B+ strom
    - Redundantní B-strom… hodnoty ke klíčům ukládá pouze do listů (a tím opakuje klíče ve vnitřních uzlech). Obyčejné variantě se říká neredundantní B-strom.
    - Přirozeně rozšiřuje hladinu listů o ukazatele na sourozence (intervalové dotazy).
    - Aplikace: DB indexy a filesystémy (indexace metadat)
    - Úprava INSERT:

        - Najdeme správný list a vložíme klíč/hodnotu. Pokud se klíč nevejde, rozštěpíme list a do otce ZKOPÍRUJEME nejmenší klíč pravého uzlu (slouží pouze pro vyhledávání). Štěpíme dále, pokud potřeba...
    - Vnitřní klíče jde komprimovat (jak? souvisí to s Grayovými kódy?)
- B* strom
    - Místo polovičního naplnění je spodní hranicí naplnění ze ⅔. Místo slučování párů a štěpení jednotlivců tedy slučujeme trojice (do páru) a štěpíme páry (na trojice). K tomu musí být dané páry/trojice naplněné zcela/minimálně, takže se snažíme přebytečné klíče předat sousedům.
    - DELETE je “somewhat more complex, however” :D

Hašování a řešení kolizí - DS, OZD

Hašování (klasické či statické - žádné šmajchly s funkcemi):

- Znovu slovníkový problém: chceme reprezentovat nějakou množinu S $\subset$ U (universum) a navrhnout efektivní MEMBER, INSERT, DELETE. Uspořádání nás nezajímá.
    - triviálním řešením je charakteristická funkce S reprezentovaná bitovým polem, dokážeme-li S transformovat na stejně velikou množinu indexů
        - navzdory okamžitému vyhodnocení bude ale velikost prostoru funkcí |U|
    - hašování je kompromisem mezi rychlostí a prostorem ch. fce, když |S| << |U|
        - obrovské zmenšení potřebného prostoru za cenu menšího zpomalení
- Prvky S ukládáme do hašovací tabulky. Značení:
    - h : S $\rightarrow$ X (hašovací funkce)
    - |S| = n
    - |X| = m (rozsah haš. funkce)
    - α = n / m (load factor), tj. faktor zaplnění haš. tabulky
- Předpoklady (jen teoretické, pro snadnější dokazování - praxe je ošemetnější):
    - (1) Haš. funkce počítá v O(1)... velikost slov z S je tedy omezena nějakou vhodnou konstantou.
    - (2) Haš. funkce rozděluje U na X rovnoměrně.
    - (3) S je rovnoměrně rozdělená (“všechny důležité kategorie dat v ní mají přibližně stejný počet zástupců”)

Kolize:

- Když h(s1) = h(s2), s1 != s2
- Řešení #1: separované řetězce formou spojáku
    - Každá buňka haš. tabulky (reprezentuje hodnotu X) obsahuje spoják
    - Neuspořádané spojáky
        - Při (2) a (3) bude průměrná délka spojáku právě α. Úspěšné vyhledávání: O(1 + α). Neúspěšné vyhledávání: $e^{-\alpha} + \alpha$
    - Uspořádané spojáky
        - Při (2) a (3) bude průměrná délka spojáku právě α. Úspěšné vyhledávání: O(1 + α). “Lepší” neúspěšné vyhledávání: $e^{-\alpha} + 1 + \alpha/2 - (1/\alpha) * (1 - e^{-\alpha})$ za cenu zpomalení INSERTu o faktor α (přímé vkládání).
    - Pozorování: separované řetězce nevyužívají přebytečnou paměť alokovanou v tabulce a jsou méně cache-friendly. |X| nemusí být nutně větší než |S|, může být i menší - podle velikosti α, se kterou se dokážeme smířit. Užití spojáků by mělo být výhodné hlavně v případech, kdy nás zajímá uspořádání prvků v buňkách nebo pouze první či poslední prvek, ať už je jakýkoli.
- Řešení #2: separované řetězce přes tabulku
    - Pozorování: budeme-li klíče (prvky z S) ukládat rovnou v tabulce (ne ve speciální paměti), o něco málo bychom měli zlepšit zavlečenou paměťovou složitost a díky cache i rychlost. Potom:
        - Každá buňka tabulky musí ukládat klíč a ukazatele na jiné buňky.
        - |X| > |S|, ideálně tak o třetinu až polovinu více.

    - Hašování s přemisťováním
        - Dva ukazatele: next a prev
        - Při INSERT(s) dáváme pozor, jestli h(s) už není obsazená - potom původní obsah přemisťujeme do jiné volné buňky, s vložíme do h(s) a vhodně upravujeme související next a prev (jinak příliš pomalé). Přemisťování zřejmě zvýhodňuje lokálně (temporálně) zpracovávané prvky (budou na první pozici).
        - Při DELETE(s) znovu přepojujeme ukazatele. Jestliže mažeme přímo h(s) a existuje next, musíme next přesunout do h(s) a přepojit ukazatele.
        - Problémy: orákulum volných buněk a přemisťování obsahu buněk ve prospěch jejich skutečných “vlastníků” (to je ale minimum práce)
    - Hašování se dvěma ukazateli
        - Idea: eliminovat druhý z problémů přemisťování - místo next a prev budeme mít next a first, kde first ukazuje na buňku, kde začíná vlastní řetězec. Při konfliktu tedy nemusíme přesouvat - stačí nastavit ukazatel.
        - INSERT(s) znovu vkládá do volných buněk a přenastavuje first h(s)
        - DELETE(s) si musí pamatovat i přímého předchůdce
        - Problémy: i nadále potřebujeme dobré orákulum volných buněk
    - Hašování s lineárním přidáváním
        - Snaží se odstranit závislost na orákulu
        - INSERT(s) hledá první následující volnou buňku
        - DELETE(s) neexistuje
        - + více cache-friendly, ale taky narůstá složitost MEMBER
            - dobrá volba pro nižší α (< 0.6), tedy když máme dost paměti a dobrou haš. funkci
    - Dvojité hašování (dvě hašovací funkce)
        - Snaha o vylepšení lineárního přidávání
        - např. hi(x) = (k1(x) + i . k2(x)) mod m
            - k1(x) určí buňku jako předtím
            - k2(x) iterativně určuje volnou buňku v případě kolize (musí být nesoudělná s m, aby hi(x) vydávala prosté posloupnosti)
        - - méně cache-friendly, ale když nejsou vhodné podmínky pro lineární přidávání, je lepší
        - DELETE(s) neexistuje (vytvářel by díry, přes něž se ten další nedostane)
- Řešení #3: neseparované řetězce přes tabulku (srůstající hašování)
    - Idea: řetězce nezávislých buněk srůstají dohromady, protože buňky obsahují jen ukazatel next.
        - DELETE(s) neexistuje...
        - Znovu potřebujeme orákulum volných buněk...
    - Standardní schémata přidávání:
        - LISCH: přidává na konec seznamů
        - EISCH: přidává za první prvek seznamů

        - Atrapa DELETE(s): buňky lze značit jako odstraněné, i když fakticky je budeme přes seznamy prolézat i nadále… pomalé
    - Schémata přidávání s pomocnou pamětí bez přímého adresování (seznam):
        - Idea: hašovací funkce nemají přístup do pomocné paměti a kolizní klíče řetězíme nejprve pomocí buněk z pomocné paměti, a pak teprve pomocí buněk z tabulky (buňky mají stejný formát)
        - LICH: přidává na konec řetězce
        - EICH: přidává za první prvek řetězce
        - VICH: v pomocné paměti řetězí na konec a v tabulce za první prvek. Snaží se o co největší kompromis.

Univerzální hašování - DS, OZD

Myšlenka:

- Pro kteroukoli hašovací funkci můžeme zvolit špatný vstup (ve smyslu kolizí), takže lépe mít systém hašovacích funkcí a náhodně mezi nimi vybírat. Do jisté míry riziko výběru špatné funkce zůstane, ale bude výrazně nižší.

Značení:

- m… velikost haš. tabulky
- K... množina indexů haš. funkcí (1...k)

Univerzální systém hašovacích funkcí:

- H { h_i: U → {0, …, m-1} | i ∈ K } je c-univerzální, pokud pro všechny různé x, y ∈ U: |{i ∈ K: $h_i(x) = h_i(y)$}| <= c . |K| / m
    - V praxi budeme chtít (c . |K| / m) samozřejmě minimalizovat, takže čím menší c a |K|, tím lepší systém máme. Podobně přirozené je zvětšovat velikost tabulky (m), protože garantuje menší pravděpodobnost kolize, nicméně velikost tabulky je v reálném světě snad vždy omezená (fixovaná).

Pro rovnoměrný náhodný výběr funkce platí ekvivalentní tvrzení:

- P(h(x) = h(y)) <= c / m
    - Důkaz přes intuitivní lemma, že pro danou h je E[počet kolizních klíčů s x] < n/m
        - Cx… počet kolizí s x
        - Cx = suma(indikátory kolize c_{xy} přes všechna y co nejsou x)
        - ECx = suma středních hodnot indikátorů
        - ECxy = 1/m . 1 + (m - 1)/m . 0 = 1/m
            - c_{xy} má zřejmě alternativní rozdělení
            - pro dané x, y a h a jestliže h rozděluje U rovnoměrně!
        - ECx = (n - 1) / m < n / m
    - Pro důkaz by nás měla zajímat veličina c_{xy}, protože pro dané x, y a h je vlastně P(h(x) = h(y)) ekvivalentní P(c_{xy} = 1), tedy 1/m. Z toho vyplývá, že při splnění předpokladů je c >= 1. Všimněme si, že díky neurčitosti c v dokazovaném tvrzení nehraje roli požadavek na h, aby rovnoměrně rozdělovala U. V teorii i praxi je to ovšem velice žádoucí vzhledem ke snaze minimalizovat c.

Teoretická existence univerzálních systémů:

- Mějme U = {0, …, N - 1}, N prvočíslo, S = U, a systém:
  - H = {$h_{a,b}$ | a, b ∈ U}, $h_{a,b}(x)$ = ((ax + b) mod N) mod m}
  - Tzn. |H| = |K| = N^2. $h_{a,b}$ navíc umíme rychle vyčíslit
- Je systém c-univerzální? Jaké je c?
  - Krátká odpověď: funkcí je v systému opravdu hodně, ale c má blízko k ideálu (1).
  - Delší odpověď:
    - Zvolme si různá x, y ∈ U, chceme nalézt všechny dvojice (a, b) ∈ U×U: $h_{a,b}(x) = h_{a,b}(y)$.
    - (ax + b) mod N ≡ (ay + b) mod N [mod m] ⇔ existuje i < m a r,s < ceil(N/m) t.ž.:
      - ax + b ≡ i + rm [mod N]
      - ay + b ≡ i + sm [mod N]
    - Máme soustavu rovnic v Z_N (v pořádku, N jsme zvolili jako prvočíslo). X, Y a m jsou zafixované. Jestliže zafixujeme rovněž i, r, s, proměnné budou už jen A a B. Potom má soustava !1 řešení, když je levá matice regulární:

$$\begin{pmatrix} x & 1 \\ y & 1 \end{pmatrix}$$

    - Ta regulární vskutku je, protože x ≠ y. Pro každou dvojici (x, y) a trojici (i, r, s) tedy máme !1 kolizní funkci a pro dané x,y je celkový počet kolizních funkcí dán rozsahem (i, r, s):
      - m . ceil(N/m)^2… nezapomínejme na 0
    - Nyní zbývá dokázat, že počet kolizních funkcí pro (x, y) splňuje definici:
      - m . ceil(N/m)^2 <= c . N^2 / m
      - ceil(N/m)^2 <= c . (N/m)^2
      - c >= 1
    - Definice je splněna a c ∈ <1, 4).

Očekávaný čas operací (pro jednoduchost separované řetězce ve spojáku):

- MEMBER, INSERT a DELETE je O(1 + cα), kde α je faktor naplnění, tj. n/m
  - Intuitivní vysvětlení: pravděpodobnost kolize pro zafixované X,Y je c/m, takže když máme v tabulce už n prvků, střední hodnota počtu kolizí by se měla pohybovat okolo c . n/m.
    - figuruje tu ovšem tíživý předpoklad opakovaně rovnoměrného náhodného výběru funkce h a rovnoměrného rozdělení univerza každé h
  - Důkaz: správně by se intuitivní vysvětlení mělo počítat i přes pravděpodobnost výběru funkce, která kolizi způsobí
  - Důsledek: srovnatelné s klasickým hašováním, ale vstupní data již nemusí být rovnoměrně rozdělená. Namísto toho musí funkce rovnoměrně rozdělovat U a v každém kroku musíme h vybírat rovnoměrně náhodně.
  - Právě s rovnoměrně náhodným výběrem h je problém. V teoretickém modelu je |H| roven $|N|^2$, takže funkcí máme opravdu hodně. Generátory náhodných čísel jsou však pseudo-náhodné a přestože vždy vracejí hodnotu, jejich zdroj entropie

nemůže takový náklad utáhnout. Jinými slovy, čím větší |H|, tím méně náhody nám bude generátor vracet. Zároveň bychom generátor mohli vyčerpat jen v určitých časových intervalech, kdy v hašovacím systému voláme zvýšený počet operací. A náš hašovací systém nebude zdaleka jedinou aplikací, která na generátor spoléhá. V praxi bychom tedy |H| měli omezit vhodnou konstantou (nikoli velikostí univerza), a to v neprospěch konstanty c nebo velikosti tabulky.

Potírání pseudo-náhody - dolní odhad velikosti univerzálního systému:

- Idea: Mějme |H| = 1. Střední počet kolizí není konstantní: N/m (opět předpokládáme rovnoměrné rozdělení U). Vhodná druhá funkce by N/m zmírnila na něco podobného N/m^2. Potřebujeme tolik funkcí, aby se nám jmenovatel redukoval na O(N). Pak bude střední počet kolizí konstantní a máme c. TOHLE JE BLBOST (ať už je funkcí sebevíc, střední počet kolizí bude pořád n/m - prostě ty prvky nemáme kam jinam narvat, ať už je rozdělujeme sebelíp)
- Odhad a důkaz:
  - Funkce z H očíslujeme jako $\{h_i\}$. Rekurzivně definujeme U_i jako největší podmnožinu U_{i-1} t.ž. $h_{i-1}(U_i)$ je jednoprvková množina. $U_0 = U$.
  - Pak platí: $|U_i| <= ceil(|U_{i-1}|) / m$, tedy $|U_i| <= ceil(N/m^i)$
  - Velikost U_i klesá s logaritmem, takže funkcí je alespoň logaritmický počet vzhledem k velikosti universa

Potírání pseudo-náhody - konstrukce malého univerzálního systému:

- Zvolíme prvočíslo m (báze). Prvky U pak budeme vyjadřovat v bázi m (každá “cifra” v rozsahu 0...m-1). Zvolíme náhodné číslo v bázi m (a). Pak mějme funkci:
  - $h_a(x)$ = suma přes $i(a_i * x_i \bmod m)$
- Počet výstupních haší je zřejmě m^r, kde r je počet cifer
- Je to univerzální:
  - Dáme do kongruenční rovnice výpočet hashe pro oba prvky, odečteme
  - Vyndáme lišící se prvky, dostaneme:
    - $a_0(x_0 - y_0) == -$ suma přes $j > 1$ $a_j (x_j y_j)$ [mod m]
  - Díky tomu, že x_0 a y_0 jsou různé existuje k jejich rozdílu inverz, tím to vynásobíme a dostaneme konkrétní hodnotu pro a_0 která způsobí kolizi
  - Tj. máme r cifer, a když si vybereme náhodně r-1 z nich, ta poslední je jasně daná (aby x a y kolidovalo), tj ze všech funkcí jich koliduje pro konkrétní data m^(r-1), což je ale taky |H| / m… a máme co jsme chtěli

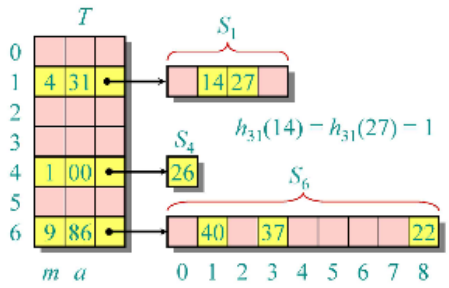
Perfektní hašování - DS, OZD

- Znovu je cílem bojovat s kolizemi. Namísto více hašovacích funkcí, vytvářejících různé kolize pro různá data, se nyní pokusíme kolize eliminovat za běhu postupnou adaptací našich funkcí či úložných prostorů. Logicky pak budeme muset tabulku přehašovávat, ledaže bychom data znali předem (téměř úplně), anebo měli z pekla kliku.
- Platí standardní předpoklady smysluplného hašování:
  - h(x) spočitatelná v konstantním čase
  - základní tabulka přibližně stejně velká jako cílová množina
- Navíc chceme:

    - h(x) != h(y) pro x != y ∈ S
    - h(x) nesmí být náročná na paměť (reprezentace) - tedy analytické vyjádření, nikoli tabulkové
    - Poznámka: čas nalezení nové perfektní funkce nás příliš nezajímá - operaci nepotřebujeme zas tak často

Teoretický model perfektního hašování:

- Množina funkcí H zobrazujících U (|U| = N) do M (|M| = m) se nazývá (N, m, n)-perfektní, pokud pro každou S ⊂ U (|S| = n) existuje h ∈ H, která je pro S perfektní.
    - Dá se dokázat, že taková množina pro dané univerzum vždy existuje a je rozumně velká (t). Jeden přístup je popsat jednotlivé funkce maticemi a počítat počet jejich podmatic - dokázat, že když t >= n * (ln N) * e^(n^2 / m), musí existovat perfektní systém.
    - TODO: konstrukce pomocí univerzálního hashování :-/
- Příklad perfektního systému z MIT
    - Máme danou množinu n klíčů a konstruujeme tabulku velikosti m = O(n) tak, aby FIND trval $\Theta(1)$ v nejhorším případě
    - Myšlenka - 2-levelový hašování (podobně jako u Cormacka)



    - Na levelu 2 žádné kolize (prostě se to zakáže)
        - Mějme H sadu univerzálních hashfunkcí pro tabulku velikosti m = n^2, pak použijeme-li náhodnou funkci h z H k zahašování n prvků, očekávaný počet kolizí je ½ (pro c = 1, obecně c/2)
            - Důkaz: z definice univerzality: pravděpodobnost kolize = 1/m = 1/n^2, párů klíčů je n**C**2, takže očekávaný počet kolizí = (n**C**2)*/n^2 = n*(n-1)/2n^2 < ½
                - Dá se použít Markov na omezení pravděpodobnosti
                - P[X >= t] <= E[X] / t
                - P[X = 0] = 1 - P[X >= 1] >= 1/2
        - Díky tomu rychle najdeme vhodnou funkci
        - Díky tomu má member zaručenou složitost - nejdřív lookup do primární tabulky a pak lookup na konkrétní pozici na levelu 2
    - Velikost bucketů na levelu 2 je druhá mocnina počtu prvků - to se zdá jako fakt moc, ale dohromady je to furt O(m)
        - Na levelu 1 zajistíme triviálně, m = n
        - Na levelu dva chceme, aby suma $O(n_i^2) = O(m)$
            - Důkaz?

Příklady statického perfektního hašování:

- Statické metody jsou rychlé na přístup, ale problém je s vkládáním - může trvat poněkud dlouho. V obou přístupech níže jde vidět optimalizace na MEMBER.
- Cormack
    - Primární funkce h(x) a sada sekundárních h'(i, x, r) = (x mod (2i + 100r + 1)) mod r, kde i je identifikátor funkce, r její obor hodnot a x prvek z S. Perfektní hašování kolidujících prvků zajišťují sekundární funkce.
    - Sestává ze dvou částí: adresáře a datových souborů
        - V každé buňce adresáře uchováváme ukazatel na začátek vlastního souboru a dvojici (i, r). Adresář držíme ideálně ve vnitřní paměti.
    - MEMBER(x): podíváme se do adresáře na pozici h(x). Pokud r != 0, počítáme h'(i, x, r) a výsledek je offsetem ukazatele na začátek souboru. Buďto prvek nalezneme, anebo ne.
    - INSERT(x): podíváme se do adresáře na pozici h(x). Pokud r == 0, vložím. Jinak zvedám r o 1 a najdu nekolidující i (počínaje od 0?), následkem čehož si musíme (i, r) poznamenat a přeuspořádat soubor. Pokud už je množina prvků na soubor příliš velká, buďto rozšířím, anebo přesunu jinam.
    - DELETE(x): stejně jako MEMBER(x), jen na konci odstraňujeme ze souboru. Žádné zpětné přehašovávání (již INSERT je pomalý až až).
    - Poznámka: soubory jsou tedy různorodých velikostí a max. kapacitu omezuje dostupná paměť.
- Larson-Kajla
    - Znovu máme dvě hašovací funkce a prvky ukládáme do stránek:
        - h(i, x) udává adresu stránky
        - s(i, x) udává tzv. signaturu záznamu ve stránce
    - U každé stránky máme poznamenán "separátor" (max. signaturu svých prvků)
    - MEMBER(x): iterativně zvyšujeme i (od 0?), dokud nenajdeme stránku se separátorem >= s(i, x). Až v této stránce se prvek může nacházet.
    - INSERT(x): stejně jako MEMBER(x). Pokud se prvek do stránky ještě vejde, super. Jinak:
        - Prvky ve stránce setřídíme podle signatur.
        - Prvky s největší signaturou vyjmeme.
        - Separátor stránky nastavíme na druhou nejvyšší signaturu. Pokud stále můžeme, vložíme x.
        - Pokračujeme ve vkládání, tentokráte vyjmutých prvků (příp. i s x).
    - DELETE(x): podstatně horší než u Cormacka, jestli se vůbec implementuje. Prázdné stránky mazat můžeme a nemusíme - to druhé pak může vést na fragmentaci vnitřní paměti, s níž by si však moderní správa (virtuální adresy) měla poradit.
    - Poznámky:
        - Srovnání s Cormackem: stránky jsou tedy fixní velikosti a v paměti si nemusíme udržovat jeho adresář. Na druhou stranu se každá operace potýká se zvýšeným počtem stránek, takže bychom si měli pamatovat

alespoň uspořádané pole (separátor, odkaz), a binárně v něm vyhledávat. Binární vyhledávání nám také vedlejším efektem zvýší počet použitelných stránek.
        - Problém: INSERT rekurzivně volá další, což nakonec může skončit vyhladověním zásobníku.
        - Problém: špatný vstup způsobí malé rozdíly mezi separátory stránek => špatné využití prostoru.
        - Problém: počet kolidujících prvků nesmí překročit kapacitu stránky (to by snad neměl… normální stránka bude mít alespoň 4KB).

Příklady dynamického perfektního hašování:

- Snaží se řešit problém vkládání u statických metod, a to při zachování rychlého přístupu.
- Fagin (někdy také “rozšiřitelné hašování”)
    - Znovu adresář a stránky. Adresář je natahovací/smrskávací - k identifikaci buňky tabulky užíváme takový počet dolních bitů hodnoty funkce (k), který jednoznačně rozliší všechny prvky v tabulce (jeden prvek na buňku). Hašovací funkce má tentokrát velký obor hodnot.
    - MEMBER(x): k dolních bitů h(x) nám určí buňku tabulky, kde máme odkaz do stránky a počet dolních bitů, kterými je prvek v ní obsažený rozlišitelný od všech ostatních (k2). Prvek se ale ve stránce nacházet nemusí (může se jednat o stránku jiné buňky).
    - INSERT(x): jako MEMBER(x) - pokud buňka ještě neobsahuje prvek, vložíme. Jinak musíme štěpit na dvě stránky podle dalšího bitu funkce, oběma vzniklým stránkám zvedneme k2. Když k2 > k1, zdvojnásobíme velikost adresáře, k1 := k2 a přeuspořádáváme.
    - DELETE(x): jestli se někdy implementuje, pak analogicky MEMBER/INSERT.
    - Poznámky:
        - Ke zvětšování adresáře by v praxi nemělo docházet často, takže se INSERT většinou vymezí jen na rozdělení dvou prvků do dvou buněk.
        - Adresář je reprezentován formou bitové trie.
- Litwin (jaksi “relaxované” perfektní hašování…)
    - Rozdíl oproti Faginovi je ten, že místo adresáře máme oblast přetečení, kam ukládáme kolizní prvky. Neštěpíme tedy ihned, ale periodicky po L vloženích (L koresponduje aktuální velikosti oblasti přetečení). Stránky štěpíme v pevně daném pořadí.
    - Problém: během štěpení se nám prvky hromadí v dosud nerozštěpených stránkách. Řešení - skupinové štěpení stránek:
        - Stránky štěpíme v K skupinách po G (velikost adresáře je mocnina dvojky). Po L vloženích štěpím skupiny na G+1 stránek, pomocí další hašovací funkce s oborem hodnot velikosti G+1. Nakonec K . (G+1) stránek přeorganizujeme na K1 . G stránek. Snad to není potřeba umět do detailu.

Třídění ve vnitřní paměti - Töpfer, ADS

Třídění obecně

- Formálně:
    - U je uspořádané universum
    - Vstup: posloupnost $\{a_n\}$ prvků z U
    - Výstup: neklesající posloupnost $\{b_n\}$ taková, že $\{a_i \mid i = 1...n\} = \{b_i \mid i = 1...n\}$
- Řadu algoritmů můžeme použít na jakoukoliv doménu, ale jsou tu i algoritmy se speciálními vlastnostmi.

Obecné třídící algoritmy ve vnitřní paměti:

- Předpokládají random access, aby mohly vždy přistoupit ke každému prvku
- Klasická trojice přímočarých algoritmů:
    - SelectSort - setříděný a nesetříděný úsek, najdeme nejmenší prvek nesetříděného úseku, dáme na konec setříděného (prohozením).
        - Narozdíl od ostatních není stabilní
    - InsertSort - setříděný a nesetříděný úsek, nesetříděný úsek procházíme postupně a každý prvek vkládáme na patřičné místo do setříděného úseku (spoják).
    - BubbleSort - pořád dokola procházíme celé pole prvků a vyměňujeme sousedy, dokud je porušeno uspořádání.
        - Má potenciál lépe využívat cache (když si budeme pamatovat úseky, kde jsme v poslední iteraci vyměnili nějaké sousedy). V praxi je solidní i pro větší množiny, protože se chová dobře pro “skoro setříděné” posloupnosti.
        - Varianty CocktailSort / ShakeSort - polem procházíme ve střídavém směru (znovu lepší využití cache).
    - Vlastnosti:
        - In-place - netřeba další datové struktury
        - Všechny jsou dnes zajímavé už jen z akademického hlediska a jako součást jiných algoritmů (v praxi se používají pro malá data).
        - Všechny O(N^2).
- MergeSort
    - První metoda typu rozděl a panuj (paralelizovatelná)
    - Rekurzivně data rozdělujeme na poloviny, dokud nejsou triviální. Od té doby poloviny zpětně sléváme (třídíme).
    - Vlastnosti:
        - Předtřídění posloupnosti nemá vliv na efektivitu.
        - Jediný algoritmus, který lze reálně (a v různých variantách) použít i pro třídění ve vnější paměti.
        - Pro menší posloupnosti do něj můžeme vkombinovat i jiné algoritmy, např. QuickSort.
        - Složitost: O(n log(n)). Důkaz se provádí přes master theorem.

- QuickSort
    - Druhá metoda typu rozděl a panuj (paralelizovatelná).
    - Idea stejná jako MergeSort, ale místo dělení na poloviny podle počtu vybíráme pivota (náhoda, průměr, medián), podle něhož rozdělujeme. Rekurzivně voláme QS a výsledky spojujeme (nesléváme).
    - Vlastnosti:
        - Výkon stoupá a klesá hlavně s volbou pivota…
        - Nejhorší složitost O(n^2), amortizovaná O(n log(n))
        - V praxi bývá rychlejší než jiné O(n log(n)) algoritmy
    - V základní verzi nestabilní, existuje ale stabilní varianta, i když ta už nemá tak dobré chování. Nejhorší případ představuje slabinu, protože očekáváme průměrné chování. V extrémním případě může útočník schválně posílat špatné posloupnosti, aby způsobil Denial of Service (DoS) nějaké aplikace/služby.
    - Složitost: volíme-li pivota náhodně, má QS očekávanou složitost O(n log n).
        - Mareš v zásadě říká, že při náhodné volbě pivota se trefím do skoro-mediánu v průměru v O(1), protože pravděpodobnost, že ho vyberu, je ½. Skoro-mediánem je myšleno číslo mezi první a třetím kvartilem. Tenhle argument se nachází i na Wiki.
            - Všechno začíná od střední hodnoty hladiny, ve které pivota vybíráme (počítáme očekávanou složitost algoritmu). Suma konverguje podle Cauchyho testu - dám do poměru dva následující prvky, pokud je limita poměru menší než 1, konvergence je zajištěna. Díky tomu můžu tvrdit, že skoro-medián najdu průměrně v konstantním čase (i kdyby byl brutální, ve skutečnosti je 2). Tohle vysvětlení mi absolutně nedává smysl.
        - Nejspíš by stačilo pouze využít Marešova argumentu a říct, že když budeme pivota volit opakovaně, dokud bude některá z polovin větší než ¾ velikosti hladiny, máme v nejhorším případě zajištěn logaritmický počet hladin, tedy log n při základu 4/3. Díky Marešovu argumentu můžeme výběr pivota počítat za O(1). Práce na jedné hladině je O(n), a celková složitost je tedy O(n log n) s jistou multiplikativní konstantou, kterou nám průměrný případ podstatně zredukuje.
        - Kdybychom dokázali určit, že je příští hladina už dostatečně předtříděná, mohli bychom na ní pustit třeba BubbleSort…
- HeapSort
    - Idea: postavit a postupně rozebrat 2-regulární haldu.
    - Složitost O(n log(n)) plyne přímo ze složitostí operací na haldě.
    - Algoritmus není stabilní a reálně bývá pro většinu dat pomalejší než QS.
- ArraySort
    - V C# se jedná o podmíněnou kombinaci jiných - pro sekvenci menší než 16 prvků použije InsertSort, jinak se snaží o QuickSort, a při přílišném rekurzivním zanoření přepne na MergeSort, aby zaručil O(n log(n)).

Třídění omezených celých čísel (přihrádkové třídění):

- Buď přímo celá čísla, anebo indexované jiné hodnoty.
- Vyžaduje speciální pole velikosti rozsahu dané množiny hodnot. Vstup do něj postupně na správné pozici přepisujeme a nakonec celé pole projdeme.
- Vlastnosti:
    - Složitost O(N + R) - N počet prvků, R rozsah omezení
    - Stabilní. Díky tomu se dá triviálně rozšířit na lexikografické uspořádání k-tic; stačí třídění pouštět postupně na i-té členy vstupu.

Třídění stringů

- Rekurzivní přihrádkové třídění.
- Alternativa: trie

Třídění ve vnější paměti - Töpfer, OZD, SMI

- Když se nám data nevejdou do vnitřní paměti...
- Kritickým faktorem už není počet porovnání, ale I/O - složitost vyjadřujeme vzhledem k operacím se soubory.
- Jednoduchý MergeSort:
    - Iterativní.
    - V prvním kroku rozdělíme data na dva kusy (soubory) - z každého postupně čteme jeden prvek a výslednou dvojici slijeme na výstup. Vznikne posloupnost setříděných dvojic.
    - V dalších krocích znovu rozdělujeme na kusy, ale už sléváme klasicky a na výstup píšeme pouze ten menší z prvků. Tím postupně vznikají setříděné posloupnosti délky alespoň aktuální mocniny dvojky a složitost je logaritmická.
    - Vylepšení:
        - Prvotního dělení se zbavíme, když budeme prvky střídavě zapisovat do dvou nezávislých souborů, buďto po mocninách dvojky, anebo ještě lépe při detekci konce setříděné podposloupnosti.
        - Prvotní předtřídění úseků pomocí vnitřní paměti.
- K-cestný mergesort
    - Máme-li k dispozici paměť velikosti K+1 stránek, můžeme nejprve rychle setřídit úseky velikosti O(K) stránek (podle nároků algoritmu) a zapisovat je do nezávislých souborů. Pak můžeme teoreticky postupně a rekurzivně slévat O(K) souborů do jednoho (každý soubor si vyžádá alespoň jednu stránku vnitřní paměti). Prakticky by nás však mohl zastavit OS s limitem na počet současně otevřených souborů.
        - Poznámka: HeapSort je nejvhodnější, protože díky průběžnému odebírání a přidávání prvků umí teoreticky vytvářet běhy neomezené délky. V praxi však jeho efektivitu vymezuje velikost vnitřní paměti (záběr, ve kterém lze třídit). V průměru by měl každopádně vytvářet 2x delší setříděné úseky než ostatní algoritmy. Detaily:
            - Z haldy odeberu minimum a vypíšu ho. Načtu další hodnotu z disku. Pokud je >= vypsané hodnotě, zatřídím ji do haldy. **Pokud je menší, vypíšu zbytek haldy a načtenou hodnotu vložím do kořene (nová halda).**
    - Kdybychom slévali méně souborů najednou, můžeme proces i paralelizovat do většího počtu vláken. Zároveň pak bude levnější operace porovnávání hodnot ze všech vstupů (i když to lze ještě řešit dodatečnou datovou strukturou, např. haldou). Hlavně ale nesmí HDD skákat z místa na místo… seek je potenciálně drahá operace. Jeden z určujících faktorů je, kolik má HDD hlav a cylindrů, i když v praxi nebudeme pravděpodobně mít kontrolu nad tím, kam se který soubor uloží. Měli bychom si rovněž vystačit s jedním až dvěma vlákny na jádro CPU - čím více jich použijeme, tím více nebude HDD stíhat. SSD je samozřejmě výhoda, ale taky má svoje limity.

    - Summary: jde vidět logaritmická složitost (při základu K), takže zvyšování K se vyplatí pouze v malých řádech (čím větší K, tím menšího zrychlení dosáhneme dodatečným navyšováním).

- Obecné zrychlování vnějšího třídění:
    - Paralelismus:
        - Na úrovni HW - více disků, běhy na různých discích, ...
        - Na úrovni SW - více vláken, pipelining...
        - Distribuované systémy?
    - Vybavení:
        - RAM
        - SSD
    - Komprese dat (SW)

Dolní odhady pro uspořádání (rozhodovací stromy) - ADS, DS

- Předpokládáme, že se snažíme uspořádat množinu čísel a jediné, co s nimi umíme, je vzájemné porovnání
- Složitost problému je složitost asymptoticky nejlepšího algoritmu, který danou úlohu řeší
- Chceme určit dolní odhad složitosti (značíme Ω(f(x))
- Trivi příklad: Složitost pro nalezení mediánu je Ω(n)
    - Důkaz: Intuitivně potřebuju medián, i kdyby mi spadnul z nebe, porovnat se všemi ostatními prvky, abych vůbec zjistil, jestli to je medián

Rozhodovací stromy

- Mějme třídící algoritmus A, který jako jedinou primitivní operaci pro prvky vstupní posloupnosti používá porovnání. Rozhodovací strom algoritmu A pro n-prvkové posloupnosti = binární strom, jehož vnitřní vrcholy jsou ohodnoceny porovnáními dvou prvků vstupní posloupnosti ($a_i <= a_j$, i, j = 1...n, i ≠ j), pokud platí:
    - Posloupnost porovnání při třídění posloupnosti π algoritmem A je stejná jako posloupnost porovnání při průchodu posloupnosti π stromem T
- Dolní odhad pro čas algoritmu A v nejhorším případě je délka nejdelší cesty z kořene do listu (tedy hloubka stromu), očekávaný čas je průměrná délka cesta z kořene do listu
- Dolní odhad pro třídící algoritmy založené na porovnání <u>v nejhorším případě</u>: Ω(n log n)
    - Důkaz: Sestavíme rozhodovací strom pro daný algoritmus, ten zjevně musí pro každou vstupní permutaci dopadnout jinak => listů je n!
        - Použijeme aproximaci n! >= n^(n/2)
        - Binární strom hloubky k má nejvýše 2^k listů => Binární strom s L listy má hloubku alespoň $\log_2 L$
        - Čili: $\log_2 n! >= \log_2 n^{(n/2)} = n/2 * \log_2 n = \Omega(n \log n)$ porovnání
- Dolní odhad pro třídící algoritmy založené na porovnání v <u>průměrném případě</u>: Ω(n log n)
    - Idea důkazu: Pokud má strom T vrchol, který má jen jednoho syna, tento vrchol vypustíme, počet listů se tím nezmění, tedy můžeme uvažovat úplný binární strom s n! listy
    - Důkaz: Tady je to horší, půjdeme na to přes součet délek všech cest z kořene do nějakého listu, ten označme B(T) pro strom T. Dále označme A(n) průměrný počet porovnání algoritmu, který dostane n-prvkový vstup. Chceme dokázat, že A(n) = Ω(n log n)
        - Zavedeme b(k) = min {B(T) | T je binární strom s k listy}
        - A(n) >= b(n!) / n!, potřebujeme odhadnou b(n!) alespoň jako $k \log_2 k$, pak by platilo $A(n) >= b(n!) / n! >= (n! \log_2 n!) / n! = \log_2 n! = \Omega(n \log n)$ [poslední rovnost máme dokázanou z předchozí věty o nejhorším případě]
        - $b(k) >= k \log_2 k$, indukcí:
            - $b(1) = 0 = 1 \log_2 1$, $b(2) = 2 = 2 \log_2 2$
            - chceme $b(i) >= i \log_2 i$ --- na to už fakt nemam

- Pozn.: existují rychlejší algoritmy, ale ty používají “něco navíc” - například přihrádkové třídění, lze vytvářet i hybridní verze, ty můžou mít očekávaně i O(n)

Relaxované vyhledávací stromy - WTF

- Idea: oddělit operace na stromě a vyvažování, vyvažovat budeme jen když je to potřeba
    - Místo provedení vyvažování se požadavek na vyvážení zaznamená do fronty požadavků
    - Obecný koncept relaxovaných stromů (přesná definice není?)
- Potřebujeme řešit:
    - Aby nám čas operací nestoupl na O(n)
    - Aby bylo možné dodatečná vyvážení udělat bez nutnosti stavět strom od začátku
- Aplikace
    - Dávkové zpracování - requesty na operace se stromem nám choděj koncentrovaně v nějaký časový úsek, pak v mezičase můžeme provést rebalancing
    - Paralelní zpracování - díky stromové struktuře se nám procesy standardně mohou prát o operace nad uzlama při rebalancování, přestože update sám provádějí jinde. Když rebalancing odložíme a necháme na jednom konkrétním procesu, bude zamykání lokální.

Příklad relaxovaného modelu (nad ČČ stromy, OMG)

- Máme ČČ strom (podle definice, jen relaxujeme požadavky na vlastnosti vrcholů, takže efektivně jediná vlastnost je, že vrchol je buď černý nebo červený)
- Nad daty současně pracuje více procesů, které jsou dvou typů
    - Uživatelský - provádí pouze vyhledávání, přidávání a ubírání prvků a když po aktualizaci vznikne požadavek na vyvažování, přidá jej do fronty (společně s vrcholem, který tuto podmínku porušil; sám nevyvažuje)
    - Správcovský - bere vhodné požadavky z fronty a provádí je; každý požadavek buď úplně ošetří, nebo ho transformuje v jiný požadavek bližší ke kořeni stromu
- K tomu máme frontu vyvažovacích požadavků
    - Pokud je prázdná, pak je strom vyvážený ČČ strom
    - S každým požadavkem je asociován vrchol, který porušil podmínku ČČ stromů, vrchol si pamatuje odkazy na své požadavky
    - Požadavky jsou dvou typů - b a v
        - Požadavek b znamená, že vrchol je červený a v okamžiku vznesení požadavku má červeného otce (to se později může změnit)
        - Požadavek v znamená, že v je černý a na cestě z kořene chybí jeden černý vrchol
        - Je-li požadavek typu b, pak už neexistuje žádný další požadavek na tento vrchol (tzn. požadavky na jeden vrchol jsou vždy ve stylu: vvvvb)
- Operace prováděné uživatelským procesem
    - MEMBER klasickej
    - INSERT vloží prvek (z listu *t* se stane vnitřní vrchol a přidají se mu dva černí synové), dále pak:

        - Pokud t má ve frontě požadavek v, pak se jeden smaže a vrchol zůstane černý
        - Pokud t nemá žádný požadavek, bude t červený (pokud je i otec červený, vytvoří to požadavek b)
    - DELETE hmm
- Správcovský proces dělá dokola následující věci
    - Jakýkoliv požadavek na kořen stromu je zrušen
        - b nedává smysl, otec nemůže být červený, když žádný není
        - v taky nedává smysl, do kořene žádná cesta nevede
    - Jakýkoliv požadavek typu b na vrchol, jehož otec je černý je zrušen
        - požadavky b znamenají poruchu, tedy červený synem červeného, tahle situace znamená, že už to někdo napravil
    - Sloučí v požadavky bratrů a předá je otci (pokud je červený, tak ho navíc přebarví na černo a jeden požadavek rovnou zruší), samozřejmě tomu z bratrů, který měl požadavků víc, nějaké zbydou
    - PROBOHA těch operací je asi milión!!! :-(:-(
    - Dál se tam pak dělají nějaký rotace, apod.
- Aby se procesy navzájem nemlátily, musí si zamykat vrcholy
- Proč je to korektní?

Analýza, návrh a management softwarových systémů (12/19)

[OK] Životní cyklus SW systémů

Zdroje:

- *Úvod do SWI (přednáška 3?)*

Fáze životního cyklu (**fáze tvorby**) softwaru na zakázku:

1. **Nápad**
2. Nezávazná **vize** či **upoutávka**
    a. obchodnická část, kde lanaříme nějakého finálního odběratele
3. Závaznější **úvodní studie** (včetně **katalogu požadavků**)
    a. definice problému, pojmů a procesů a jejich základní analýza, přehled o obecných nárocích a požadavcích, sebrání všech argumentů pro a vyvrácení argumentů proti
    b. příp. deklarace obecných garancí z naší strany (hlavně po implementaci/dodání)
4. Detailní **specifikace** problému, konečného stavu a našich garancí (**rozšíření úvodní studie**)
    a. detailní analýza problému (souvisejících entit, procesů a požadavků) - jinými slovy, “odkud a kam se chceme dostat + náležitosti”
    b. příp. přizpůsobena přímo zákazníkovi a potom by měla vznikat za jeho součinnosti, resp. výhradně se znalostí jeho vnitřních záležitostí
5. **Návrh řešení**
    a. detailní navržení logických celků a jejich vlastností tak, aby v souladu se specifikací propojily začátek a konec problému… v podstatě “jak nejlépe splnit specifikaci vzhledem ke stanoveným nárokům a požadavkům”
    b. ideálně včetně vysvětlení jednotlivých rozhodnutí a nastínění alternativ
    c. případně přizpůsoben přímo zákazníkovi - potom by měl vznikat za jeho součinnosti a výhradně se znalostí jeho vnitřních záležitostí
6. **Implementace řešení**
    a. po jednotlivých celcích - plán bychom měli rozvrhnout tak, abychom mohli každý celek co nejdříve otestovat (na základní úrovni)... musíme co nejdříve identifikovat nesoulad mezi specifikací/návrhem a technologií/realitou!
7. **Akceptační testy**
    a. testování díla jako jednoho celku - vyhodnocujeme jeho vlastnosti a porovnáváme se specifikací, případně ještě optimalizujeme a provádíme drobné úpravy
8. **Dodání a instalace**
    a. příp. včetně HW a lidí
9. **Údržba a služby**
    a. až na výjimečné okolnosti nesmíme zapomenout nechat si je zaplatit...
10. Retirement (někdy také “phase-out”)
    a. teoreticky lze chápat i jako součást bodu 9

b. zajištění systematického ničení našeho díla (či pomoci v této činnosti), resp. migrace do systému někoho jiného (a nikoliv nutně lepšího) - podívejme se např. na to, jak to koulí státní správa s periodickou veřejnou soutěží a následně státními zakázkami…
c. může zahrnovat i odvoz HW a jiné věci

Návrh a implementace řešení se může někdy drobněji rozdělovat na dekompozici do komponent, řešení komponent, implementace komponent, testování komponent a integraci komponent do celku.

Smlouva by se měla sepisovat nejpozději do začátku implementace (ideálně po sestrojení a schválení specifikace), takže nejpozději první verze návrhu řešení by měly rámcově odhadnout investici a zisk (business model), včetně případné dlouhodobé údržby… ostatně, nemá smysl se pouštět do obchodu, když se na ceně nedohodneme ani rámcově...

Jednotlivé fáze, jejich přesná podoba a návaznost jsou dány okolnostmi, hlavně modelem životního cyklu. Ideální poměr analýzy, návrhu a implementace co do rozložení úsilí by měl údajně být 40-40-20. Obecněji řečeno MUSÍME dbát hlavně na analýzu, specifikaci a návrh problému - jinak si koledujeme o pěkné maléry (když už je podepsaná smlouva).

Je vhodné být před zákazníkem co nejvíce napřed, ale nesmíme to přehánět… zákazník může teoreticky kdykoli měnit specifikaci, přidávat požadavky a v extrémních případech dokonce i samotný problém (třeba když vyplyne, že obchodníci a manažeři vlastně pořádně nevědí, co se pod nimi děje - to je mimochodem specialita státní správy).

Metodiky vývoje SW (nebo také “modely životního cyklu”):

- V zásadě otázka, jak k jednotlivým fázím přistoupíme...
- Waterfall (vodopád).
  - Idea: vyvíjet lineárně po fázích a nevracet se do předchozích fází.
  - Metodika pro ideální případy, kdy všichni vědí, co dělají. Spoléhá na to, že předchozí kroky jsou perfektní nebo neměnné vůči externím vlivům (např. zákazníkem - viz výše).
- Pioneer (průkopník).
  - Idea: vyvíjet lineárně po fázích podle aktuální situace, hladově (na bázi best-effort). Počítáme s návratem kamkoliv zpět a přepracováváním.
  - Pravděpodobně vhodné pro případy, kdy máme málo času a hodně všeobecné nejistoty.
- Iterative (přírůstky).
  - Idea: po dekompozici provádíme návrh a implementaci komponent nezávisle na ostatních. Jedna komponenta tedy může být hotová, než začneme navrhovat druhou.
  - V praxi se jedná o kompromis mezi vodopádem a spirálou.
- Spiral (spirála).

- Idea: vyhrocení přírůstků. Jednotlivé komponenty i testujeme a provozujeme, než začneme vyvíjet jiné.
- V praxi to vypadá na flexibilní přístup, u kterého se však může vyskytovat nesoulad mezi plánováním a reálnou situací.

[Víceméně OK] Základní metriky pro životní cyklus SW

- Metrika = funkce vydávající čísla, která o něčem svědčí.
- Metrikami se snažíme ohodnotit nějaký atribut SW, jeho dokumentace či jejich části. V praxi není snadné výsledné indikátory správně změřit. Atributy jsou vlastně interní vlastnosti SW (např. jednotlivé ukazatele kvality), které se nakonec promítají do externích a obecných vlastností SW (např. kvalita či použitelnost).
- Obecněji metriky dělíme do dvou kategorií:
    - Statické - atributy nezávislé na spuštění (např. kvalita kódu).
    - Dynamické - atributy závislé na spuštění (např. účinnost a spolehlivost).
- Typy užití (nejen po implementaci, ale i v rámci řízení softwarového procesu):
    - Ohodnocení kvality SW a jeho následné zlepšování. Identifikace anomálií, především složitých částí SW, náchylných na problémy.
    - Srovnání s jiným SW či předchozími verzemi (statistika).
    - Predikce spojená se SW - některé metriky mohou na aktuálně zkoumanou působit a např. změření výkonu nám může dát vhled do výkonu na jiných strojích (predikce).
- Problémy:
    - Obecná absence standardů. Několik jich přesto existuje, ale zastarávají, postrádají flexibilitu, nejsou udržované a v praxi je zapotřebí specialistů.
    - Vedoucí projektů je víceméně nepotřebují a manažeři je neumějí systematicky využívat, ba je dokonce zneužívají proti vedoucím?
    - Zvýšení nákladů: nevíme, jestli se proces měření nakonec vyplatí (z pohledu investic). Navíc mají metriky tendenci měnit zažité SW procesy a vnášet do projektu nová rizika (jak se říká, “ignorance is bliss”).

Základní metriky životního cyklu SW (o tom tahle otázka primárně je):

- Time
    - Jak dlouho co trvalo nebo jak dlouho na co potřebujeme.
- Size
    - Velikost kódu, počtu komponent, atd.
- Effort
    - Kolik úsilí si co vzalo anebo vezme - základní jednotka jsou asi člověko-hodiny.
- Quality
    - Indikátory kvality našeho díla, takže výkonnost, kvalita kódu a dokumentace, atd.

Některé jsou důležité pro řízení či výslednou kvalitu a v praxi velmi často opomíjené.

Vymýšlení metrik:

- Musíme dávat pozor na celý proces měření - může se stát, že bude metrika správně definovaná a měřitelná, ale bude měřit něco trochu jiného, anebo nepřesně.
- Obecně by ve standardu nemělo chybět (“Metric evaluation dialogue”):
    - Purpose (Co je účelem metriky)
    - Scope (Co do metriky zahrnout a co ne)
    - Attribute (Co se snažíme změřit)

- Natural scale (Rozsah výsledného číselného indikátoru)
- Variability (Proměnlivost, neboli rozptyl indikátoru)
- Metric + instrument (Funkce metriky a jak/čím hodnotu změřit)
- Side effects (Může mít průběh měření vliv na jiné metriky, ba dokonce tuto?)
- Další:
  - Diskuse vztahu atributu a metriky…
  - Diskuse validace a verifikace vstupních i výstupních dat.
  - Stanovení periody a přesnosti pro zpracování výstupních data zpracovávat.

[OK] Volba cílů, smlouva a principy vyjednávání

Zdroje:

- *Pokročilé aspekty SWI (přednáška 11?)*

Tady k tomu nikdo nic nenapsal, tak se pokusím něco stvořit…

1) Stručný popis životního cyklu.
2) Smlouva by se měla sepisovat nejpozději do začátku implementace (ideálně po sestrojení a schválení specifikace), takže nejpozději první verze návrhu řešení by měly rámcově odhadnout investici a zisk (business model), včetně případné dlouhodobé údržby…

Nemá smysl se pouštět do obchodu, když se na ceně nedohodneme ani rámcově :). **Ještě předtím musíme pečlivě volit cíle a požadavky ve vztahu s tím, pro koho systém děláme (jaké jsou jeho preference)**. I když bychom si mohli přát jistou volnost a nakonec být schopni produkt prodat i někomu jinému, jestli nám ho platí konkrétní zákazník, má právo na jistý service a šití na míru. Princip vyjednávání se od toho odvíjí (náš zákazník, náš pán). V praxi může kdykoli měnit specifikaci, přidávat požadavky a v extrémních případech dokonce i měnit samotný problém (třeba když vyplyne, že obchodníci a manažeři vlastně pořádně nevědí, co se pod nimi děje - to je mimochodem specialita státní správy).

Když si to člověk rozmyslí, je vhodné být před zákazníkem ve všem možném co nejvíce napřed (ale nesmíme to přehánět). Třeba:

- Vyvíjet specifikaci, příp. i řešení, ve spolupráci s techniky zákazníka. Snažit se vyhovět potřebám zákazníka, ale neslibovat hory doly - neopouštět realitu a pokusit se před sepsáním smlouvy ještě otestovat hlavní předpoklady specifikace (či brzkého návrhu řešení).
- Do rozpočtu a časového rozvrhu zahrnout dostatečné rezervy. Nikomu neprospěje, když cenu podceníme (nedej Bože schválně) a nakonec nedodáme nic nebo něco, co se “ničemu” docela podobá. Tohle je mimochodem specialita státní správy, resp. státních zakázek nastavených tak, aby prakticky vzato vyhrála nabídka s nejnižší cenou…
- Mít na obou stranách alespoň jednu či dvě osoby, které zařídí vzájemnou komunikaci. Zároveň by mohli zákazníka předem dloubat do boku, že je potřeba např. zařídit prostor pro dodaný HW.

Po extenzivním vyjednávání a plánování by z nás ideálně měla spadnout většina nejistoty ohledně úspěšnosti projektu. Rizika budou existovat snad vždy a nevyhneme se jim, ale jak se říká… těžko na cvičišti, lehko na bojišti. Měli bychom mít jasně stanoveno, jaké máme zdroje a jak se je pokusíme vynaložit, abychom projekt dovedli do zdárného konce. Součástí je samozřejmě harmonogram a sousled jednotlivých procesů či úloh. Techniky zobrazení třeba:

- Rozvrh úloh a zdrojů (jednoduše graf těchto entit).
- Ganttovy diagramy (sloupcové grafy) - horizont je čas a vertikála množinou úloh (mohou na sebe navazovat, ale nemusí).

Cenu je dobré odhadovat na základě:

- Předchozí zkušenosti.
- Dekompozice či plánování (třeba právě Ganttových diagramů).
- Rozsahu či úsilí - řádky kódu (COCOMO), člověko-dny a tak.

Co se týče smlouvy, třeba:

- Snažme se vyjednat podmínky, které omezují zpětné zásahy zákazníka do specifikace, anebo ho alespoň nutí platit náklady s tím spojené, resp. prodlužovat termín dodání.
- Zákazník nás zase bude chtít hnát k odpovědnosti za nesplnění specifikace, prošvihnuté termíny, atd. Musíme mu vyhovět, ale o to více úsilí, dobré práce a analýzy musíme odvést.
- Ideálně si stanovme, že nám jistí lidé od zákazníka budou pravidelně k dispozici (třeba schůze).

[OK] Zjišťování a analýza požadavků na softwarový systém

Zdroje:

- *Pokročilé aspekty SWI (přednáška 2 a 11?)*
- *Úvod do SWI (přednáška 4?)*

Potřebujeme vědět, jaké vlastnosti má náš systém mít ještě předtím, než ho začneme vyvíjet. Sběr, analýza a zpracování požadavků má své fáze:

- Nejdříve se scházíme, jednáme a zapisujeme si...
- Pak analyzujeme...
- Poté specifikujeme...
- A nakonec verifikujeme (zpětná vazba od stakeholderů)...

Požadavky mohou mít mnoho forem, minimálně např.:

- Požadavky na funkcionalitu či specifické chování
- Požadavky na specifické rozhraní komponent
- Požadavky na kvalitu (výkon, bezpečnost, atd.)
- Požadavek na soulad s legislativou (ten je tak nějak implicitní, ale musí být analyzován!)

Všechny požadavky jsou nakonec shrnuty ve specifikaci, která by měla být:

- Správná, úplná, jednoznačná (konzistentní), ověřitelná

Neboť specifikace je závazná :). Důkladnost je zásadní - chybějící požadavky v důsledku odbytého sběru požadavků jsou primární riziko neúspěchu. Od požadavků se navíc mnohdy odvíjí cena projektu. Je vhodné si také seřadit kvalitativní požadavky podle priority (na nízké úrovni mohou dokonce být ve vzájemném sporu - např. bezpečnost a použitelnost).

V praxi se hodí umět dopátrat, proč zahrnout ten které požadavek, resp. co vedlo k čemu. Nejlépe mapovat na případy užití (“Requirements traceability matrix”), z nichž požadavky vlastně primárně vycházejí.

[Víceméně OK] Popis, návrh a modelování architektury softwarového systému.

Zdroje:

- *Architektury SW systémů - kapitola "pohledy"*

Architektura SW = struktura struktur SW, sestávající z popisu elementů, jejich externích vlastností a vztahů mezi nimi. (Bass, Clemens, Kazman)

Architektura

- Bass & Clemens & Kazman
    - Module viewpoint - struktura systému z hlediska kódu
        - Concerny: Responsibility modulů, usage, vztahy modulů
        - Různé typy
            - Decomposition (moduly, package, rozděl a panuj)
                - Důležité dokumentovat rozhraní modulů
                - Posuzování modifiability
                - Dělba práce
            - Usage
                - "uses" relationship
                - Pomůže identifikovat závislosti
                - Plán vývoje (co musí být dřív)
            - Generalization (Class)
                - Identifikace abstrakcí
                - Není jen dědičnost ve smyslu OO - můžeme zdědit vlastnosti které nejsou vyjádřeny kódem
    - Component and connector - runtime entity a jejich interakce a chování
        - Concerny: Identifikace a interakce komponent, sdílená data, repliky uvnitř systému, datová cesta, paralelizace, dynamická struktura systému
        - Typy
            - Process - thread, distribuovaný proces…
                - Vztahy jsou komunikace, synchronizace, paralelismus, závislosti
            - Data flow - trasování dat skrze systém
                - Datová integrita, výkon
                - Identifikace datových závislostí komponent
    - Allocation - přiřazení nesoftwarovým strukturám
        - Deployment view - klasický deployment diagram
        - Implementation view
        - Work assignment view

Popis

Nakonec je to vlastně kolekce dokumentů. Podle prominentního standardu IEEE 1471 jsou na systém kladeny určité zájmy (stakeholders) a:

- Každý zájem je charakterizován architektonickými pohledy.
- Architektonický pohled je charakterizován architektonickými hledisky.
- Architektonické hledisko je popsáno sadou modelů ve standardní notaci (např. UML).

Popis architektury je kompletní ⇔ když pokrývá všechny zájmy.
Jeden model nemusí příslušet jen jednomu hledisku/pohledu. Zřejmě tedy popisují elementy struktur celého systému. Pohledy/hlediska jsou poněkud abstraktní a existuje vícero názorů na to, co to vlastně může být. Příklady názorů:

- Bass, Clemens, Kazman
    - **3 druhy hledisek: module, component-and-connector, allocation.**
    - Modulární hledisko staticky popisuje logickou strukturu užívanou převážně ve vývoji.
        - v podstatě komponenty SW a jejich dekompozice + závislosti funkční i strukturální
        - dokumentovat rozhraní!
    - CnC hledisko dynamicky popisuje chování a interakci logické struktury systému.
        - třeba Data-Flow a Process diagramy...
    - Allocation hledisko popisuje konečnou fyzickou architekturu a elementům přiřazuje zdroje.
- Kruchten
    - **4 druhy pohledů: logický, vývojový, procesní a fyzický.**
    - Logický pohled mapuje logickou strukturu na strukturu funkcionality, a klíčové chování.
    - Procesní pohled mapuje funkcionalitu na procesy (sady úloh) a popisuje jejich chování či interakci (třeba sekvenční diagramy, diagramy aktivity).
    - Vývojový pohled popisuje dekompozici do celků vyvíjených nezávislými týmy.
    - Fyzický pohled je opět přiřazením funkčních komponent na fyzické konfigurace. Měl by být nezávislý na vývojovém pohledu.
    - Průmětem všech 4 pohledů jsou případy užití pro stakeholdery (“Scenarios”).

Návrh
Odvíjí se od specifikace a je nadmnožinou popisu architektury. Měl by dokumentovat i naše volby při navrhování a jaké jsou alternativy. Výstupem je tedy sada dokumentů a modelů - modely typicky vytváříme ve specializovaném SW (e.g. Enterprise Architect). Lepší nástroje umí dokonce generovat kód.
Během navrhování je vhodné:

- Užívat operace/styly (viz další státnicové téma), resp. dobře známé konstrukce/technologie. Potřebujeme-li popsat dynamiku či mobilitu systému, lépe užívat místo UML formální metody specifikací a modelování (třeba ADL).
- Navrhovat iterativně a zkoumat způsoby, jak návrh ještě vylepšit.

Modelování

Prostředky: UML (nejužívanější), ADL.

ADL (“Architecture Description Language”)

- Definice: ADL je jazyk pro popis softwarové, technické nebo systémové architektury. Není to jednoznačná definice a jednotlivé architektury se překrývají.
- Od ADL vyžadujeme formální popis architektury, která má být zpracovávána jak lidmi tak i stroji. Grafická forma výhodou.
- Formální ADL: založeny na Petriho sítích (např. NOAM), temporální logice či procesní algebře (Wright).
- Musí umět vyjadřovat základní věci: komponenty, konektory a konfigurace, příp. akce systému, jejich chování a vztahy mezi nimi.
    - Popisuje tedy nejen statické architektury, ale i dynamické (komponenty vznikají a zanikají) či mobilní (komponenty se stěhují). Jinými slovy, vhodnější pro různé automaticky spravované cloudy, resp. instance nějakého SW. PaaS, SaaS, apod.?
    - Konfigurace se mohou dále dělit na:
        - Implicitní - je jednoznačně určena komponentami a konektory (na jeden konektor se může připojit pouze jedna jiná komponenta)
        - In-line - je určena až rolemi komponent (na jeden konektor se může připojit více komponent a konfiguraci určuje např. protokol, dynamicky)
        - Explicitní - konfigurace určena opravdu zcela explicitně

ACME

- Někdy také “univerzální ADL” - ostatně je to ADL velmi podobné a dokonce se to využívá pro převod mezi některými ADL.
- 7 architektonických konstruktů (**základní**)
    - **Components**
    - **Connectors**
    - **Systems**
    - Ports
    - Roles
    - Representations
    - Rep-maps
- Rozšiřitelný pomocí systému anotací
- Typový mechanismus
- Obsahuje sémantický framework podporující logické uvažování o architektonických popisech ?
- Příklad:

```
System simple_cs = {
        Component Client = { Port send-request; };
        Component Server = { Port receive-request; };
        Connector RPC = { Roles { caller, callee}};
        Attachments {
```

client.send-request to RPC.caller;
server.receive-request to RPC.callee;
}
}

- Vlastnosti v ACME
  - Nejsou interpretované (slouží spíš jako anotace)
  - Název, hodnota, typ (jazyk vlastnosti)
  - Properties můžeme interpretovat externě - generace kódu, simulace
- Reprezentace v ACME
  - Detail / hierarchický rozklad komponenty nebo konektoru
  - Několik reprezentací (různá abstrakce, způsoby realizace)
- Omezení v ACME
  - Podmínka kterou musí architektura splňovat
  - Sada logických konstruktů (exists, = , !=, atp.)

Další ADL

- Wright
  - Na procesní algebře
  - Komponenty, konektory, konfigurace
  - Příklad:

System SimpleExample
component Server =
port provide = [provide protocol]
computation =
component Client =
port request = [request protocol]
computation =
connector C-S-connector =
role client = [client protocol]
role server = [server protocol]
glue = [glue protocol]
Instances
s: Server
c: Client
cs: C-S-connector
Attachments
s.provide as cs.server
c.request as cs.client

  - Po konektorech obecně chceme
    - Možnost vyjádřit architekturní interakce (pipes, eventy, volání)
    - Dynamické interakce (inicializaci konektoru)
    - Odlišení různých typů konektorů
    - Formální model

    - Wright postaven na CSP (Communicating Sequential Processes)
        - CSP má užitečné vlastnosti - interní a externí volba, paralelní kompozice, existující tooling
        - e?x (vstupní) e!x (výstupní) data
        - Prefixy e → P
        - interní / externí volba (| a čtvereček)
        - Paralelní kompozice P || Q- spojená interakce pro události v průniku možností P a Q
        - Propletenec pro úspěšný konec
        - Scope procesů v rozsahu
    - Wright komponenta je struktura s porty (CSP události) a sémantikou (computation, CSP procesy)
    - Konektory - propojené CSP procesy
        - Role - povinnosti účastníku
        - Glue - koordinace činností rolí
- Petriho sítě
    - Grafický jazyk, primárně určený na modelování souběžnosti a sdílení zdrojů
    - Šestice (S, T, F M_0, W, K)
        - S = množina míst (misky)
        - T = množina přechodů
        - F = orientované hrany mezi místy a přechody
        - M_0 = počáteční ohodnocení misek
        - W definuje váhy hran kladným číslem vyjadřuje konzumaci a generování
        - K = kapacitní omezení misek
    - Stav M = vektor počtu tokenů v místech
    - Firing sequence = seznam dvojic (M_i, t_i), t_i uskutečnitelný ve stavu M_i
    - Přechod je uskutečnitelný pokud máme dostatek zdrojů
    - Seznam stavů vzniklý projekcí spouštěcí sekvence na první složku je trajektorie
    - Krok výpočtu má libovolné (i nulové) množství uskutečněných přechodů
    - Modelování distribuovaných systémů
    - Petriho sítě samy o sobě nejsou ADL, dají se ale použít jako základ pro modelování DS
        - Komponenty ADL udržující stav jsou misky, operační komponenty jsou přechody
        - Konektory ADL jsou hrany
        - Konfigurace ADL je dána výčtem aktivních hran
    - Problémy - spojujeme jen různé typy komponent, tedy nemůžeme řetězit operace -> pomocné triviální komponenty
    - Absence rozhraní - jen jeden typ teček
        - Obarvené Petriho sítě řeší jen u míst (rozhraní pořád netypované)
        - Je potřeba nějaký další formalismus
    - NOAM
        - Petri + Hierarchický objektový přístup

        - Systém = síť specifikací modulů (komponent), pospojovaných komunikačními kanály (konektory)
        - Specifikace modulu = komponenta + chování, semantika operací, závislosti, atp.
- TOGAF = The Open Group Architecture Framework
    - Na bázi pohledů - tvorba architektury
    - 3 architektury, 4 domény
        - Byznys
        - Informačních systémů - Datová a aplikační doména
        - Technlogická
    - Byznys modelování - BPMN, Use case, Class models, další.
    - Archimate = grafický jazyk pro TOGAF
        - Na bázi UML
        - Má metamodel
        - 18 pohledů - všechny s metamodelem
        - Funkcionální pohled - run-time pohled na systém, zodpovědnosti, rozhraní a interakce
            - UML komponenty a chování + stereotypy
            - Hierarchický model
            - Subsystémy a jejich elementy
        - Informační pohled - správa, uložení a distribuce dat
            - UML CD
            - Tok dat
            - Lifecycle významných objektů
- ISO 42010
    - Standard pro popis hotové architektury
    - Metamodel popisu architektury
    - 1 architektura, více pohledů na ni
    - Architektura = konstrukce i vlastnosti
    - Popis architektury odráží zájmy (concerns)
    - Popis má více pohledů
    - Pohled má 1+ modelů
    - Popis obsahuje zdůvodnění

[OK] Styly softwarových architektur.

Zdroje:

- *Architektury SW systémů - kapitola “architektonické styly”*
- *Pokročilé aspekty SWI - přednáška 3, stránka 15?*

Při návrhu architektury se snažíme dodržovat základní principy:

- Oddělení zájmů (separation of concerns)
- Vysoká soudržnost / malá provázanost (high cohesion / low coupling).
- Oddělení modelu, pohledů a řízení (MVC).

K dosažení těchto principů užíváme architektonické operace a styly…

- V praxi musíme při navrhování uplatnit spíše sílu kompromisu a dosažení vytyčených výsledků. Operace/styly jsou hezké, ale konečné řešení z nich v pokročilých případech snad nikdy nesestává… Navíc, kdo by navrhoval specifické architektury takto “exaktně”?

Architektonické operace:

- Kodifikují návrh architektury.
    - Kroky, kterými lze během návrhu odvozovat styly, vzory a referenční architektury.
    - Samy o sobě nic neřeší, pouze vylepšují návrh.
- Příklady:
    - Abstrakce
        - V praxi abstrakcí vznikají např. vrstvené systémy a obecná rozhraní k heterogenní množině implementací. Má tedy vytvářet virtuální komponenty, jejichž funkcí je řídit či simulovat jiné.
    - Komprese
        - Spojení komponent do jediné (silní požírají slabé).
        - Cílem je převážně zjednodušení, nakonec pak také urychlení vývoje nebo dokonce systému jako takového (na to bych být vámi příliš nesázel…).
    - Dekompozice
        - Opak komprese: rozdělení jedné komponenty na menší části.
        - Cílem je převážně znovupoužitelnost, rozšiřitelnost a srozumitelnost.
    - Generalizace
        - Subkomponenty jsou specializací funkcionality rodičovské komponenty.
        - Cílem je podle slidů znovupoužitelnost, ale podle mě spíše rozšiřitelnost, robustnost a udržitelnost :). Ostatně, vlastně se jedná o speciální případ abstrakce.
    - Replikace
        - Komponenty v návrhu replikujeme.
        - Cílem je převážně spolehlivost a výkonnost.
    - Sdílení zdrojů
        - Zapouzdření dat či služeb (zase speciální případ abstrakce?).
        - Cílem je podle slidů integrovatelnost, přenositelnost, modifikovatelnost. Podle mě převážně interoperabilita a udržitelnost.

Architektonické styly:

- Styl = popis typů komponent a vzorů jejich chování za běhu či přenosu dat.
    - V podstatě abstraktní modelové řešení (třída architektur) určité třídy problémů. Jednotlivé styly se mohou překrývat či vzájemně vylučovat a systémy je mnohdy kombinují. Jsou dobře známé a dokumentované.
- Styl je definován (inu, moc toho není…):
    - množinou typů komponent (např. proces nebo objekt) a jejich topologií,
    - množinou komunikačních mechanismů (mezi komponentami),
    - množinou sémantických omezení komponent (např. nesmíme měnit určité hodnoty)
- Příklady stylů:
    - Systémy řízené událostmi
    - Dávkové/sekvenční datové toky
    - Centralizovaná data
        - cíl: interoperabilita, integrovatelnost, škálovatelnost
    - Virtuální stroj (interpreter)
        - cíl: zajištění přenositelnosti něčeho jiného či prototypování
    - Call and wait
        - cíl: znovupoužitelnost
- Styly se liší v několika hlavních konceptech (v podstatě vyplývá už z definice…):
    - Typy užitých komponent a vazeb mezi nimi.
    - Způsobem sdílení, alokace a předávání řízení mezi komponentami.
    - Způsobem výměny dat a interakce.
- Heterogenní systémy
    - Kombinují styly. Jsou 3 způsoby integrace stylů:
        - Orientované na lokalitu - nepřekrývají se.
        - Překrývající se - odlišné a vzájemně kompatibilní styly, pro dané komponenty.
        - Hierarchické - komponenta jednoho stylu je interně řešena jiným stylem.

[OK] Hodnocení kvality, integrace a znovupoužitelnost architektury.

Zdroje:

- *Architektury SW systémů - kapitola "kvalita"*

Kvalitu je dána splněním funkčních i nefunkčních požadavků:

- S funkčními požadavky je to jednoduché: buď jsou splněny, anebo ne.
- S nefunkčními požadavky (bezpečnost, spolehlivost, atd.) je to složitější.

Obecně nefunkční požadavky (také kvalitativní požadavky či atributy) dělíme do následujících kategorií:

- Kvalita architektury.
- Kvalita systému.
- Kvalita pro obchodníky.

Kvalitu do značné míry určuje již architektura, resp. mapování funkcionality na softwarové struktury. Jak tedy hodnotíme kvalitu architektury? Jednoduše si vytyčíme atributy, především:

- Korektnost
    - Hlavně vnitřní integritu a fakt, že architektura popisuje řešení problému ze zadání.
- Úplnost
    - Teoretické splnění funkčních i nefunkčních požadavků, případně ověření taktik či strategií kvality (preferencí).
- Upravitelnost
    - V zásadě nás zajímá, jak moc je architektura optimalizovaná a na co, jakou má míru soudržnosti a provázanosti. Poté je tu celá plejáda drobných ukazatelů - viz níže.
- Znovupoužitelnost.
    - Hlavní ukazatele jsou samozřejmě dva:
        - Je řešený problém obecný či specifický? Jaké funkce architektura nabízí, co po nás předpokládá a jaké má vedlejší efekty?
    - Více jich však plyne z upravitelnosti (flexibility):
        - Lze vhodnou malou úpravou architekturu přizpůsobit i k něčemu trochu jinému?
        - Jak kompaktní je architektura (ne vždy na to máme rozpočet nebo stroje)?
        - Jaká je cena potřebného HW či SW (někdo potřebuje high-end a někdo ne). Kdybychom provedli upgrade nebo downgrade, bude architektura stále použitelná?
        - Jak bezpečná je architektura proti vnitřním/vnějším vlivům?
- V moderní době již také bezpečnost, dostupnost a škálovatelnost? :)

Abstraktně bychom mohli říci, že možnost integrace architektury do jiné závisí hlavně na obecnosti či specifičnosti řešeného problému, celkové míře nezávislosti architektury a její

upravitelnosti. V podstatě je to stejný příběh jako výše. Např. bychom se mohli pokusit jí odebrat databázi a předložit jí jinou (v rámci jiné architektury). Záleží, jaké na ní máme požadavky :).

Hodnocení kvality SW:

- Stěžejní bývají: dostupnost, spolehlivost, použitelnost, výkonnost, bezpečnost, přenosnost
- Příklady dalších: testability, scalability, integrability, reusability, discoverability, composability
- Typicky jsou jednotlivé atributy dávány do vztahu s konkrétními situacemi či aspekty - např. se nám replikací dat zvýší spolehlivost.
- Dynamické atributy měříme pomocí stimulů a reakcí (např. výkon a doba odezvy).

Dostupnost

- Pravděpodobnost, že systém bude v daném okamžiku provozuschopný.
    - *= mean time to failure / (mean time to failure + mean time to repair)*
- V zásadě chceme odolat výpadku služby. Značení:
    - *chyba (fault)* - událost obecné chyby v systému
    - *výpadek (failure)* - fault pozorovatelný ze strany uživatele
- Zajímá nás:
    - Detection & prevention - jak chybám předcházet
    - Notification - jak se o chybě dozvíme?
    - Tolerance - když chyba nastane, jak dlouhý může případně být výpadek?
    - Vyjednání plánu odstávek systému, jestli je to možné. Pokud máme dostatečnou replikaci, můžeme chyby odstraňovat průběžně.
- Interně chyby maskujeme, aby nenastal výpadek a snažíme se odstranit jejich příčinu.
- Typy chyb (vzhledem k požadavku/stimulu):
    - Omission - odezva nepřijde
    - Crash - odezva nepřijde, a z velmi dobrého důvodu...
    - Timing - odezva přijde, ale pozdě
    - Response - odezva přijde a není korektní
- Environment - první nebo opakovaný fault?
- Meření v tomto případě představuje měření downtimu, času na opravu, atp.
- Taktiky na zvyšování availability
    - Dva cíle - mask a repair
    - Fault detection - ping/echo, heartbeat, exceptions
    - Fault recovery - active, stand-by, passive redundancy
        - Příprava na opravu + oprava samotná
        - Hlavní koncept - redundance
        - Basic redundancy (voting)
            - Požadavek zasíláme komponentám paralelně a “nejlepší” odezvu vracíme. Můžeme i napříč prostředím.
        - Diverse redundancy
            - Prostředí s extrémními následky faultů

                - Redundantní komponenty pocházejí z různých backgroundů (dev team, prostředí, platforma atp.)
            - Active redundancy
                - Požadavek zasíláme komponentám sériově - pokud se jedné nepodaří správně odpovědět, zkusíme druhou. Typicky užíváme v sítích.
            - Stand-by redundancy
                - Komponenty zaskakují hlavní komponentu na základě informací co jim ona periodicky poskytuje
            - Passive redundancy
                - Když selže hlavní, zapneme náhradní a restoreneme state podle nějakých informací (třeba logu)
        - Podpůrné taktiky pro recovery
            - Shadow operation - resuscitujeme failnutou komponentu a chvíli běží “nanečisto”
            - State resync - místo inkrementálních updatů keyframy
            - Checkpoint and rollback - v případě že nemáme redundantní komponentu, uděláme rollback

Upravitelnost

- Míra ceny změny
- Zajímá nás
    - Co: Změny funkcí, platforem, datového modelu, kvalitativních atributů
    - Kdy: Během návrhu, implementace, běhu
    - Kdo: User, dev, admin
- Standardní diagram - stimulus je změna, artefakt měněná část, environment vývojová fáze
- Request na změnu je buď od stakeholdera, nebo zvenku
- Taktiky
    - Lokalizace změn
        - Minimalizace modulů
        - Abstrakce běžných služeb, struktur atp. od softwaru a uložení dat
        - Abstrakce datových struktur
        - Abstrakce doménových konceptů
        - Abstrakce služeb
        - **A abstrakce obecně :D**
        - Pozor na Ripple effect - změna velmi používaného lokalizovaného modulu udělá kaskádu
    - Prevence “Ripple effect”
        - **Minimalizace propagování změn**
        - Rúzné typy závislostí komponenty B na A
            - Syntaktická
            - Sémantická
            - Identita rozhraní

            - Fixní sekvence dat a časování
            - QoS (např. přesnost měření sensoru)
            - Resources (sdílená DB)
        - Řešení:
            - Hiding, zapouzdření, dekompozice, závislosti na abstrakcích
            - Zachování rozhraní - zajistí syntaxní závislosti, ale už ne semantiku nebo QoS
                - Verzování interfaců
                - Adaptéry
                - Stuby místo smazaných objektů
            - Prostředníci
                - Pivot
                    - Centrální sdílená data či funkcionalita
                - Broker
                    - Maskuje změny identit, verzí atp.
                    - Víceméně Service locator
                    - Dynamické vazby mezi komponentami
                    - ESB
- Tedy taktiky jsou vlastně standardní zásady správného objektového návrhu - SOLID, DI

Výkonnost

- Reakce na requesty
- Můžeme měřit různé metriky, jako
    - Počet transakcí za sekundu
    - Variaci v době potřebné k nastartování motoru (requesty generuje průběh času)
- Plnění requestu vyžaduje resources
    - Měříme spotřebu resources
    - A block time
- Stream - abstrakce přicházení requestů
    - Periodický
    - Stochastický
    - Sporadický
- Arrival pattern je důležitý
- Standardní diagram
    - Stimulus - počet requestů + pattern
    - Artefakt - systémová služba
    - Evnironment - operational + normal, emergency, overload
    - Response - zpracování requestů
    - Measure - latence, throughput, jitter, miss rate
- Request je buď zpracován v několika krocích, nebo čeká
    - Blocked time - jak dlouho čeká
        - Buď kvůli resource contention
        - nebo resource availability (offline resource)

        - nebo na ostatní výpočty (synchronizace)
- Taktiky
    - Resource Demand
        - **Zmenšujeme nároky**
        - Minimalizace resource nároků - vylepšování algoritmů, cache
        - Minimalizace requestů - přímo u zdroje (pokud je pod naší kontrolou), fronty
        - Kontrola používání resource - limitovaný výpočetní čas
    - Resource Management
        - **Využívání toho co máme**
        - Lepší resources
        - Konkurence - paralelismus, load balancing
        - Redundance dat a výpočtů
            - Dvě sady dat pro dva procesy
            - Synchronizace
    - Resource Arbitration
        - Kontrola přidělování
        - Scheduling
            - FIFO
            - Fixed priority (semantic, deadline, rate - podle streamové frekvence)
            - Dynamické (round-robin)

Bezpečnost

- Rezistence vůči útokům nebo únikům, jak zvenku tak i zevnitř
- Klasická trojice atributů:
    - Authentication (autentizace) - požadavek přichází od dané entity
    - Confidentiality (soukromí) - komunikace mezi systémem a entitou je chráněná proti odposlechnutí
    - Integrity (integrita) - požadavek není někde mezi systémem a entitou neoprávněně změněn
- Další:
    - Authorization (autorizace) - daná entita má oprávnění provádět dané akce
    - Auditing - v podstatě těžkej monitoring a logování
    - Zero-knowledge - náš systém ukládá a zpracovává prakticky výhradně kryptograficky ošetřená data, která tedy sice můžeme neoprávněně (bez souhlasu majitele) vydat někomu jinému, ale nebudou jim k ničemu :). Tenhle atribut je ve 21. století skoro stejně důležitý jako klasická trojka.
- Ze slidů ještě (ale neberte to příliš vážně, naši profesoři opravdu nejsou bezpečáci):
    - Nonrepudition (nepopiratelnost) - pokud přijmeme autentizovaný požadavek, právně předpokládáme, že ho daná entita sama odeslala. Daná entita potom nemůže odeslání požadavku popřít, ledaže by u soudu předložila nezvratný důkaz.
        - Tohle je typicky případ kvalifikovaných certifikátů

    - Availability, Assurance
- Standardní diagram
    - Stimulus - útok
    - Artefakt - část systému
    - Env - online, offline, za FW atp.
    - Response - reject, log, audit trail…
- Taktiky
    - Resist attack
        - Co nejvíce snížit “attack surface” (limit exposure).
            - s vnějším světem komunikuje co nejmenší počet strojů a každý stroj dělá pokud možno jednu věc (ne, že na něm běží ještě sto jiných aplikací)
        - Klasická trojka (autentizace, šifrování a podpisy/kontrolní součty), příp. i s autorizací (podle řešeného problému - někdy je kontraproduktivní).
        - Bezpečnostní bariéry soustředit na komunikaci s vnějším světem a uvnitř vybudovat demilitarizovanou zónu (DMZ). No jo, ale bez opravdu dobrýho a drahýho firewallu je to spíše ke škodě… I uvnitř by měly být zábrany - především by stroje měly komunikovat s co nejmenším počtem dalších.
    - Detect attack (IDP produkty)
        - Anomálie - porovnáváme s historickými daty
        - Misuse detect - známe patterny útoků (Nod)
        - Drahé (ne tak, jako dobré firewally)...
    - Recover from attack
        - Resuscitace -> availability
        - Identifikace útočníka - audit trails, logy

[OK] Odhady pracnosti a doby řešení.

Zdroje:

- *Pokročilé aspekty SWI - přednáška 11?*

GANTT

- Ganttovy grafy jsou jednoduchý prostředek pro odhad času, pracnosti i ceny.
- Definuje sadu úloh, přiřazuje jim odhad doby řešení a zavádí mezi nimi závislosti.
    - => odhad doby řešení je logický důsledek
- Odhad pracnosti:
    - Pracnost = čas potřebný pro řešení vzhledem ke zdrojům, které máme k dispozici (např. tedy člověko-dny).
        - Každé úloze přiřadíme nějakou pracnost a všechny je nakonec sečteme.
- Cena se pak odvíjí od pracnosti a cen zdrojů:
    - Cena za použití zdroje (člověka) = konstanta + přesčasy a bonusy.
    - Cena úlohy = konstanta + cena za zdroje podle přiděleného času.

COCOMO

- Víceméně jednoduché rovnice výpočtu nákladů, založené na reálném pozorování:
    - Vstupem je odhad rozsahu projektu (řádky kódu).
    - Následuje černá magie (různé varianty, různé koeficienty):
        - Pracnost = 2.94 * (Rozsah)^0.91 (v člověko-měsících)
        - Čas = 3,97 * (Pracnost)^0.28
        - Cena = Čas * Plat (tady se asi myslí průměrný plat programátora?)

KARNER

- Také počítá náklady, ale z modelu jednání (aktérech a případech užití).
    - Aktérům přiřadíme váhu podle toho, se kterou částí systému pracují a co dělají.
    - Váhy aktérů sečteme (UAW).
    - Případům užití přiřadíme váhu podle počtu transakcí (pravděpodobně četnosti výskytu).
    - Váhy případů užití sečteme (UUCW).
    - UUCP = UUCW + UAW
    - UCP = (UAW + UUCP) * TCF * EF
        - TCF = 0.6 + 0.01 * technický faktor (0-5)
        - EF = 1.4 - 0.03 * faktor prostředí (0-5)
        - faktory se spočítají dle předem definovaných vážených hodnot
    - Předpokládaná pracnost = UCP * náročnost jednoho případu užití (15-30, std. 20)
        - v člověko-hodinách

Případně “planning game” z XP (agilní řízení projektu). Vidíme tedy, že nejlépe se spolehnout na sebe a hlavně na zkušenost. COCOMO může být zajímavý informativní výsledek (pro porovnání), ale Karnera je lépe poslat do blázince… třeba by ho tam přivedli k rozumu.

[OK] Modelem řízený vývoj. - OKS, SWI

Zdroje:

- *https://www.ksi.mff.cuni.cz/~richta/publications/Richta-LATES-2009.pdf*
- *https://edux.fit.cvut.cz/oppa/BI-SI1/prednasky/BI-SI1-P10m.pdf*

Definice:

- Systém chápeme nejen jako software a jeho aspekty, ale i lidi, podniky, nadnárodní uskupení atd. které se na něm podílí.
- Model = popis systému nebo nějaké jeho části či procesu. Bývá abstraktní a nebývá úplný (soustředí se na nějaký pohled/hledisko/aspekt). Typicky má vizuální formu, ale může být vyjádřen i formálně (analyticky). Dobré modely obsahují i slovní komentáře.

Modely jsou užitečné např. pro:

- Návrh a dokumentaci částí systému, resp. jeho architektury.
    - Užitečné hlavně pro vývojáře, uživatele a stakeholdery...
- Hlubší analýzu či diskusi systému či jeho části.

MDA/MDD/MDE (Model Driven Architecture/Development/Engineering):

- Principem je využití modelů při tvorbě systému, takže se nejedná o nic překvapivého (všichni jsme se učili všemožné druhy modelů).
- Modely běžně vznikají ještě před vlastním programování, ale v některých případech je lze dokonce generovat ze zdrojového kódu - např. doménový model (schéma DB).
    - => Richta uvádí pojmy dopředné/zpětné inženýrství...
- Bez modelů bychom životní cyklus SW zkrátili na sepsání katalogu požadavků, implementaci, testování a provoz.
- Druhy modelů:
    - CIM (Computation Independent Model).
        - Základní prostředí a požadavky, v podstatě zadání problému a formality pro businessmany. Změna nějakého aspektu řešení se v tomto modelu nemá projevit.
    - PIM (Platform Independent Model).
        - Funkcionalita a struktura systému, nezávislá na platformě.
    - PSM (Platform Specific Model).
        - Pořád ještě logický model, tentokráte i se specifiky platforem.
    - ISM (Implementation Specific Model)
        - Už konečný, fyzický model.
    - => lineární vývoj, v podstatě koresponduje s životním cyklem SW.
- Při tvorbě modelů se **vzhledem k udržitelnosti a sdílení** doporučuje užívat příslušné nástroje (nikoli tužka a papír). Někdy umí nástroje z modelů dokonce generovat kód nebo pomáhat při ladění/testování.
- Problémy:

    - Modely, kód i dokumentaci je třeba udržovat v souladu. To je docela hodně práce, takže modely nakonec žerou potenciálně hodně času a nákladů.
- Výhody:
    - Mnohem menší obtíže při výměnách pracovníků nebo změnách systému. Jinými slovy, udržitelnost a znovupoužitelnost, a skvělá pomoc při kontrole/auditu :).
- Podporu pro MDA nalezneme třeba v Eclipse.
- Oficiálně MDA zastřešuje Sdružení OMG.
    - Jejich snem je možnost generování modelů z hotového systému, jejich drobná úprava a opětovné vygenerování kódu (pak by stačilo už jen testovat/ladit). K tomu je zapotřebí umět modely reprezentovat a zpracovávat automaticky (metamodelování). Příklady jazyků pro metamodelování a převody modelů: relační a operační QVT a ATL.
    - Jedním z problémů metamodelování je zřejmě abstrakce modelů, a té se svět jen tak hned nezbaví :).

Analýza a návrh softwarových systémů. - ASS, SWI

Zdroje:

- *Úvod do SWI - přednáška 3?*
- *Pokročilé aspekty SWI - přednášky 2 a 3?*

Sem se toho vejde hodně… podrobněji vylíčíme část životního cyklu SW. Začněme analýzou.

Úvodní studie:

- Vstupem jsou: nápad, vize, požadavky na systém.
- Dílčí otázky:
    - Vyjasnění hranic systému (scope).
    - Analýza proveditelnosti (feasibility study).
    - Odhad potřebných zdrojů a mockup business plánu.
- Výstup:
    - Prvotní rozhodnutí o spuštění projektu.
    - Definice problému - termínů, hranic, modelu jednání a katalogu požadavků.
    - Odborný článek - sběr relevantních odborných zdrojů na dané téma.
    - Deklarace záměru - služby, cílová skupina, omezení.
    - Harmonogram - rozdělení týmové práce a zodpovědností
    - Rozpočet.
- Na co si dát pozor (nezapomenout):
    - Sekundární aktéři (např. administrátoři nebo externí pracovníci).
    - Seznam událostí - událost je něco, co vzniklo mimo systém a my musíme reagovat.
    - Případně (vyžaduje-li to problém):
        - Kontextový diagram - základní interakce, určení hranice systému.
        - Datový slovník - definice oborů hodnot a rozsahů doménových dat. Má vlastní syntax (Yourdon) pro:
            - Volitelnost ()
            - Opakování {}
            - Opci [X | Y]
            - Klíč @
            - Např. X = @Y + [A | {B}] + ©
        - Diagramy aktivity (BPMN).
        - Sekvenční diagramy.

Po úvodní studii následuje závazná specifikace, kterou ladíme se zákazníkem. Hlouběji analyzujeme a rozšiřujeme úvodní studii, dokud nebudou obě strany spokojeny. Dodatečně se zavádí technologická vymezení a podobné věci.
Po schválení specifikace jdeme ještě hlouběji a začínáme navrhovat řešení, resp. jeho architekturu, a to tak, abychom specifikaci splnili co nejlépe. Návrh architektury zpracovává téma Popis, návrh a modelování architektury softwarového systému.

Datové modelování

- Musíme mít hotový sběr požadavků a ostatní kroky
- 3 fáze
    - Analýza dat a vytvoření konceptuálního modelu (domain data, PIM)
    - Návrh reprezentace - logický datový model (CD, PSM)
    - Implementace - fyzický model

Funkční model

- Detailnější use cases, scénáře + popisy akcí (jaká data čte, mění nebo vytváří, co vyvolává, co předpokládá, co zajišťuje)
- Funkce ~= akce
- Z datového modelu můžeme odvodit funkce maticí CRUD, a dynamiku (hledáme změny stavu systému)
- Matice CRUD - řádky odpovídají objektům, sloupce funkcím, políčko C (create), R (read) U (update), D (delete)
- Stavové diagramy (KA) - popis dynamiky systému, objektu, či protokolu
- Stavové diagramy jde použít i k modelování řídících procesů (workflow)
- Životní cyklus systému (dynamika systému vzhledem k scénářům)

Kontrola analýzy

- Výstupem je konceptuální model sestávající z datového modelu (entity), funkčního modelu (služby) a dynamického modelu (změny stavů)
- Kontrolujeme pohledy samotné...
- … ale i jejich vzájemnou konzistenci
- Datový model
    - Úplný (existuje entita pro každý typ objektu? nadbytečné entity? správné vztahy? normální forma? integritní omezení?)
    - Vyvážení (porovnání s datovým slovníkem, funkčním modelem, minispecifikací - CRUD)
- Funkční model
    - Úplný (metoda či funkce pro každou událost, funkce musí být popsány, nadbytečné fce?)
    - Vyvážení (porovnání se slovníkem, datovým modelem, dynamickým modelem - řídící procesy)
- Dynamický model - úplný (model pro každou vícestavovou entitu, řídící proces, systém)?

Návrh

- Vícero fází
    - Návrh architektury
    - Návrh UI
    - Návrh komponent
    - Návrh komunikace komponent
    - Návrh integrace komponent a testování celku

- Návrh obsahuje několik rozhodnutí
    - Architektura systému
    - Datové zdroje
    - Distribuce modulů, komunikační mechanismy
    - Typy a formy výstupů
    - UI
    - Vývojové prostředí
- Konceptuální model -> Logický
    - Návrh reprezentace dat pro každé úložiště
    - Převod samotný
        - Normalizace konceptuálního modelu (vyloučení násobných atributů a multihodnot, vyloučení funkčních závislostí - odstranění redundance dat, binarizace vztahů...)
        - Návrh reprezentace (vztahů) - pro každý jednoduchý typ navrhneme tabulku (třídu), navrhneme jak reprezentovat vztahy
            - Hodně záleží na tom o jaké se jedná úložiště - v případě DB řešíme věci jako klíče, integritní omezení, triggery, pohledy...
        - Návrh reprezentaci integritních omezení
    - Návrh zajištění integrity dat
    - Návrh konzistence, zálohování, archivace
- Návrh - Funkční model
    - Dekompozice na moduly, komponenty
    - Vstupem analytický funkční model
    - Různé techniky
        - Objektový návrh - přímo vytváříme objekty
        - Na základě datových toků - převod diagramů datových toků na diagramy struktur
        - Na základě struktur - z analytických popisů datových struktur vytvoříme popisy programů
        - Funkční dekompozice - postupně dekomponujeme popisy funkcí
- Návrh dle frameworku Fusion
    - Vstupem je
        - Datový slovník
        - Objektový model - CD, ER
        - Operační model - kontext, model jednání, scénáře, atp.
        - Lifecycle model
    - Výstupem je
        - Datový slovník
        - Architektura
        - Popis komponent jako popis tříd
        - Lifecycle model

    - Klíčová idea - třída vzniká kombinací všech artefaktů (objektový model nám dá atributy, interakce metody, graf dědičnosti ehm, dědičnost, graf viditelnosti modifikátory), odtud Fusion
    - Postup
        - Pro každou operaci uděláme diagram interakce (volání metod)
            - Vytvořené operace musíme taky popsat, aby kodér věděl jak to psát
        - Pro každou třídu datového modelu uděláme graf viditelnosti (podle diagramu interakce)
            - Nutné vazby mezi třídami
        - Pro každou třídu popis třídy
            - Speciální syntax
        - Doplnit diagramy dědičnosti
            - Společné vlastnosti dat a vztahy odhalené při analýze
            - Na základě použitých knihoven
- Návrh funkční dekompozicí - dekompozice + generalizace
- DFD - Data Flow Diagrams
    - Hierarchická sada diagramů - od diagramu kontextu po elementární listy (popsané elementárními operacemi - minispecifikace)
    - Notace (Yourdonova)
        - Aktéři (rectangle)
        - Procesy (circle)
        - Paměť (rectangle s pacičkama, ňuňu) :D
- Návrh podle MSA
    - Zavedeme fci pro každou událost
    - Vstupní a výstupní toky pro událost
    - Přidat paměti
    - Zprostředka do krajů - abstrahujeme ke kontextovému diagramu (proč proboha?), dekompozicí se dostáváme k elementárním fcím
    - Diagramy struktur
        - Proces (fce) něco volá, data jsou argumenty a result
        - Dá se převádět mezi DFD a DS (MSA?)
            - Transakční analýza - DFD vybíráme transakce
            - Transformační analýza - výběr šéfa, překreslení do DS
            - Přidání transakčního monitoru (nástroj na výběr transakce)
    - DS převedeme na programy

[OK] Návrh uživatelského rozhraní.

Zdroje:

- *Úvod do SWI - přednáška 3?*
- *Pokročilé aspekty SWI - přednáška 2?*

Cílem je maximálně příjemný uživatelský zážitek, tedy ideálně rozhraní, které je:

- Správné - zajišťuje požadovaný výsledek
- Estetické - dobře vypadá, působí a neunavuje
- Rychlé, resp. responsivní - reakce do 150ms
- Jednoduché, hlavně s ohledem na časté akce
- Intuitivní - chová se podle očekávání
- Transparentní - nepřekáží tomu, co chce uživatel udělat?
- Konzistentní - nejen samo se sebou (podobné prvky a koncepty napříč celou aplikací), ale i s předchozími verzemi či prostředím (design guidelines)

Prostředky:

- Hlavně a především, komunikace se zákazníkem!
- Správná identifikace požadavků, případů užití a jejich frekvencí
- Zohlednění platformy, resp. způsobu interakce (klávesnice a myš vs. NUI)
- Dodržování obecných či specifických design guidelines
- Mapování na reálné objekty, metafory
- Prototypování

Prototypování:

- Popis UI pomocí diagramů (např. CASE) a programových prototypů.
    - Interaktivní prototyp > obrázek
    - Vhodné zejména pro jazyky podporující rapid prototyping (ne C++)
- Dává nám šanci dát UI do rukou uživatele poměrně brzo ve vývojovém cyklu, ideálně pravidelně.
    - Eliminace základních nepochopení
    - User nám přímo může reportovat co mu přijde divné
    - Zároveň je možné ho pozorovat a dělat si vlastní závěry
    - => **usability testing**
- Různé varianty a metody...
- Výhody:
    - Možnost automatického vyhodnocení použitelnosti (usability inspection). Levnější než usability testing a hodí se, když nemáme žádného konkrétního zákazníka/testera, resp. když na to nemáme peníze.

Norma ISO 9241 definuje 7 základních principů pro “feel” rozhraní (strukturu dialogů).

- Suitability for the task - rozhraní je vhodné pro řešené úlohy
- Controllability - úlohy rozděleny do kroků (flow), které uživatel nemá problém řídit
- Self-descriptiveness - rozhraní dobře popisuje každý krok (vizuálně i textově) a uživatel ho nemá problémy chápat

- Conformity with user expectations - rozhraní se snaží používat prvky, se kterými uživatel běžně pracuje a které nejsou nad rámec jeho zázemí či schopností
- Error tolerance - rozhraní uživateli umožnit opravit chybu v některém z předchozích kroků
- Individualization - rozhraní se snaží uživateli přizpůsobit, resp. pokrývá rozsah jejich zázemí a schopností
- Suitability for learning - rozhraní obsahuje prvky, které popisují jiné (+ jistá úroveň error tolerance)

A dalších 7 atributů prezentace

- Clarity
- Discriminability - přesné rozlišování informací
- Conciseness - nepřetěžovat uživatele zbytečnými informacemi
- Consistency - jednotný design, naplňování očekávání
- Detectability - vedeme usera k důležitým částem
- Legibility
- Comprehensibility - je jasné co co znamená

Jednotlivosti:

- Typografie se dá použít k hierarchizaci obsahu.
- Měněný objekt je na obrazovce a uživatel vidí změny hlavně implicitních akcí.
- Respektovat pozice věcí na obrazovce, neschovávat aktuálně nepřístupné funkce.
- Ověřování nebezpečných (destruktivních) operací, snažit se předcházet problémům.
- Techniky pro získávání pozornosti používat spíše střídmě, hlavně pro feedback. Třeba texty, animace, zvýrazňování, progress bary, číselný progress, změny ikon.
- Současný trend - orientace na obsah.
- Akce je dobré rozdělit na explicitní (např. menu) a implicitní (např. drag & drop), ale není to jednoduché. Implicitní jsou nebezpečné a musíme dát pozor, aby korelovaly s mentálním modelem.

Mentální model

- Uživatel má konkrétní představu, jak plnit úkoly na základě svého zázemí, domény a jiného SW.
- Na těchto očekáváních bychom tedy měli stavět.
- Vede to na jednoduchost - uživatelé mají streamlinované modely.
- Měli bychom stimulovat/podporovat prozkoumávání...

Obecně platné guidelines

- Law of clarity - uživatelé se vyhýbají elementům (a obecně věcem) které neznají a jejichž význam jim není jasný, důležité zejména u ikon
- Law of preferred action (detectability) - userovi musí být z rozhraní jasné co bude dělat nejčastěji
- Law of context (prostorová lokalita) - související věci a činnosti by měly být u sebe, akce by měly být poblíž věcí které ovládají
- Law of defaults - defaultní nastavení musí být dobré, drtivá většina uživatelů nemění
- Law of guided action - chceme-li aby uživatel něco vykonal, řekněme mu
- Law of feedback - jasný a všudypřítomný feedback - vizuální i textový

- Law of easing - rozdrobit větší tasky do několika menších

Metafory

- Doménový jazyk a koncepty
- Vizuální metafory
- Konkrétní, uživateli známé věci
- Balance mezi tím co metafora představuje a jak ji chceme použít

Špatné příklady:

- Windows - kradení focusu
- Windows - špatně rozeznatelné aktivní okno.
- Windows - metro design.
- NUI - rozhraních na telefonech a jejich problémy (např. big finger)

Datové a procesní modelování. - ASS, SWI

Zdroje:

- *Úvod do SWI - přednáška 5.*
- *Architektury SW systémů - přednáška 5 a 12?*

Pomáhá nejen s návrhem služeb či systému, ale i s porozuměním zákazníkovi, dokumentací nebo dokonce hodnocením finálního díla. Vzniklé modely by nakonec mohl interně užívat i sám zákazník.

- V rámci fáze návrhu SW modelujeme:
    - Informační model:
        - Datovou doménu
        - Dokumenty
    - Organizaci samotnou:
        - To co vytváří (přidaná hodnota)
        - Byznys procesy - abstraktní případy užití a sledy aktivit, které mají vytvářet přidanou hodnotu. Aktivity mohou být automatické a nemusí se týkat jedné jediné organizace. Mohou tedy pracovat např. s lidmi, týmy, dokumenty, daty.
            - Typy: management (top brass), operational (tvorba samotná, core processes), supporting (podpora operational)
        - Byznys cíle
- Modelování organizace:
    - Nejdříve use case diagramy => identifikujeme všechny aktéry v byznysu a jejich potřeby. Pro každý use case diagram vyhotovíme scénář:
        - Sekvence kroků, kde máme hlavní, příp. i alternativní flow (cesty).
    - BPMN - přímo pro modelování byznys procesů - 1 proces pro min. jeden use case.

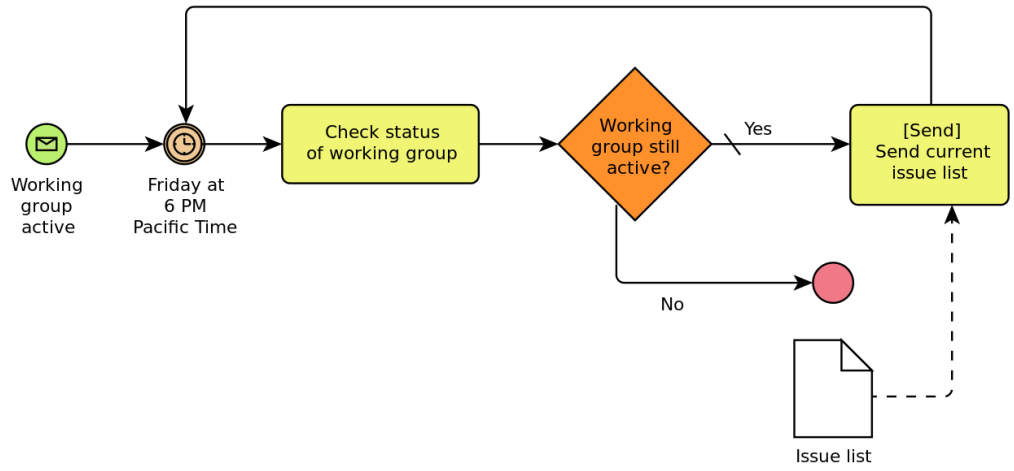
Týká se hlavně situace, kdy vyvíjíme software na zakázku, jako podpůrnou strukturu byznysu.

- Dává sice smysl i v jiných situacích, ale zaměřili bychom se spíše na informační model...
- Příklad: organizace potřebuje udělat novou IT infrastrukturu a integrovat ji s ostatními. Potřebuje, aby byla flexibilní, znovupoužitelná a škálovatelná. Potřebuje ji monitorovat a optimalizovat, a skrze ni tedy i své byznys procesy.

BPMN (Business Process Modelling Notation) 2.0:

- Cheatsheet.
- Events (události) - řídí proces z hlediska času.
    - Kolečka (zelené startovní, červené cílové, oranžové vnitřní), pro odlišení mohou obsahovat ikonky.
- Activities (aktivity) - podprocesy BP, anebo atomické úlohy:
    - Zaoblené obdélníky, pro odlišení mohou obsahovat ikonky.
- Gateways (brány) - rozhodnutí na základě nějaké textové podmínky:
    - Kosočtverce. Pro odlišení mohou obsahovat:
        - o: v každé instanci bude aktivní 1 až N větví (blíže neurčené větvení).

        - x: v každé instanci bude aktivní pouze jedna větev (exkluzivní větvení).
        - +: v každé instanci budou aktivní všechny větve (paralelní větvení).
        - *: v každé instanci bude aktivní 1 až N větví (blíže neurčené větvení), ovšem nikoli podle vstupních dat, nýbrž podle následující události.
- Data (datové objekty):
    - Obdélník s přehnutým rohem značí informaci, která procesem proplouvá (např. email).
    - Válec značí místo, kde tuto informaci lze získat.
- Participants (swimlanes a pools) - oddělení zodpovědnosti mezi různé aktéry
    - Příkladem budiž zadání (user) a vyřízení (prodejce) objednávky
    - Případ s jedním BPMN odpovídajícím více use casům
- Seskupení - dashed rectangle, seskupení aktivit kvůli dokumentaci nebo logické souvislosti
- Anotace - doprovodný text popisující jednotlivé prvky (např. aktivity nebo podmínky).
- Vztahy různých typů
    - Sequence - navázání aktivit, standardní černá šipka
    - Message flow - doručení zprávy, dashed šipka, bílý čumáček
    - Asociace - propojení s další informací nebo objektem, dashed lajna



Datové (doménové) modelování má dvě části

- Konceptuální doménový model - UML CD na nejvyšší úrovni abstrakce (jen entity a vztahy, žádné specifikátory přístupu, metodu, nic, pure domain data)
- Document model - popisuje třídu dokumentů vyskytující se v byznys procesech
    - Typicky hierarchická struktura - zakořeněný strom
    - Různé dokumenty mohou sdílet části domény, ale reprezentovat je jinak

Notace UML. - SWI, ASS

- Industry notace popisu systému z mnoha úhlů pohledu a levelů abstrakce
- Obrázek > text, lidi přemýšlejí hodně vizuálně a vztahově
- Komunikace s manažery a stakeholdery
- Díky unifikaci může být jednoznačnější než text
- CASE
- Tři hlavní typy - structure, behavior a interaction

Class diagram

- Strukturální
- Není přímo dáno pro jakou doménu je použit (typicky implementační)
- Vztahy mezi objekty + vlastnosti a schopnosti objektů
- Vlastnosti popsány dvěma způsoby - atributem (jednoduché typy) či asociací (třídy)
- Atributy mohou být složitější než typicky jsou
    - + name : String [0..1] = “no-name” {readOnly}
- Asociace představuje semantické propojení dvou tříd
    - “Navigable” šipka na konci asociace - efektivní přístup k property
    - Asociační třídy - asociace s vlastnostma
    - N-ární asociace
        - kardinality specifikují počet vazeb s (N-1)tice s třídou (v čemž se stejně nikdo z manažerů nevyzná, takže to trochu ztrácí smysl)
    - Jména vlastnosti jsou u konce asociace (tj. na druhé straně než to dává smysl, ale zase jsou u kardinalit)
    - Asociace standardně unikátní, může se použít {non-unique}
- Vlastnosti se dají definovat jako zobrazení, ale je to zvrhlost (obecně z n-tice instancí [až na n-ární asociace vždy n = 1] do m-tice instancí [array]
- Vlastnost se dá také specializovat (redefining, subsetting)
    - subsetting znamená, že property má podmnožinu instancí
    - subsettlá property musí mít jiné jméno a logicky omezené kardinality a typ
- Typy asociací
    - Agregace (nevyplněný kosočtverec u vlastníka) - “part of”
    - Kompozice (vyplněný kosočtverec u vlastníka) - také “part of”, ale jedná se exkluzivní vlastnictví, části nemohou existovat samotné
    - Generalizace
        - Dá se seskupovat do množin, specifikace kompletnosti a překrývání (např. profese)
        - Power types?
    - Realizace, značená jako generalizující závislost (přerušovaná čára)
    - Závislost na interfacu podobně jako realizace, ale standardní typ šipky (ne generalizace)
- Stereotypy

    - Přidání semantiky, rozšíření diagramu o další standardní koncepty (interface, enum)
    - Značení <<bla>> před jménem třídy
    - Seskupovány do profilů pro různé situace

Package diagram

- Zapouzdřuje další objekty (třídy, pomocné soubory, další balíčky…)
- Tvoří subsystém nebo vrstvu
- Kromě základní struktury může i exportovat elementy (syntakticky stejně jako u CD)
- Import je standardní jednosměrný vztah se stereotypem <<import>>
- Když se nám nevejde inkluze do balíčku - asociace s plusem v kolečku
- I balíčky můžeme generalizovat - přetěžování prvků
    - Nenapadá mě moc rozumné použití
    - Na slajdech příklad s balíkem Kernel, který merguje dva balíky z infrastruktury
- Merging balíků - virtuální balík, asociace <<merge>>, popřípadě @

Component diagram

- Komponenty jsou principielně skoro objekty - překrytí s CD
- Struktura systému
- Komponenty ~= rozhraní
- Komponenty můžeme vnořit
- Důraz na rozhraní a nahraditelnost
- Komponenty mají stereotyp <<component>>
- Opět můžeme koukat buď na celou komponentu a její rozhraní (black box)...
- ...nebo na vnitřní organizaci (stereotyp <<delegate>> pro navázání vnitřností na brány (čtverečkové vstupy a výstupy do komponenty)
- Pokrývá jak návrh (samička / samec, realizace pomocí ifacu), tak architektonické závislosti (standardní jednosměrná přerušovaná šipka)
- Obsahuje artefakty - fyzický projev komponenty

Deployment diagram

- Mapování elementů systému na uzly a komunikace
- Komponenty potřebné pro běh aplikace
- Stereotypy uzlů <<device>> (HW), <<environment>> (SW)
- Konkrétní zařízení jsou podtržena (jako instance tříd)

OCL

- Object constraint language
- Extenze UML o možnosti zapisovan integritní omezení
- Specifikační jazyk
- Silně typovaný
- Základní typy + kolekce
- Popis invariantu: context (x:)<jméno třídy> [inv | pre| post] (Jméno omezení): výraz

- Popis metody:
- context Osoba::pridej(kolik:Integer) : Integer
    - pre : kolik > 100
    - post : result = self.příjem + kolik
- Výhody OCL je že je strukturované a typované - není to volný text
    - MDD -> QVT -> generování IO v cílovém modelu

<TODO: Další typy diagramů - viz OCS 3-5>

Plánování a řízení projektů, řízení kvality.

- Dodržovat plán není to hlavní, jde o proces plánování - nutí nás přemýšlet o náročnosti projektu, skrytých problémech, atp.
    - “The plan is nothing, the planning is everything” - Eisenhower
- Před tím než projekt odstartujeme, je potřeba provést v rámci úvodní studie analýzu benefitů a nákladů
    - Na základě zkušeností z minula
    - Pomocí plánování
    - Pomocí statistických metod (COCOMO et al.)
- Kromě motivace k přemýšlení má plánování i další výhody
    - Odstraňuje nejistotu (vymezuje hranice systému)
    - Pomáhá v řízení projektu
    - Harmonogram a rozpočty
    - A konec konců se jich dá i držet
- Plánování se používá na různých levelech
    - Strategické - desítky let, cíle organizace, byznysu
    - Taktické - měsíce až roky, realizace alternativy (projektu) vybrané na základě strategického plánování
    - Operační - hodiny až měsíce, konkrétní úkony v rámci projektu
- Plán je tvořen několika elementy
    - Úlohy (tasks) - assignmenty vyžadující čas a zdroje na splnění
    - Milníky (milestones) - checkpointy
    - Zdroje - lidi, místa, stroje
    - Vztahy - vazby úloh, přiřazení zdrojů
- Pro vytváření plánů je spousta CASE toolů, např. MSProj a MinuteMan (vyzkoušet!)
- Plán (set elementů) jde vizualizovat mnoha technikami
    - Gantt Charts - sloupcové grafy, navazující
    - PERT (šíťové plánování) - klasický graf (ve smyslu diskrétky)
- Realizace projektu podléhá magickému trojúhelníku čas-peníze-rozsah
    - Zmenšení čehokoliv sníží kvalitu projektu
- Proces plánování je poměrně zřejmý
    - Dekompozice na úlohy
    - Identifikace vztahů mezi úlohami
    - Doplnění zdrojů, odhad potřeb
    - Přiřazení nákladů úlohám a zdrojům
    - Úpravy a ladění plánu
    - Monitorování a sledování plánu
    - Uzávěrka (schováme plán)
- Seznam všech úloh je nepřehledný - seskupujeme je do tzv. sumárních úloh, které zastupují úlohy nižší úrovně, prostě hierarchizace
- Vazby mezi úlohami jsou různých typů
    - všechny kombinace S(start) a F(finish)

        - např. vztah FS znamená že druhá úloha může začít teprve až skončí první
    - Lead a delay umožňují fuzzy vazby (např. úloha může začít když je předchozí z půlky hotova
- Pro odhad délky projektu slouží metoda kritické cesty (CPM)
    - Kritická cesta je maximální cesta mezi začátkem a koncem grafu
    - Kritická úloha je úloha ležící na nějaké kritické cestě
        - Její prodloužení způsobí prodloužení projektu

Stupně zralosti softwarových procesů.

- SW Proces = proces tvorby softwaru
- Procesy mohou být různě vyspělé, kvalitní
- Abychom měli šanci odhadnout jak kvalitní proces výroby má dodavatel, vznikají standardy
- ISO 9001
    - Kvalita řídícího procesu
    - Povinná pro dodávky státu a do zahraničí
    - “Módní”
- CMMI (Capability Maturity Model Integration)
    - Soubor pravidel, požadavků a doporučení pro firemní procesy za cílem maximální efektivity a spolehlivosti procesu vývoje
    - Narozdíl od ostatních norem, které jsou primárně o procesu výroby, CMMI se zabývá procesem vývoje
    - Vzniká na Carnegie Mellon University
    - Vyžadován v USA zejména pro kontrakty pod Department of Defense a státem, speciálně u vývoje SW
    - Sjednocuje do jednoho systému samostatné modely pro
        - Vývoj SW (-SW)
        - Elektronických zařízení (SE)
        - Integrovaný vývoj firemních procesů (-IPPD)
        - Řízení subdodávek (-SS)
    - Obecně se dá aplikovat tam kde se vyvíjí unikátní funkčnost pro splnění požadavků zákazníka
    - Modely se liší pouze pouze v konkrétních doporučeních, stejný framework
    - Je to svým způsobem open-source norma - CMU ji sice spravuje, ale nemá problém přijímat smysluplné modifikace, takže se na vývoji podílí spousta odborníků
    - CMMI je trvalý proces - nejedná se o nálepku, ale o způsob zlepšování procesu na základě doporučení a sledování úspěšnosti
    - Organizace posuzována vždy vzhledem k nějaké úrovni
    - Úrovně CMMI
        - Initial
            - Procesy jsou náhodné a chaotické
            - Nejde předvídat neboť se stále mění
            - Schopnost procesů je dána kvalifikací a schopnostmi jednotlivců
        - Repeatable
            - Je definován proces řízení projektů a vybudovány procedury pro implementaci (?)
            - Založeno na zkušenostech s předchozími projekty
            - Cílem je zavést efektivní řízení procesů

            - Schopnost procesů je “disciplinovaná” (rozumně organizovaný proces)
        - Defined
            - Definovány, dokumentovány i standardizovány jak procesy řízení, tak inženýrské činnosti integrované do celé organizace
            - Standardní sw proces organizace
                - Customizován -> proces projektu
            - Zaměstnanci i manažeři vyškoleni
            - Schopnost procesů je standardní a konzistentní, vše stabilní a opakové - definovaný proces
        - Managed
            - Stanoveny metriky sw procesů a řízení kvality produktu
            - Uchovávání, analýza a extreapolace dat
            - Odchylky od mezí jsou odhalovány a zkoumány
            - Schopnost procesů predikovatelná - proces je měřitelný
        - Optimizing
            - Prostředí pro kontinuální zlepšování procesů
            - Slabé a silné stránky procesů jsou odhalovány předem
            - Data používána pro analýzy přínosů nových technologií a změn procesů
            - Analýza defektů za účelem odhalení příčin
            - Schopnost je kontinuálně se zlepšující
    - CMMI má několik výhod oproti ostatním normám
        - Stavěná na míru vývoji
        - Nejde ji zavést “naoko”
        - Celosvětová úroveň

Řízení rizik. - SWI

- Rizika zahrnují nejistoty a ztráty
- SW projekty implicitně rizika obsahují -> analýza rizik
- Rizika jsou různých kategorií
    - Projektová - výpovědi, změna managementu, nedostupný HW
    - Techická (Product) - špatný výkon nástrojů, (hybrid s projektovým) podcenění velikosti, zpoždění specifikace...
    - Obchodní (Bussiness) - neprodejný produkt, produkt mimo obchodní strategii, ztráta podpory vedení, ztráta rozpočtu…
- Rizika se také dají dělit podle prediktability (známá odhalíme pečlivou analýzou plánu a prostředí, předvidatelná na základě předchozích projektů, nepředvidatelná těžko)
- Nebo scope - obecná a specifická pro produkt
    - Pro obecná rizika má smysl si udělat seznam - risk item checklist
- Obecná rizika dělíme do kategorií
    - Velikost produktu (rozsah, velikost db, počet uživatelů, procento reused kódu…)
    - Obchodní dopad (zisk společnosti? viditelnost produktu? dodací termín? rozptyl požadavků uživatelů? ceny průserů?)
    - Charakteristiky zákazníka (známe ho? ví co chce? je to debil? uvědomuje si že se bude muset účastnit?)
    - Definice procesu
        - Otázky procesu (správa změn, konfigurací? podpora standardizace procesu? máme použitelnou metodiku? kurzy? správně realizované inspekce?
        - Otázky techniky (podpůrné techniky? metody sw analýzy? konvence? používají se nástroje?
    - Vývojové prostředí (nástroje pro management? proces? překladače? testovací nástroje? řízení konfigurací? integrace? uměj s tim všichni?)
    - Technologie (nová technologie? nové algoritmy? rozhraní s neprověřeným SW? speciální UI? nové metody analýzy a návrhu?
    - Velikost a zkušenosti týmu (máme odborníky? máme jich dost? dostali školení? mají plný úvazek?)
- Základní projektová rizika
    - Chybný časový plán
    - Inflační množství požadavků na změny
    - Fluktuace pracovníků
    - Nízká produktivita
    - **Selhání díky špatnému zadání (1/7 projektů!)!**
- Základní nástroj je tabulka rizik - jméno, kategorie, popis, pravděpodobnost, dopad, protiopatření
    - Tvoří se seřazením rizik podle významnosti (dopad * pst)
    - Někde se udělá dělící čára

- Pro rizika nad čárou se udělá RMMM (Risk Mitigation, Monitoring and Management) plán
    - Mitigation - zmírnění pravděpodobnosti a dopadu, ideálně před započetím projektu
    - Monitoring - definujeme signály, které souvisejí s rizikem, a kontinuálně je sledujeme, zároveň zajišťujeme průběžné provádění prevenčních aktivit
    - Management - jak řešit v případě že riziko nastane (plán)
- Dokument k RMMM obsahuje kromě tabulky rizik, popisu rizik nad čárou a MMM pro každé riziko také určení zodpovědností jednotlivých rolí (nebo lidí)
    - Matice odpovědnosti?
- Je nutné dobře odhadnout zda se rizika mohou skutečně objevit a zajistit aplikaci plánovaných kroků
- Stejně jako u všeho jiného - sbírat informace pro zlepšování procesu
- Odezvy na rizika jsou
    - Předcházení (avoidance?) - eliminujeme
    - Zmírnění (mitigation) - zmenšíme dopad nebo pst
    - Akceptance - ošéfujeme riziko poté co nastane
- Opatření
    - Havarijní plán
    - Alternativní strategie
    - Rezervy
    - Obstarávání (outsourcing)
    - Pojištění

Řízení vývojového týmu, jeho dynamika, uspořádání.

Uspořádání týmu

- Strukturovaný (specializace) vs. nestrukturovaný (objem práce) tým
- Ve strukturovaném týmu jsou nejrůznější role - product a project manager, developer, lead programmer, UX designer etc.
- Strukturovaný tým může mít mnoho podob: Chirurgický tým (jeden řeže, ostatní asistují), tým hlavního programátora, agilní skupina
- Volba záleží na typu a velikosti projektu
- Pro velké projekty potřebujeme více-týmovou organizaci
- Chirurgický tým je nejlepší ale i nejdražší
- Pro menší projekty bývá nejlepší agilní skupina

Tým

- Tým je tvořen jednotlivci - stojí a padá s jejich kvalitami, potenciálem a souhrou
- Fungování týmu je potřeba budovat dlouhodobě
    - Může trvat měsíce než se nový tým dostane na rozumný výkon
- Diverzita je dobrá
- Chceme-li mít dobrý tým, musíme si jednat profiltrovat kdo se do něj dostane, a jednak věnovat kontinuální úsilí jeho zlepšování
    - Učením
- Správně bychom se měli starat o všechny lidi v týmu - pro optimální výkon musí být spokojení
    - SCRUM Master

Manažer

- Logicky hlavně ve strukturovaných týmech
- Měl by jít ostatním příkladem
- Snaží se dosáhnout cílů stakeholderů řízením
- Potřebuje spoustu pozitivních vlastností, na to by se dal napsat elaborát
- Manažer je ovlivňován týmem a naopak
- Různé styly řížení
    - Autokratický
        - Autorita, neradí se
        - Vzor
        - Pevná struktura
    - Liberální - nefunguje, nemá zájem o dění ve skupině, nehodnotí
    - Demokratický
        - leading by example
        - přesvědčuje, informuje
        - umí delegovat, motivovat, ovlivňovat mínění a iniciativu
- Manažeři jsou v různých úrovních podniku, leadeři jsou potřeba všude
- Co takový manažer dělá
    - Plánuje

        - Organizuje
        - Vyjednává
        - Vede
        - Motuvje
        - Alokuje
        - Kontroluje
        - Hodnotí

Organizační struktura

- Jak je firma stavěná
- Různé typy
    - Funkcionální - různí nadřízení pro různé oblasti, možný konflikt
    - Liniová - boss a zaměstnanec

Zavádění, údržba a rušení systémů.

VIZ údržba níže

Prototypy, verifikace a validace softwaru, testování.

VIZ Testování ve vývoji

VIZ MDD

Akceptační test

- Smyslem je ověření funkčnosti řešení (projektu)
- Definice obsahuje
    - Podmínky testu
        - Požadavky na prostředí
        - Popis všech používaných vstupních dat (DB, konfiguráky, atp.)
    - Dokumentace akceptačního testu
        - Co je potřeba pro vytvoření a instalaci produktu
        - Instalaci testovacích dat
        - Uživatelská příručka
        - Definici akceptačního testu (aktéři, use cases)
        - Protokol o provedení testu
    - Akce akceptačního testu
        - Popis všech scénářů - sada musí dostatečně ověřit funkčnost a splnění požadavků
        - Odpovídající reakce systému (kde to jde)
        - Základní chybové situace a jejich řešení
        - Každá akce musí mít identifikaci, popis, předpoklady, reakce, jaká používá data, co zajišťuje...

<TODO inspekce>

MISC

Metodiky

- Soubory doporučení a struktury procesu
- Metodika je jen šablona, nelze ji aplikovat přímo
- Customizace metodiky pro konkrétní prostředí a proces
- Dělí se na klasické a agilní
- Klasické jsou obecně striktnější, agilní flexibilnější

MSA (Moderní strukturovaná analýza)

- Glorifikovaný waterfall
- Hodně orientován na data, možná by se to dalo použít pro data-centrické aplikace?
- Trochu připomíná databázové postupy (dekompozice a ta druhá potvora)
- Analýza: Use case -> Funkce, vymyslet ukázková data -> (dekompozice, generalizace) Elementární funkce, model jednání
- Návrh
    - Transakční analýza (selekce datových toků pro každý use case)
    - Transformační analýza (pro každou transakci najít jádro co provádí samotnou transformaci
    - Jádro + input, output
    - Navíc transakční monitor (vážně?) - třída co vyvolává transakce podle typu operace

UP (Unified Process)

- Lepší než MSA (co není?)
- Industry standard, struktura popsaná pomocí UML
- Iterativní!
- Řízen use-casy
- Řízen rizikem
- Staví na architektuře, komponentách a reusability
- Každá iterace pevně strukturována coby waterfall (narozdíl od agilních metodik)
    - **P**ožadavky, **A**nalýza, **N**ávrh, **I**mplementace, **T**estování
- Iterace seskupovány do fází, fáze oddělovány milníky
- Iterace mohou být paralelní
- Fáze (a milníky):
    - Incepce - system scope, identifikace rizik, feasability study, prototypy, proof of concept, vize architektury (hrubá), odhad nákladů a výnosů
    - Elaborace - architektura, vyhodnocení rizik, výběr 80% funkčních požadavků, plán, identifikace zdrojů
    - Konstrukce - finalizace specifikace, realizace analýzy, návrhu, implementace a testování, produkt připraven k betatestu
    - Přechod - opravy chyb, zapracování změn dle problémů a požadavků uživatele, dokumentace, příručky, vyhodnocení projektu, release

- Sběr požadavků viz výše
- Obsahuje SRS a Glosář projektu (co na to domain driven?)
- Změny v UP jsou sice možné, ale nejsou moc oblíbené (narozdíl od agile)
- UP se může při malém počtu iterací a špatném nasazení zvrhnout ve waterfall

XP
Scrum
Lean?
Kanban?
Agile vs. klasické

Business-driven Development (BDD)
- Provázán se SOA
- Byznys se rychle mění
    - Konkurence
    - Regulace
    - Je tedy potřeba mít systém připraven na rychlé změny, zejména vznik a změny BP
- Několik klíčových principů
    - Adaptace metodologie
        - Forma metodologie a její přísnost (strictness) by měla záležet na velikost a fázi projektu, a měla by se vyvíjet na základě předchozích zkušeností
        - Cílem je výkonnost a lepší risk management
        - Přísnost se vyvíjí během projektu podle ujasňování konkrétních nejistot
        - Kontinuální zlepšování procesu
    - Balancing protichůdných priorit stakeholderů
        - Pohybujeme se v doméně - BP, požadavky stakeholderů
        - Prioritizace
    - Kolaborace napříč týmy
        - Cílem produktivita týmu
        - Tymy by měly pracovat jako celek
        - Motivace
    - Iterativní demonstrace hodnoty (demo stakeholderům)
        - Redukce rizika
        - Důvěra stakeholderů
    - Elevate the level of abstraction - automatizace, high-level tooling, reuse, kvalitní architektura
    - Kontinuální kontrola kvality
        - Testovat, testovat, testovat
        - Automatizace testů

CO SE JINAM NEVEŠLO

- Agile (agilní).
    - Idea: vyvíjíme lineárně, až na návrh řešení a implementaci (částečně třeba i specifikaci)? Specifikace a návrh řešení se odvíjí od implementace?
    - Pravděpodobně vhodné pro případy, kdy je technologie zdrojem velké nejistoty.
- Lean
    - Idea: neznáme přesný výsledek naší práce (nacházíme se ve fázi nápadu či úvodní studie) a rovnou se vrháme na implementaci, z níž nakonec vzejde i specifikace a návrh řešení.
    - Pravděpodobně vhodné např. pro startupy, které se vrhají do nových problémů a zákoutí a musí si zajistit financování až na základě reálných výsledků, a to v krátkém časovém horizontu.

ANALÝZA ARCHITEKTURY

- Potřebujeme nějaký framework pro porovnávání, abychom nesrovnávali pouze subjektivně podle “kvality”
- Hodnotíme vždy z určitého hlediska (např. udržovatelnost)
- SAAM (Software Architecture Analysis Method)
    - Prakticky nejznámější metoda
    - Kontext pomocí scénářů
        - Stručný popis interakce se systémem na vyšší úrovni než use case
        - Včetně sekundárních aktérů (devs, správci, admini)
    - Hlavně hodnocení okolo změn
- SAAM postupuje následovně:
    - Identifikace aktérů
    - Vytvoření scénářů a jejich prioritní seřazení
        - Scénář = činnosti které systém musí podporovat, nebo předpokládaná změna
        - 10-20 scénářů
    - Popis kandidátních architektur
        - Typicky modelová struktura, odpovědnosti, procesní struktury, interakce za běhu, komunikace
        - Konkrétní podoba závisí na tom co hodnotíme
    - Klasifikace scénářů
        - Přímé - systém zvládá splnit
        - Nepřímé - systém potřebuje upravit
        - Stupňovat hodnocení obtížnosti
    - Vyhodnocení scénářů
        - Pro nepřímý scénář zhodnotíme náročnost změn
        - Na základě počtu komponenent, datových a řídících spojení a rozhraní které je potřeba změnit / přidat / odebrat
        - Konkrétní ohodnocení vymyslí architekt
    - Odhalení interakcí scénářů
        - Více scénářů se prolíná stejných komponent -> možný problém

        - Ok pokud scénáře jsou jen varianty
        - NeOk pokud je důsledkem nedostatečného oddělení zájmů
    - Vytvoření hodnocení
        - Podle priorit (určuje firma)

Přijatelné / nepřijatelné ?

Vývoj softwarových systémů (7/10)

Architektury sw systémů, vývoj multiplatformních aplikací. - ASS, Xamarin

- Architektura je struktura (nebo struktury) systému tvořená na nejvyšší úrovni jeho součástmi, vlastnostmi a schopnostmi
    - tj. existuje nezávisle na tom, jestli o ní něco víme, každý systém má nějakou architekturu (přinejhorším monolitickou)
- Architektura je důležitá hlavně z hlediska dokumentace, návrhu a komunikace
- Typicky neexistuje jediná architektura - systém má různou architekturu v závislosti na úhlu pohledu (viewpoint, Obi-wan by měl radost)
- Referenční model je obecně známá dekompozice problému na komponenty
- Architektonické styly jsou popis komponent a jejich chování za běhu systému
    - Abstrakce sdružující podobné architektury
    - Restrikce všech možných architektur
    - např. klient-server
    - Osvědčené styly -> architektonické vzory (MVC, MVVM)
- Referenční architektura = referenční model + styl
- UML se kamarádí s popisem architektury na různých úrovních
    - Logické - class, component, package
    - Fyzická - deployment, package
- Vyplácí se oddělit data, prezentaci a řízení

MVC

- Architektonický vzor / princip
- Oddělení modelu od pohledu na něj
- UI objekty delegují na funkcionalitu modelu
    - Žádné side-effects
- Model = doménová vrstva
- Model komunikuje s pohledem pomocí DP Observer (events)
- Výhody
    - Lepší udržovatelnost
    - Možnost více rozhraní, či více modelů
    - Nezávislý vývoj obojího
- MVC má i Controller část - ta “ovládá” view a model
    - Jde přes něj většina komunikace mezi View a Modelem

PAC

- Každá komponenta architektury PAC má 3 části
    - Prezentační vrstva - pohled do dat
    - Abstrakce - data systému

    - Řízení - služby systému
- Narozdíl od MVC
    - Controller je součástí prezentace
    - Model v MVC je rozdělen do Abstrakce a Řízení

<TODO Multiplatformní aplikace WTF?>

Webové služby a servisně-orientovaná řešení, webově orientovaná řešení. - WS

- Jádrem webových služeb je komunikace mezi dvěma komponentami pomocí zpráv postavených na kontraktu (standard nebo proprietární řešení)
- Různý formát komunikace
    - Request-response
    - Subscribe-notify
    - Libovolné zprávy mezi dvěma a více komponentami
- Webová služba = komponenta na webu se známým, programově přístupným rozhraním
- Rohzraní webové služby
    - Množina operací
    - Binding operací na technologie (HTTP, SOAP, TCP)
    - Binding operací na formát (XML, JSON, RDF, TSS)
    - Service location
- Zpráva =
    - Dokument (nestrukturovaný - text, HTML, PDF; semi-strukturovaný - XML)
    - Datová struktura (XML, JSON,...)
    - Množina parametrů - request na počasí pro danou GPS lokaci a čas
    - Média
- Web service contract
    - Definuje používání služby
    - Technická část
        - Strojově čitelná
        - XML Schema, WSDL, WS-*
    - Netechnická část
        - Čte člověk (konzument služby)
        - Sémantika
        - Dokumentace
        - SLA
- SOA = architektonický styl (a asi už i vzor)
- Na WS se dá pohlížet několika způsoby, v závislosti na tom, která část struktury nám přijde důležitější
    - Message oriented - agenti si zasílají zprávy složené z headeru a body
    - Service oriented - zpráva znamená vyvolání nějaké služby, službu popisují metadata, agenti používají službu (vyšší abstrakce)
    - Resource oriented - resource má nějaké URI a možná reprezentaci, služba je resource
    - Policy oriented - policy o akcích a resourcech, agenti musí dodržovat polici
    - Všechno vlastní nějaká organizace
- Na co jsou WS vhodné
    - Interop s jinými aplikacemi přes Web
    - Po částech vyvíjené aplikace

    - Různé platformy, organizace
    - Zpřístupnění funkcionality přes web
- Hlavní výhody
    - Platformně nezávislé
    - Reusable
    - Interop
    - Škálovatelné
    - Adaptable

W3C služby

- Postavené na SOAP zprávách, typicky přes HTTP (ale klidně cokoliv jiného, třeba TCP)
    - Peer-to-peer
    - Rozšiřitelný
    - Použitelný v heterogenním prostředí
    - Lightweight
    - Ve skutečnosti vůbec není S, a je víc než OAP
- SOAP je bezstavový, jednosměrný, ale dá se použít k vytvoření složitějších paradigmat
- SOAP = XML, obalí posílaná data
- Header (volitelný) a body v envelope
    - Body = samotná aplikační XML data
    - Header = aplikační XML metadata + SOAP modules & faults
        - Složen z header bloků
- Faults - reporting chyb
    - Code (typ chyby, Value + Subcode)
    - Reason
- SOAP jde od sender k receiverovi přes intermediaries, které nesmějí číst tělo
    - Každý node má URI
    - Header block může mít atributy
        - role (none, next, ultimateReceiver + aplikační) - určuje target, target smí použít blok
        - mustUnderstand (bool) - vytváří bloky které targety musí zpracovat -> při neschopnosti fault
        - relay (bool) - forwardování header bloku
            - intermediary sežere zpracované a non-relayable bloky, ostatní přepošle - forwarding intermediary
            - volitelně mění a přidá -> active intermediary
- Header vs. block data - aplikační rozhodnutí
    - Header viditelný intermediaries
    - Body primární obsah
- SOAP komunikace buď ve stylu RPC (entity jsou objekty), nebo dokumentová (blíže k realitě)
- SOAP via HTTP - request, response, UDP, TCP (vlastní implementace paradigmat)
- Velkou výhodou je modularita - díky namespacům můžeme zpracovávat různými knihovnami a verzemi

- SOAP nevýhody
    - HTTP původ (performance problémy)
    - Bezstavový
    - By value, nejsou reference
    - XML?

REST služby

- REpresentation State Transfer
- Architektonický vzor pro “distributed hypermedia systems”?
- Principy
    - Orientace na resources
    - Unikátní identifikátor resource
    - Bezstavový
    - Uniformní interface
- Resource = věc kterou chceme publikovat (akce / objekt, konkrétní / abstraktní)
- Resource se dá reprezentovat dokumentem
    - Formát se musí stanovit
    - Reprezentace může obsahovat odkazy (formát musí podporovat)
- Resource ID neidentifikuje reprezentaci
    - Přímá identifikace
    - 303 URI - dvoufázová komunikace, request resourcu s ID vyustí v získáná ID reprezentace
- Bezstavové -> stav v klientech a resourcech
- Manipulace resources pomocí společné množiny operací
    - Např. CRUD
    - Rozšiřitelná
    - V HTTP
        - Get
        - Put - idempotentní
        - Delete
        - Post - může být granulární, obecně není bezpečný ani idempotentní
- REST + WS
    - Typicky XML, JSON nebo RDF reprezentace
    - URL = URI
    - Bezstavový
    - HTTP metody tvoří interface

WSDL

- Popisujeme rozhraní WS strojově čitelným způsobem
    - Podobná role jako IDL
- XML
- obsahuje více sekcí
    - types - XSD definice typů (může být i jiného formátu, ale XSD je standard)

    - messages (používá types) - message parts (name, type)
    - port types - kolekce operací
        - operations - I+O+F messages, dohromady různé typy operací
    - bindings - protokoly pro konkrétní port type a jeho operace
    - services - kolekce portů
        - ports - binding + síťová adresa

SOA

- Služba = samostatný kus funkcionality zpřístupněný ostatním aplikacím
- SOA je design pattern a architektonický styl
    - Založený na dekompozici na služby
- Smyslem SOA je podpora tvorby propojených enterprise aplikací za účelem řešení business problémů.
- Snaha o řešení fragmentace
    - Legacy systémy typicky samostatné bloky nepřipravené na komunikaci s externím světem
    - SOA se naproti tomu snaží neštěpit procesy a odstranit závislost na konkrétních proprietárních technologiích
- Služba se stává základním stavebním kamenem podobně jako na nižší úrovni komponenty a objekty - vlastně se jedná o další level abstrakce s podobnými koncepty
- WS nemusí být konformní se SOA
    - Ale poskytuje nám poměrně fajn možnost jak SOA realizovat
- SOA = *The policies, practices, frameworks that enable application functionality to be provided and consumed as sets of services published at a granularity relevant to the service consumer. Services can be invoked, published and discovered, and are abstracted away from the implementation using a single, standards-based form of interface. (CBDI)*
- SOA = *“Service-oriented architecture (SOA) is a style of building a software system of an organization which makes it easier to reuse and combine activities (services) and whole processes performed by the business and monitor these activities and processes.” (Byznys)*
- SOA nám umožňuje lepší domluvu s businessáky, lepší správu systému a alternativy
    - A obecně je dobrý pro překlenutí mezery mezi IT a byznysem
- Služby definujeme skrze rozhraní (viz Web services)
    - Operace a zprávy
    - Protokoly pro přístup k operacím
    - Fyzické umístění
- Rozhraní + další dokumenty popisující vše potřebné = kontrakt
- Různé typy služeb podle vztahů k částem byznysu
    - entity - konkrétní byznys doménový koncept (zákazník)
    - task - konkrétní byznys proces
    - utility - infrastruktura a technologie
- SOA má 8 základních principů

- Standardizace - standardní forma service contractu (jednodušší chápání, interop)
  - Naming konvence
  - Standardní formální jazyky pro popis datových typů zpráv, operací, protokolů…
  - Centrální model domény
  - Standardizace sémantiky - slovníček (SBVR), používání doménových pojmů
- Loose-coupling - snaha o izolaci, modifiability, minimalizaci dopadu chyb
  - Klíčová je snaha o minimalizaci závislostí na konkrétních věcech (ostatních komponentách, platformě, technologiích komunikace atp. -> abstrakce, nerozdrobená rozhraní...
- Abstrakce
  - Zveřejňovat jen nejdůležitější informace o službě
  - Cílem je loose-coupling, prevence assumptions of fungování služby
  - Řešení primárně hiding (všeho)
- Reusability
  - Reusovat služby (defacto enterprise komponenty)
  - Agnostické služby - nezávislé na konkrétních BP, technologiích a platformách, neutrální kontrakt
  - Aplikací předchozích tří principů
  - Logic Centralization Pattern - oficiální instance služeb, zákaz alternativ
  - Functional Normalization Pattern - minimalizujeme overlapy mezi dvěma službama
  - Enterprise Inventory Pattern - centrální repo a katalog, má prioritu
  - Domain Inventory - více repozitářů dle domén
- Autonomie
  - Snaha o možnost změny služby
  - Dát službě šanci poskytovat garance
  - Run-time autonomy - míra ovládání svého prostředí
    - Řešíme dedikací zdrojů
  - Design autonomy - míra kontroly nad změnou, vývojem služby
    - Loose-coupling a Abstraction
  -
- Bezstavovost - scalability, agnostika, fault tolerance
  - Někdy nejde (když nechceme dát userovi do ruky jeho vlastní state)
- Discoverability - jednoduché nalezení služby a její implementace
  - Vyhledávání a dobrý popis metadat (kontrakt, jméno, keywords, platforma…)
  - Design-time (easy) vs. run-time (hard)
- Composability
  - orchestrators vs. composition members

SOA lifecycle

- Je to skutečně cycle - obsahuje v sobě přímo adaptaci na změny v reakci na pozorování systému (důsledek rychle se měnícího byznys prostředí)
- Pět fází
    - Modelling (viz samostatná kapitola)
    - Development
        - Identifikace, design a implementace služeb A JEJICH ROZHRANÍ
        - Definice SLA
        - Propojení služeb
            - P2P
            - Service registry (Service locator pattern)
            - ESB - Enterprise Service Bus
        - Kandidáti na služby jsou z modelovací fáze (tasks, subprocesy, procesy)
            - Eliminujeme duplicity, generalizujeme… hledáme patterny
            - Logic Centralization a Normalization patterny
    - Deployment
    - Monitoring (+measuring)
        - Byznys i tech aspekty
        - Business Activity Monitoring
            - Definujeme si Key Perfromance Indicators (KPI) - počet prodaných kusů, počet nových zákazníků etc.
            - Ať si to dělaj byznysáci sami
        - Adaptace
            - Flexibilní reakce na změny
            - Verzování kontraktů a implementací
            - Buď změnou existujících procesů (pozor na důsledky), vytvořením nového…
            - Změny informačního modelu mění doménový model
            - Změny kontraktů -> meziprvek mediator
            - Změny implementace můžou způsobit změnu kontraktu a SLA

WOA

- ~ SOA + webové technologie
- AngularJS
    - Klíčová idea: Implementace MVC čistě pomocí HTML5, CSS a JS na straně klienta
    - Pro UI je použit deklarativní jazyk - podobná idea jako ve WPF, máme bindingy na model
    - Jako zdroje dat nám mohou posloužit statické nebo dynamické JSON dokumenty - můžeme je získat z webových služeb
    - Používá dependency injection

2-, 3- a 4-vrstvé architektury IS a související problémy. - ASS

nástroje, správa verzí, nástroje pro kompilaci a sestavení. - SWI, Parízek, zkušenosti, MSBuild

CASE

- Aplikační nástroje které nám pomáhají s určitými částmi fáze vývoje
- Implikují nějakou míru automatizace
- Obsahují rozsáhlé kategorie, od nástrojů pro čmárání UML diagramů a UI prototypů po nástroje pro vedení projektů
- MS Project, MS Visio, Enterprise Architect, VUE, etc.
- Za CASE se typicky považují hlavně věci okolo analýzy, návrhu a automatizace práce s kódem, tedy
    - Generování kódu
    - UML čmárátka
    - Tooly pro refaktoring
    - Tooly pro MDD
    - Nástroje pro správu konfigurací (včetně verzování)

Správa verzí

- Nástroje pro správu verzí poskytují při správném použití řadu výhod
    - Historie vývoje
    - “Checkpointy”, dá se vrátit k předchozímu kódu v případě nějakého fuckupu
    - Paralelní vývoj
    - Organizace práce - dá se rozdělit do různých branches
    - Pojmenování změn
    - Paralelní změny v dev a production tracku
    - Propojení lidí se změnami, vyhledávání defektů
- Subfield správy konfigurací
- Různé scénáře dle “integrace”
    - Manuální - na úrovni OS a uživatele, vytváření pojmenovaných verzí a variant pomocí složek
        - Nepřehledné a nepraktické pro větší než základní množství verzí
        - Emulace branchingu, ale už nejde moc dobře emulovat merging (chybějí metadata)
        - Jakž takž ok pro různé jednoduché artefakty (např. prezentace)
    - Standalone version control program (Subversion, Mercurial)
        - Obecný (jednotkou typicky soubor, ovšem bez interpretace významu)
        - Podpora branchingu i mergingu (v různé míře)
        - Jasnější přiřazení odpovědností
        - Efektivnější uložení
        - Možnost navázání na další tooling, pluginy
    - Integrované verzování (GDocs, TexMaker)

            - Potenciálně může fungovat lépe (s větší granularitou) než pomocí externího toolu, neb pracuje s datovým modelem, nikoliv jeho binární reprezentací
            - Verze většinou nejsou pojmenované
            - Hlavně textové editory
- Verzovací systémy dělíme na
    - Centralizované
        - SVN
        - Architektura server-klient
        - Pracuje se složkami (a soubory)
        - Vytvoření verze = zpřístupnění ostatním
        - Nemožnost offline práce
        - Single point of failure
        - Zámky -> práce s binárními soubory
    - Distribuované
        - Mercurial, Git
        - Peer-to-peer
        - Pracuje se soubory
        - Oddělení vytvoření commitu od jeho zpřístupnění
        - Typicky stejně existuje centrální komponenta kvůli koordinaci a kontrole, ale
            - Máme automatické zálohy v lokálních repozitářích
            - Jde stavět libovolnou hierarchickou strukturu
        - Mercurial a Git mají robustnější merging než SVN
        - Konceptuálně může být složitější na naučení
            - Hlavně při přechodu z centralizovaných vs
- Verzovací systémy jsou typicky stavěny na binárních changesetech (SVN, Hg)
    - Což funguje velmi dobře pro plaintext dokumenty a source code...
    - ...ale už ne tak dobře pro komprimované soubory (většina dokumentů) a velká složitá data (herní content)
        - Typicky se dá řešit např. pluginem rušícím kompres
        - Na velká data je stavěná Perforce, popřípadě se dá oželet verzování a použít pouze synchronizační systém typu Dropbox, ovšem za cenu separace obsahu od kódu (desync)
- Branchování a mergování nám zejména u DCVS dává velmi silný nástroj
    - Oddělení dev větve (vývoj pro další verzi) od production větve (fixy), zamergování fixů do dev větve bez ovlivnění production větve
    - Open source development
    - Merging se dá použít jako kontrolní bod pro změny -> pull requesty, code review

Sestavení (Build) <TODO doplnit C++ a C# překlad>

- Transformace z adresářové struktury do executablu
- Obecněji víceméně libovolná transformace

- Požadavky:
    - Automatizovaný překlad
    - Portabilita (více platforem)
    - Výkonnost (jeden průchod každým filem, reuse získaných informací a fragmentů)
    - Robustnost
    - Obecnost (ne na konkrétní aplikaci)
    - Jednoduchost použití (build skripty)
- Kromě vytváření spustitelného kódu má build fáze i další zodpovědnosti
    - Packaging
    - Spouštění unit testů
    - Generování dokumentace
- Build skripty a automatizace nám umožňuje používat CI -> popovídat o Continuous Integration
- Build config má typicky podobu XML souboru (jak v Antu, tak MSBuildu) se společnými elementy
    - Task - nějaká akce během buildu (kompilace třídy, tvorba diru)
    - Target - nějaký obecnější cíl (fáze) co má builder vykonat (kompilace, packaging, testing)
    - Project - množina targetů
    - Property - key-value pair
- S configem se dají dělat docela kusy, jsou tam různé pokročilé fíčury jako podmínky, atp.
- MSBuild je interně používán VS, project file = build config

Vývojová prostředí, nástroje pro ladění a testování funkčnosti a výkonnosti. - Parízek

- Dříve práce v jazyku X = volba varianty jazyka, platformy a kompilátoru, debuggeru
- Dnes IDE - kompilátor + spousta toolů okolo
    - Kontinuální kompilace, package management, refaktoring, pluginy, správa build configů, atp.
    - Dnešní IDE podporují často více jazyků a souvisejících technologií (Technology stack)
    - Schopnosti IDE mohou ovlivnit volbu jazyka

Debugging

- Testování (ať už experimentální, unit testing, atp.) nám pomůže odhalit přítomnost bugu, ale ještě ho musíme lokalizovat a opravit -> k tomu slouží debugger
- Ladíme buď
    - Manuálně - hraní si s proměnnými, sledování běhu
    - S asistencí nástroje - tracing, watches, breakpoints, změna paměti
        - Debugger lze často také připojit k běžícímu programu
- Core dump = kompletní záznam stavu programu při crashi
- Debugging pomocí vypisování ladících hlášek - když potřebujeme mnoho dat
- Online debuggery - současný stav
- Dnešní debuggery mají mnoho pokročilých funkcí
    - Historical debugging
    - Remote debugging
    - Attach debugger
    - Multi-threaded debugging
- <TODO debugging multi-threaded>
- Automatizovaný run-time checking
    - Knihovní funkce mají checky přímo v sobě (např. u malloc a free v C++)
    - U managed jazyků kontroly za běhu

Testování funkčnosti - viz testování obecně, zejména unit testy

Logování

- Run-time monitoring
- Typicky v produkčním prostředí - nechceme hned ze všeho dělat aféru
- Různé úrovně - např. error, warning, notify
- Má smysl hlavně u dlouho běžících programů
- Log4j 2, Windows Event Log, NLog, Enterprise Library
- Jako spouštěče můžou složit různá volání systémových a knihovních funkcí, nebo přímo programátorem definované události
- Offline analýza
- Log4j
    - Javí

    - Hierarchická struktura
    - Konfigurace v souboru (dynamická)
    - Více výstupů (i podle severity hlášky)
    - Formátování
- Standardní způsoby implementace logování
    - Přidaná závislost
    - Singleton správce
    - Aspekty

Pohovořit trochu o bugtrackingu - úzce souvisí s laděním
Zmínit statickou analýzu
Performance

- Souvisí buď s konkrétními nefunkčními požadavky na rychlost, nebo aspoň obecně s použitelností
- Cílem je najít kód kde je tráveno nejvíce časů a ten zoptimalizovat
    - Nikoliv naopak (předčasná optimalizace)
- Základní nástroj je Call graph
    - Jména funkcí a čas
    - Kdo volá koho
    - Inclusive vs. exclusive
- Rúzné performance charakteristiky jsou důležité pro zákazníka
    - Throughput
    - Latence
    - Střední doba zpracování požadavku
- Hlavní přístupy
    - Profiling
    - Benchmarking
    - Load testing
- Profiling
    - Měříme frekvenci a trvání volání
    - Dva přístupy
        - Sampling - stav programu v krátkých časových intervalech
            - Přesnost pochopitelně záleží na době běhu
        - Instrumentation - vstupy a výstupy z funkcí, modifikace bytekódu
    - Alternativně ještě HW performance čidla
- Některé základní informace lze zjistit z informací o procesu
    - JConsole (přímo napojena na JVM)
    - ProcExp
    - Monitoring zátěže karty
- Korektní performance analýza není žádná trivka
    - Spousta skrytých činitelů (cache, garbage collection, JIT, atp.)
    - Drobné odchylky v měření -> statistika
    - Více viz <TODO vyhodnocování výkonnosti>

Objektově orientované jazyky a technologie, návrhové vzory. - OKS, NV, Wiki, GoF

- Dnes je drtivá většina jazyků objektová (C#, Java), nebo objekty aspoň podporuje (C++)
- OOP má tři základní koncepty
    - Encapsulation (zapouzdření) - schovávání detailů implementace, seskupení souvisejících věcí do jedné třídy
    - Abstraction - k potomkovi se dá přistupovat jako k předkovi
    - Polymorphism - konkrétní chování záleží na jedinci
- Třída je abstraktní definice vlastností a schopností nějaké entity
- Objekt je konkrétní instance
- Abstrakce je typicky implementována pomocí dědičnosti
    - Single nebo multiple
    - Multiple inheritance potenciálně způsobuje dost problémů
        - Diamond problem - ze kterých implementací poskládat vytvářený objekt?
        - Diamond inheritance je řešena různými způsoby
            - Zákazem multiple inheritance s výjimkou implementace ifaců
            - Virtuální dědičností (C++)
        - Co znamená volání super.m()?
        - Fragile base class
            - Synaktická - nekompatibilita vzniklá změnami base třídy -> dispatch tabulky až při startu
            - Semantická - jak zajistit aby implementace potomka byla stále validní?
        - Inheritance anomaly
            - Změny metod v potomkovi nutí změny v předkovi a ostatních potomcích - což celkem boura pointu dědičnosti
            - Je různých typů, ale to snad není nutný hrotit
            - Problém je hlavně v synchronizačních primitivech - např. čekání na uvolnění kritické sekce je přímo součást kódu metodu, změny nás nutí psát celou metodu znova kopírováním kódu
        -
    - Dědičnost ve skutečnosti implementuje dva koncepty dohromady - subclassing (skládání implementace) a subtyping (záměnnost, polymorfismus)
        - Private dědičnost v C++ do určité míry umožňuje subclassing bez subtypingu
        - Subtyping bez subclassingu - duck typing, záměnnost dle chování (dynamic)
- Typová parametrizace
    - Templates (C++) - specializace jsou různé (naprosto nesouvisející) třídy

    - Generics (C#, Java) - speacializace je jedna třída (<object>), kompilátor dělá statickou kontrolu, zkompilovaný kód obsahuje casty
        - V Javě nemůžeme použít generika na value typy
        - V C# ano - specializace pro primitivní typy a object
    - Variance - přenáší se dědičnost typových parametrů i na generika?
        - Covariance (přenáší se ve stejném směru), contravariance (přenáší se v opačném směru)
        - Invariance - u mutable typů (aby nebylo možné castovat na obecnější typ a přiřadit někam jinou specializaci typového parametru)
- Další fíčury moderních programovacích jazyků
    - Reflekce - modifikace schopností tříd buď během kompilace, nebo za běhu (C#)
    - Runtime reflekce - vyžaduje podporu runtime typového systému
        - Možná více o metatřídách?

Návrhové vzory

- Pojmenované a popsané řešení typického problému
- Z architektury
- Přínosy na mnoha úrovních
    - Řešení je známé a ověřené, méně prostoru pro chyby
        - Ale pořád je potřeba správně identifikovat možnost použití a neposrat implementaci!
    - Lépe se nám o tom mluví, protože to má jméno
    - Líp se to vysvětluje manažerům, stakeholderům a dalším (pokud je už nutný jim to vysvětlovat)
    - Líp se to dokumentuje (“implementuje to rozhraní společné s dalším objektem, který…” vs. “proxy.”)
- Základní vzory jsou z GoF, nicméně nejsou jediné (Fowler má spoustu enterprise vzorů, architektonické vzory jako MVVM, MVC atp. se taky dají svým způsobem považovat za NV)
- Ve skutečnosti se typicky až na složitější vzory (koukám se na tebe, visitore) jedná o řešení na která by dobrý programátor po kratším zamyšlení přišel, což jim ale neubírá na užitečnosti

Pohovořit o pár vybraných vzorech

- Singleton
    - Zmínit problémy:
        - Globální instance - nejsou přímo vidět závislosti metod a objektů
        - Tight coupling
        - Persistentní stav (testování!)
        - Single responsibility (schopnosti třídy jako takové + kontrola lifetimu)
- Proxy
    - Částečně vzniká samovolně dobrým objektovým návrhem (dependency on abstractions)
- Observer

    - Zmínit spojitost s event systémem v .NET
- Factory method
- Abstract factory
    - Zmínit spojitost s multiplatformními implementacemi - viz nVidia PhysX, atp.
- Template method
- Bridge
    - Klasický scénář s GUI

Testování, testovací scénáře, metody testování černé, šedé a bílé skříňky, testování uživatelského rozhraní. - SWI + prodat všechno o TDD a Pexu

- Testování = ověřování výsledku pro vstupní data
- Validace = testování subsetu všech možných vstupních dat (vyrobili jsme správný sw?)
- Verifikace = testování množiny vstupních dat (vyrobili jsme ho správně?)
    - Typicky vyžaduje velkou dávku formalizace
- Význam testování
    - 40% času vývoje a 50% nákladů je ověřování
    - 75% aplikací má problém s kvalitou
    - 1% je doděláno včas, v rámci rozpočtu a v požadované kvalitě
    - Tedy velké riziko problému - **testování je nástroj na řízení rizika**
    - Zejména by mělo sloužit k ověření
        - Reakce na nepredikovatelné události
        - Chování v reálných podmínkách nasazení
        - Ověření přenositelnosti instalace
        - Chování při havárii a zotavení
        - Spokojenosti zákazníka
    - Zajištění kvality, ale i důvěry v kvalitu (ze strany zákazníka a vyšších pater)
- Testy se dají kategorizovat různými způsoby
    - Podle části systému (testy UI, modulů)
    - Podle jejich primárního smyslu (integrační testy, finální testy, akceptační testy, regresní testy)
    - Podle způsobu testování (struktura dat, struktura programu, časové závislosti u paralelních programů)
    - Podle formy (dynamické-programové, statické-uživatelské, zejména code review)
- Požadavky na proces testování
    - Testy založeny na specifikace a uživatelské dokumentaci
    - Minimálně ověřit všechny požadavky specifikací a doménovými standardy
    - Testy jsou opakovatelné, přesně definované a dokumentované
    - Stanoveno a dodrženo testovací prostředí a podmínky, testy nic nesmí ovlivňovat
- TPC testovací benchmarky?
- Blackbox testování
    - Testujeme funkčnost nějaké dílčí jednotky pouze z hlediska správnosti dat (testování struktury dat)
    - Bez znalosti procesu výpočtu
    - Testy vymýšlí člověk na základě znalosti problematiky a specifikace
    - Typicky snaha zaměřit se hlavně na okrajové případy (edge cases)
    - Ideální je mít po ruce zákazníka (on-site customer) pro asistenci s testováním
- Whitebox testování
    - Testujeme se znalostí vnitřního fungování
    - Často podle struktury programu
    - Tester může modifikovat program aby se lépe testoval

    - PEX and Moles
- Graybox
    - Jako whitebox, ale tester kód jen vidí, nemůže na něj šahat
- Ostatní metody
    - Simulace činosti uživatele (UI testing)
    - Inspekce (audit)
    - Porovnání se standardem (atest)
    - Sledování a vyhodnocování (monitor)
- Fáze testování
    - Strategie - definice účelu a pozice testování
        - Musí zvážit
            - Specifická rizika a cíle projektu
            - Ekonomická stránka
            - Organizace a životní cyklus projektu
        - Specifikuje
            - Systém testování (typy, zaměření, postupy)
            - Způsob vyhodnocování
            - Standardy
            - Testovací týmy
    - Plánování
        - Co testovat (identifikace klíčových oblastí)
        - Jak testovat (výběr metod a způsob jejich použití, jak se zpracovávají neshody)
        - Formalizace
        - Kdo testuje
        - Kdy se testuje
    - Návrh
        - Definice částí aplikace k testování
        - Určení testu a definování cíle
            - Hlavně odhalení chyb
            - Testy efektivní
        - Vstupní data pro test
        - Na bázi
            - Dokumentace systému
            - Znalosti tvůrců
            - Reverzního inženýrství
    - Tvorba
        - Dle návrhu (s případnou korekcí)
        - Příprava dat
        - Specifikace podmínek
    - Návrh testovacích cyklů
        - = skupiny testů podle cílů a účelu testování (integrační testy, testy modulů. atp.)

        - Provádění a vyhodnocování
            - Dokumentování výsledků!
            - Dokumentace nesouladů
                - Kategorizace
                - Návrh řešení
        - Změny (odstranění neshod)
    - Akceptační testy? SWI
    - **-> TDD**
    - **-> Dependency injection, SOLID, mocking (závislosti na abstrakcích, jak docílit, atp.)**

Typy náhražkových objektů

- Dummy - předáváná reference, ale není používána
- Fake - zjednodušená implementace (MemoryStorage v Shadow World)
- Stub - předdefinované odpovědi
- Mock - stub + kontrola správnosti používání (např. pořadí volání metod)

Provoz a údržba, detekce a odstraňování chyb, konfigurační řízení.

Údržba

- Systém se nachází ve stavu kdy byl dodán, akceptován a provozován
    - A při troše štěstí neobsahuje moc chyb
- Cílem údržby může být systém
    - Udržovat v chodu (oprava chyb, konfigurace)
    - Dělat drobné změny
    - Systematicky rozvíjet
- Rytmus dodávek - pravidelný, nebo malé dodávky pro naléhavé věci
- ISO/IEC 14764 rozlišuje různé typy údržby
    - Corrective - oprava nalezených chyb a problémů
    - Adaptive - adaptujeme SW aby byl stále relevantní (byznys)
    - Perfective - vylepšujeme
    - Preventive - detekce a prevence faultů
- Každý požadavek na změnu by měl projít regulerním life-cyclem
- Je potřeba dbát na to abychom změnou pokud možno nezanesli do systému chyby - je třeba si uvědomovat, že systém se nachází v produkčním prostředí
- Má cenu si definovat metriky za účelem budoucích servisních smluv
- Pro údržbu je bezpodmínečně nutné mít kvalitní dokumentaci - systém upravujeme dlouho po jeho vytvoření, a možná ani nejsme autoři
- Je dobré používat vývojové prostředí maximálně podobné produkčnímu a kontinuálně testovat
- V rámci údržby se můžeme střetnout s architekturou
    - Nový požadavek není podporován současnou architekturou
    - Malými změnami architekturu postupně zlikvidujeme
- Hodí se mít ve změnách pořádek - evidence požadavků, definovaný proces změnového řízení, namapování na dodávky
- Ve fázi údržby už prakticky nemáme šanci zavést komplexní testování - buď existuje a funguje, nebo ne
- Pořád bychom na údržbě chtěli vydělat - maximalizovat efektivitu procesu a přesnost odhadů
- Ještě větší důraz na kvalitu a efektivitu než v normální produkci -> nutná disciplína
- Narozdíl od produkce se také pohybujeme ve známém, dobře definovaném a stabilním prostředí…
- … ale zase velmi složitém
- Tím že se nejedná o tvorbu komplexního systému napříč časovým horizontem můžeme použít pro změnové řízení metodiky jinak nepoužitelné - například klasický waterfall

Vývojové prostředí, dodávky systému, akceptační a produkční prostředí, distribuce a instalace software.

- Základní problém: Software je určen pro nějaké konkrétní reálné prostředí (nebo více) u zákazníka - my ho ale potřebujeme zkoušet, podrobit akceptačnímu testu, atp.
- Existuje nám tedy více prostředí, ve kterých produkt běží, ne nutně funkčnost v jednom implikuje funkčnost v druhém
    - Prostředí by mělo být co nejvíce věrné tomu reálnému
- Continuous integration kombinuje vývojové, integrační a testovací prostředí
    - Dobře se páruje s TDD -> pohovořit
- Měli bychom mít zálohy tak, aby bylo možno vývojové prostředí z ničeho opět postavit
- Dodávky
    - Musím ji umět:
        - Vyrobit (build)
        - Nainstalovat
        - Připravit pro instalaci zákazníkem
        - Dodat celý systém
        - Opravit drobnosti
        - Zvládat různé typy prostředí
    - Měl by existovat jednoduchý a ideálně automatizovaný systém dodávek
    - Chceme je identifikovat
    - Každý release by měl odpovídat nové verzi - nové vývojové prostředí a jeho dokumentace, změna integračního serveru, konfigurací, db…
    - Měli bychom mít dohodnutý pevný formát dodávky, a ten vždy dodržet
- Daily build
- Co největší automatizace a kontrolní výstup (logy, reporty testů)
- Opět je dobré dát všemu co nejpevnější strukturu - popíšeme prostředí do detailu - z čeho sestává, jaké jsou v kterém prostředí platformy, atp.
- Je potřeba najít nějaký kompromis mezi počtem prostředí (a tedy i pokrytím), a jejich udržovatelností
    - Přehled o tom co je kde nainstalováno za release - identifikovat release!!

Správa a řízení konfigurace = SCM

- Sledování a kontrola změn v konfiguraci softwaru
- Konfigurace = konkrétní stav SW produktu (včetně všech částí) v nějaké fázi vývojového procesu
    - Tj. jak dílo samotné, tak všechny ostatní artefakty - dokumentace, datové soubory,
- Primární cíl SCM - zajistit pořádek a řád v rámci celého SW produktu
    - Zajistíme evidenci
    - Zajistíme identifikaci všech částí
    - Zajistíme, že provádění změn nepoškodí produkt (regression testing?)
    - Zajistíme možnost získání přehledu o stavu konfigurace
- Pro kód nám pomáhá všechno týkající se verzovacích systémů - umožňuje to sledovat konkrétní změny, kdo je udělal, současnou práci více lidí, atp.
- Chybí nám ale stále vazba mezi changesety ve verzování a ostatními částmi systému - zavedeme nějaký tooling pro řízení změn (CM)
    - Např. Bugzilla, Jira, Trac… prostě něco co nám umožňuje sledovat požadavky na změny samotné, a jejich stav
    - Rozlišujeme různé typy změn - opravy bugů, přidání specifikované funkcionality, atp.
    - Styčný bod mezi specifikací a verzovacím systémem
    - Každý potenciální změna tak prochází předem daným vlastním životním cyklem
        - Někdo ji pojmenuje, identifikujeme item(y) který je pro změnu potřeba změnit, formálně zadefinujeme popis requestu (a několikrát ho vrátíme k doplnění, je-li to třeba)
        - Uděláme analýzu jestli má změna smysl a jestli je možná, odmítnem nebo přijmem
        - Někomu ji přiřadíme, změna projde malým waterfallem
- SCM plán - dokument definující SCM postupy v projektu
    - Účel
    - Odpovědnosti, politiky, procedury…
    - Řízení verzí (DCVS)
    - Řízení změn (Jira)
    - Koordinate se zbytkem aktivit
    - Zdroje
- V jednoduchosti je síla - čím je proces jednodušší a transparentnější, tím bude efektivnější -> pečlivá volba toolingu

MISC

Komponentové systémy

- CBSE - component-based sw engineering
- Orientace na reusability
- Samostatné části se službami
- Základní principy CBSE
    - Nezávislé komponenty, komunikace skrze interface
    - Komponentové standardy pro jednodušší integraci
    - Middleware support pro interop
    - Vývojový proces zaměřený na reuse
- Komponenty jsou stále víceméně high-level komponenty kombinované s rozumným SWI (abstrakce, hiding, virtualní platforma)
- Standardy
    - Nutnost pro komunikaci a interop
    - Je jich několik, navzájem nekompatibilní -> omezilo CBSE
        - Java Beans
        - COM a .NET
        - CORBí CCM
- Potýkáme se s různými problémy
    - Jak zajistit že se dá získané komponentě věřit? (trustworthiness)
    - Kdo bude komponenty certifikovat a kontrolovat? (certification)
    - Jak predikovat důsledky propojení komponent (emergent property prediction)
    - Jak porovnávat komponenty? (requirements trade-offs)
- Komponenta poskytuje službu bez znalosti konkrétního kontextu
- Charakteristiky komponenty
    - Standardizovaná
    - Nezávislá
    - Composable
    - Deployable (self-contained)
    - Zdokumentovaná
- Komponenta se dá brát jako service provider
- Komponenta je spojena s okolním světěm skrze dva typy rozhraní - provides (API) a requires (dependencies)
- Modely definují standardy pro implementaci, dokumentaci a deployment, včetně toho jak definovat rozhraní a co všechno rozhraní obsahuje
- Modely sestávají ze tří hlavních částí
    - Interfaces
    - Usage - hlavně unikátní identifikace,
    - Deployment - packaging
- Na modelech jsou postaveny middlewary
    - Musí poskytovat způsob komunikace mezi komponentami (platform services)
        - Adresace, definice ifaců, exceptions, komunikace

    - A aplikačně nezávislou základní sadu služeb -(support services)
        - Security, správa resources, správa komponent, atp.
    - Komponenty deployujeme do kontejnerů (Container) - sada interfaců pro přístup k implementacím služeb
- CBSE procesy jsou SW procesy beroucí CBSE v potaz
    - Buď přímo vývojem komponent (dev for reuse)
    - Nebo vývojem z existujících komponent (dev with reuse)
- Podpůrné procesy
    - Component aquisition - získávání zevnitř nebo zvenku
    - Component management - správa komponent které máme ve společnosti (katalog, storage)
    - Component certification
- Komponenta je jednotka SW jejíž funkcionalita a závislosti jsou kompletně definovány jejími rozhraními
- Dev for reuse
    - Asociace se stabilní doménovou abstrakcí ~= reusability
    - Generalizace existujících komponent
        - Odstranění věcí specifických pro aplikaci
        - Změna jmen
        - Přidání metod
        - Konzistentní exception handling
        - Konfigurační iface
        - Integrace dependencies
    - Trade-off mezi reusability a usability
    - Exception handling - publikovat ven, ale jsou s tím problémy (bloated iface, vnitřní handling může být potřeba pro správnou funkcionalitu)
    - Může stát víc než normální specifická implementace -> cáluje organizace, ne projektovej budget
- Dev with reuse
    - Trade-offs, kompromisy
    - Měníme požadavky na základě dostupných komponent (i opakovaně)
    - Systém pak složíme z komponent
    - Problémy s důvěrou (trust)
    - Problémy s se splněním požadavků
    - Problémy s validací - nevyžádaná funkcionalita, malá specifikace
    - Validace komponenty
        - Vytvoření test casů a jejich spouštění (funkcionalita)
        - Chtělo by to i testovat jestli tam není něco ošklivého navíc
    - Kompozice - vytváříme integrovanou komponentu (sekvenční, aditivní a hierarchická kompozice)
    - Různé typy nekompatibility (parametrická, operační nekompatibilita, operační nekompletnost) -> Adaptér
    - Sémantická kompatibilita je dána dokumentací

- Propojení requirements engineering a system design

Distribuované systémy (12/13)

Komunikace zasíláním zpráv (a JMS). - MWy

- Spojovaně / nespojovaně
- Rozhraní: send / recv
- Implementace: send a receive fronty
- Zpráva = blok bajtů / ale může být i nějak typovaná
- Adresuje se typicky proces, nebo fronta
- Nonblocking = send nečeká na odeslání, recv funguje jako peek (neblokuje)
- Blocking = send čeká na odeslání, recv blokuje, dokud něco nepřijde
    - Jednodušší použití, omezenější možnosti
- Synchronní = čeká se na doručení / asynchronní
    - Kdyby se někdo ptal, tak může existovat i synchronnous nonblocking komunikace (OMG), protože můžeš poslat zprávu neblokovaně a pak synchronně čekat

JMS

- Messaging interface, podstatný je, že umí point-to-point komunikaci (queues) i publisher-subscriber model (topic)
- Sun; část J2EE, kooperuje s EJB a JNDI (ať je to co je to)
- Základní prvky (objekty):
    - Connection - spojení s ostatními klienty (serverem), typicky asi bude jen jedno
    - Session - přibližně odpovídá komunikačnímu kanálu - v rámci sešny se provádí ordering, listening a transakce
    - Destination = cíl zprávy, dvě možnosti
        - **Queue** = pojmenovaná fronta, kterou si někdo vytvořil a my mu do ní pošlem zprávu (point-to-point komunikace)
        - **Topic** = publisher-subscriber, zpráva se doručí všem, kdo jsou k topicu přihlášeni
            - existuje “durable subscriber” - pro něho se pak schovávají zprávy i když je zrovna odpojen, jinak by se zahodily
    - Message - zpráva, může být sekvence bajtů, text, objekt, stream, nebo mapa (klíč, hodnota). Zprávy se posílají buď do front nebo do topiců
    - Producer - vytváří zprávy (je to taková factory zpráv)
    - Consumer - čteč zpráv (vytváří se nad frontou nebo topicem), lze číst blokovaně i neblokovaně, nebo nastavit async. listener

RPC (a RMI, CORBA, SOAP).

- RPC je vlastně synchronní posílání zpráv popisujících akce na server, který odpovídá výsledkem - častý pattern, proto implemetováno middlewary
- Na straně klienta stub, který jen posílá zprávu na server, tam skeleton, co to provede a vrátí výsledek, který se zase zprávou pošle zpátky

- RPC = obecný princip “volání vzdálené procedury”, typicky implementované jako RMI (volání metod na vzdálených objektech)
- Transparentní (ale ne úplně), může selhat síťová komunikace, problém s referencema

Java RMI

- Takový klasický jednoduchý RPC, integrovaný v Javě
- Používá server *rmiregistry*, ve kterém se registrují vzdálené objekty
- Server:
  - Vytvořit třídu odvozenou od RemoteObject, metody musí mít throws RemoteException, hodnotou lze předávat jen serializable typy
    - Lepší mít interface odvozený od Remote a ten implementovat
  - Vytvořit instanci třídy a zavolat Naming.rebind(“//server/jméno”, obj)
  - Tím se vytvoří nový vlákno, který spravuje requesty
- Klient:
  - Naming.lookup(“//server/jméno”) vrátí referenci na vzdálený objekt (proxy)
  - Na získaném objektu pak můžeme volat jeho metody, that’s all
- Kdy uvolnit objekt z paměti?
  - Pokud neexistují žádné lokální, ani vzdálené reference
  - Reference counting nevhodný => klienti by museli ref. odhlašovat
    - co když crashnou?
  - Jak se spravují vzdálené reference?
    - Lease = 10 minut, pak je reference volná
    - Klient automaticky prodlužuje lease po 5 minutách

CORBA

- Jazyk IDL (viz další sekce)
- Servant = třída obsahující kód (na serveru)
- IOR = interoperable reference - odkaz na vzdálený objekt (nutno asi nějak poslat)
- Klient
  - Používá proxy, která vypadá jako standardní objekt
  - marshalling, unmarshalling (viz další sekce)
  - Klient nepoužívá new, buď dostane IOR, nebo mu objekty vrátí nějaká metoda (typicky nějaká factory třída “Registry”)
- Server
  - POA (portable object adaptor) analyzuje příchozí request, najde správnou proxy a na ní zavolá invoke
  - Proxy nevypadá jako normální objekt, má invoke metodu, ta unmarshalluje parametry a zavolá správnou metodu na servantovi, výsledek zamarshalluje a pošle zpátky
- POA
  - Active Servant = třída v paměti serveru, která dělá ten “business”
  - Object Id = identifikace objektu
  - Object Reference = dvojice (POA identity, ObjectId)

    - POA spravuje asociace mezi ID objektů a servanty, udržuje si mapu ObjectId->Servant a v ní hledá
        - Můžeme mu to ale zakázat a spravovat si to sami (default servant, servant manager), nebo si můžeme sami přidělovat IDčka
    - Je možno mít víc POA a tím volit různou politiku pro různé třídy
- Výhody: object-oriented, platform/language independent, celkem robustní
- Nevýhody: nutí používat IDL typy, obecnost může být na škodu (oproti např. RMI)

SOAP

- Výměna zpráv na XML přes síť, typicky pro komunikaci mezi webovými službami
- Většinou nad HTTP
- Typicky jeden je klient, druhý server, klient zadává požadavky, server vrací výsledky (v tom je to RPC)
- Formát dotazu něco jako: <m:GetStockPrice><m:StockName>bla</…..></>
- Envelope, header, body
- … více viz Buckeyho poznámky k Webovým službám ...

Objekty v distribuovaném prostředí (koncepty IDL, proxy, marshalling, reference, předávání argumentů, paralelismus, distribuovaný garbage collector). - OKS+MWy+???

- Jazyk IDL (interface description language)
    - Jen popisuje rozhraní (žádný kód)
    - Neobsahuje věci specifické pro konkrétní prog. jazyky
        - Limitující, ale zajišťuje univerzalitu
    - Umí primitivní typy, struktury, výjimky, uniony, enumy, interface (vč. abstraktních i. a dědičnosti), sekvence, typedefy
    - Mapování IDL typů do C++ a Javy - snad netřeba hrotit
- Proxy = “reference” na vzdálený objekt (má klient)
    - Rozhraní jako vzdálený objekt, místo metod stuby, které posílají zprávy
    - Chytrý RMI je umí vygenerovat na základě vzdáleného rozhraní
- Marshalling = “zabalení” dat do zprávy
    - v RPC argumenty, návratové hodnoty
    - Formát dat (kódování, endiáni) - buď daný sítí, nebo domluvou, nebo napsáno ve zprávě
- Unmarshallling = “rozbalení” parametrů/návratové hodnoty
- Reference = problém s pointery, klient a server jsou v různých adres. prostorech
    - Poslat hodnotou (jak promítnout změny na objektu? poslat ho zpátky?)
    - Poslat nějakou proxy - CORBA to umí u objektů v IDL
- Předávání argumentů
    - Hodnotou?
    - Odkazem? IDL podporuje, v Javě/C++ se to pak mapuje na nějaký Holdery
- Paralelismus
    - Jak zpracovávat požadavky - sériově / paralelně

    - Při paralelním zpracování třeba mít thread-safe servanty
    - Threading modely v CORBě (nastavují se jednotlivým POA)
        - Thread-per-request
        - Thread-per-object
        - Thread pool (podle mě je kombinovatelný s těma dvěma nahoře)
- Distribuovaný garbage collector - viz Java RMI

Skupinová komunikace, virtuální synchronie, doručovací protokoly. - PDS

Skupinová komunikace

- Obecně, zprávy všichni všem (jeden odesílatel, více příjemců)
    - Neexistence sdílené paměti => zprávy
- Kdo může zajišťovat spolehlivost (sender-initiated, receiver-initiated)
- Souvisí s tím: (logické) hodiny a jejich synchronizace, distrubuované vyloučení, volba koordinátora, doručovací protokoly, virtuální synchronie, změny členství ve skupinách, detekce globálního stavu, distribuovaný konsenzus

Virtuální synchronie

- Pojem “group view”
- Pravidla pro pořadí zpráv a změn group view
- Algoritmy Transis a ISIS

Doručovací protokoly

- Zajišťují uspořádání zpráv ve skupinové komunikaci
- Globální (nejde)
- Sekvenční
    - sekvencér
    - dvoufázový distribuovaný algoritmus s využitím skalárních logických hodin
- Kauzální
    - vektorové hodiny
    - maticové hodiny (pro překrývající se skupiny)
    - spolehlivé doručování (Trans algoritmus)

- Multicast (broadcast) - spolehlivost
    - Sender-initiated - uzly posílají ACK, sender musí mít přehled o všech (aby zbylým přeposlal zprávu) => špatně škáluje, může zahltit síť mnoha ACKy
    - Receiver-initiated - uzly posílají NACK, musí poznat, že něco prošvihly (číslování paketů), u nepravidelných zpráv lze použít keep-alive
    - (Logická) stromová topologie - otec je zodpovědný za doručení svým synům

- Cíl: Doručení zpráv v nějakém rozumném pořadí
- Rozlišujeme různé druhy “rozumného uspořádání”
    - Globální - zprávy doručovány v pořadí odeslání, v distribuovaném prostředí NEMŮŽEME zaručit (nemáme globální hodiny :( )
    - Sekvenční (total order) - zprávy jsou doručovány v nějakém stejném pořadí na všech uzlech, čas odeslání není nutně určující, řešeno buď sekvencérem (centralizace) nebo dvoufázovým dist. algoritmem (?)
        - Atomický multicast = spolehlivé sekvenční doručení
    - Kauzální uspořádání - kauzálně vázané zprávy jsou v pevném pořadí (správném), ostatní libovolném, kauzální závislost viz logické hodiny (obsah kauzálně závislé zprávy MŮŽE být ovlivněn závislostí, ale nemusí)
    - Source order - zprávy od jednoho uzlu se navzájem nepomíchají

- Kauzální doručování - dvě kauzálně závislé zprávy budou doručeny v kauzálním uspořádání ve všech procesech kam přijdou obě
- K zaručení kauzálního uspořádání slouží vektorové hodiny
    - Délka vektoru = počet procesů ve skupině
    - Každému procesu patří jeden slot, který indikuje kolik poslal zpráv (a tedy i jejch id)
    - Notace VTa(p)[i] = Hodnota v ítém slotu vektorových hodin pro proces (nebo zprávu) p vzhledem ke skupině a (volitelné)
    - Obecně o práci s VT
        - Při startu je vektor nulový
        - Před odesláním zprávy si zvednu hodnotu svého slotu
        - Při doručování si aktualizuju svůj vektor tak, že beru po složkách maxima ze svého vektoru a vektoru zprávy (ovšem kdy konkrétně zprávu doručíme není dáno)
        - Platí že vektory dvou zpráv se vždy liší
    - Porovnáváme po složkách, tedy VT1 <= VT2 ⇔ všechny složky VT1 <= VT2 (Neplatí že by všechny vektory šly porovnat!)
    - Ostré porovnání vyžaduje <= a existenci aspoň jednoho prvku VT1 který je menší než odpovídající prvek VT2
- Doručovací protokol pro jednu skupinu
    - Při odeslání z procesu p s ID i zvedáme VT(p)[i]
    - Při přijetí zprávy od procesu s ID i na procesu p pozdržíme doručení do té doby, než bude platit
        - VT(m)[i] = VT(p)[i] + 1 (známe všechny předchozí zprávy od odesílatele)
        - VT(m)[k] <= VT(p)[k] pro k != i (známe všechny zprávy ostatních procesů na kterých je zpráva kauzálně závislá, potažmo známe zprávy novější)
        - Při doručení upravíme hodiny podle výše zmíněného algoritmu
- Překrývající se skupiny
    - Idea: Nasadíme masky a maticové hodiny - sloty procesů u VT skupiny do které nepatří jsou zamaskované (*)
    - Do zprávy evidujeme VT pro všechny skupiny kterých je odesílatel členem (měníme VT jen skupiny kam odesíláme) - podle ilustrace tam ale dáváme VT uplně všechny
    - Při přijetí pozdržím dokud neplatí
        - VTa(m)[i] = VTa(p)[i] + 1 (známe všechny předchozí zprávy do této skupiny od odesílatele)
        - VTa(m)[k] <= VTa(p)[k] pro k != i, k ze skupiny a (známe všechny zprávy ostatních procesů do cíleové skupiny)
        - VTb (m) <= VTb (p) pro všechny skupiny b do kterých příjemce (a odesílatel) patří (kauzalita vůči zprávám z jiných skupin)
    - Po doručení klasicky aktualizuju VT procesu
- Distribuovaný total order (sekvenční doručování)
    - Stačí nám skalární hodiny

    - Při příjmu zprávy pošlu potvrzení s časem přijetí odesílateli
    - Odesílatel po přijetí všech potvrzení vezme maximální čas přijetí mezi všemi příjemci, a ten prohlásí víceméně za oficiální čas doručení, zašle ostatním
    - Ostatní doručují zprávy podle oficiálního času doručení
    - Jedná se přesně o případ kdy principielně centralizovanou věc řešíme distribuovaně => bullshit, sekvencér je efektivnější a mocnější
- Spolehlivé doručování
    - Nestačí nám posílat zprávy spolehlivým kanálem, neb nám můžou mezitím umřít příjemci nebo odesílatel
    - Dá se řešit transakcemi, což sice funguje, ale je to příliš striktní a zanáší to další nepříjemné důsledky
    - V zásadě Flooding (spolehlivý, neefektivní) vs. Potvrzování
    - Potvrzování - uchováváme si zprávu dokud nevíme, že ji dostali všichni co ji dostat měli, přeposíláme když zjistíme crash, všem o kterých nevíme doručení
        - Ale jak zjistit kdo už zprávu přijal?
- Alg Trans
    - Protokol pro spolehlivé kauzální doručování
    - Základní idea - transitivní potvrzování, Ack(m) potvrzuje zprávu a zprávy které jí kauzálně předcházejí
    - Stabilní zpráva = přijata všemi členy skupiny
    - G = graf kauzality, vrcholy jsou zprávy, hrany jsou opačné kauzální závislosti (vztah “potvrzuje”)
    - Pro větší efektivitu potvrzení přibalujeme do standardních zpráv (piggybacking)
    - Z potvrzení poznám co mi chybí (potvrzení něčeho co jsem v životě neviděl)
    - Implementačně: mám ack_list (to co jsem už doručil), nack_list (vím o jejich existenci, ale ještě jsem je nedostal), undelivered_list (zprávy které mám, ale ještě nemůžu doručit kvůli kauzalitě), G (graf kauzality), Gp (část G obsahující zprávy které jsem přijal, ale nejsou stabilní)
    - Odeslání zprávy
        - Do zprávy (m) dám svůj ack a nack
        - Do acku a grafu kauzality přidám m (ack list vyčistím)
        - Rozešlu m
    - Přijetí zprávy
        - Rebroadcastuju všechny zprávy co jsou v přijaté zprávy evidovány jako nack, a které jsou nestabilní (jsou v Gp)
        - Pokud jsem už zprávu přijal ale není stabilní, exit
        - Pokud odesílatel potvrzuje zprávu o které nevím (není v Gp) , přidím si ji do nack_listu
        - Pokud přijatá zpráva byla v nack_listu, odeberu ji (už ji znám)
        - Přidám si zprávu do grafu kauzality a undelivered_listu
        - Všechny zprávy z undelivered_listu pro které jsem doručil kauzálně předcházející zprávy můžu doručit (a odebrat z undelivered)

        - Všechny zprávy, které ještě nejsou stabilní (jsou v Gp), mám všechny jejich kauzální předky a neexistuje jiná zpráva v Gp která by je tranzitivně potvrzovala přidám do ack_listu
        - Všechny stabilní zprávy odeberu z Gp
    - Causal line - odděluje to pro co už mám všechny kauzální závislosti od zbytku
    - Má slabinu v havárii procesu - vyústí v neomezenou paměťovou složitost (proč?), dá se řešit protokolem na změnu členství ve skupině
    - Všechny procesy vytvářejí stejný G, ovšem s jiným postupem přidávání

Virtuální synchronie

- Pojem Group view - množina uzlů ve skupině
    - Značení L (globální pohled), L_p (pohled procesu p), L^X (pohled verze X)
- Virtuální synchronie
    - Pokud zašleme zprávu skupině před změnou pohledu na další verzi, buď
        - Všechny uzly z L^X doručí zprávu m před změnou na L^{X+1}
        - Žádný uzel provádějící změnu na L^{X+1} zprávu nedoručí
    - Ve formálnějších definicích je bordel
    - Je potřeba dodržet konzistenci - uzly jsou si navzájem ve skupinách (p z Lq => q z Lp)
        - Udržují stejný L
        - Instalují ve stejném pořadí
- Transis alg.
    - Postavený na Trans a ISIS (????????)
    - Spolehlivý kauzální multicast + členství
    - Protokol je monotónní
        - Nejde dosáhnout d. konsensu (wat? proč?)
        - Paranoia - k vyloučení stačí jedno podezření
        - Jednosměrnost - nejde se samovolně vrátit (jen explicitně znovu připojit)
    - Idea: Stojí na změnách pohledů, doručujeme v rámci pohledů
    - Podezření na havárii => broadcast a rebroadcast zprávy FAULT(q)
    - Kauzální hranice pohledu - doručíme předcházející, pozdržíme následující
    - Průběh algoritmu je stejný jako v Trans - FAULT jsou speciální zprávy, také je potvrzujeme, tak je zahrnujeme do G
    - Zprávy před FAULT doručíme
    - Zprávy od havarujícího procesu před FAULT doručíme
    - Zprávy od havarujícího procesu konkurentní k FAULT nebo kauzálně po FAULT zahodíme
    - Vlastně řežeme podle causal line dané FAULTY
        - Při více faultech najednou to není uplně line
- ISIS
    - *Zjednodušení: při příjmu zprávy o instalaci nového pohledu přepošle všechny nestabilní zprávy a za nima flush; používá zajištění kauzality (maticové hodiny),*

takže jakmile mu přijde flush od všech procesů, má hotovo, (protože kvůli kauzalitě všechny zprávy před flushema už musel zpracovat)

- Stejný význam jako Trans, ale místo tranzitivního kauzálního potvrzování používá maticové hodiny
- Maticové hodiny jsou vektorové hodiny všech procesů - tedy co víme o doručování zpráv jiným procesům
- MTp[j][k] - co víme o doručování zpráv od k pro j
- Při příjmu na procesu j od procesu i aktualizuju
  - MTp[j][i] = VTm[i] - standardní aktualizace vektorových hodin příjemce podle VT zprávy
  - MTp[i][*] = VTm[*] - aktualizace informací o doručování zpráv odesílateli od ostatních procesů
- Stabilní zprávy - o všech procesech víme z odpovídajících slotů MT že jim došly
- Trans je svým způsobem komprese MT (místo abychom posílali celou, stavíme postupně)
  -
- Optimista - nízká režie při normálním provozu, vysoká při chybách
- Změny pohledů jsou v ISIS řešeny také, zprávy udržujeme dokud se nestanou stabilní
  - Změna pohledu ⇔ řachnul uzel (možná i se připojil nový)
- Při příjmu zprávy o instalaci nového pohledu:
  - Přepošleme nestabilní zprávy
  - Flush zpráva - oznamujeme potvrzení instaluje
  - Čekáme na Flush od všech ostatních
- Zároveň udržujeme informace o havarovaných procesech, přeposíláme se zprávami, sjednocujeme
- Havarované zprávy zahazujeme (všechny?)

Kauzalita, logické hodiny.

- Neexistují globální hodiny - i kdybychom všechny uzly synchronizovali podle UTC, nevyhneme se nepřesnostem v řádech miliónů instrukcí
- Můžeme je zkusit synchronizovat mezi uzly
  - Christian - pasivní UTC timeserver, ostatní se ho periodicky ptají
    - Komunikace a zpracování taky něco trvá - k odpovědi přičítáme (Příchozí čas - odchozí čas - doba zpracování) / 2
    - Tedy odhadujeme dobu přenosu informace přes kanál
    - Nepřeřizovat víc uzlů najednou
  - Berkeley - aktivní timeserver - ptá se ostatních, počítá průměr, seštělovává
  - Distribuovaný algoritmus, Intervalový čas snad není třeba umět
- Logické hodiny
  - Postaveno na idee od Leslieho Lamporta - ve skutečnosti nám nezáleží na tom, jaký je skutečný čas, ale jaké je pořadí operací
    - Netřeba synchronizovat to co spolu nekomunikuje

- Kauzální závislost = jedna událost mohla ovlivnit výsledek druhé
  - Pokud existuje proces (výpočetní uzel) v rámci kterého se událost e1 stala před e2, je e2 kauzálně závislá na e1
  - Pro všechny zprávy musí platit že přijetí zprávy (kdekoliv) je kauzálně závislé na jejím odeslání
  - Platí transitivita
  - Značíme a -> b
- Události e1, e2 konkurentní ⇔ e2 není kauzálně závislá na e1 a vice versa
- Logické hodiny jsou takové, že kauzální závislost b na a implikuje čas události b menší než čas události a, dle hodin:
  - Pokud a -> b, pak C(a) < C(b)
- Leslieho sync logických hodin
  - Při odeslání zprávy se do ní přihodí lokální čas
  - Při přijetí se testuje, zda je splněna podmínka logických hodin, případně se opraví na nebližší možnou hodnotu
  - Pro zúplnění můžeme zavést i podmínku, že v případě stejného logického času dvou událostech v jiných procesech má vyšší proces vyšší čas
    - C(a) = C(b) && Pi < Pj => C'(a) < C'(b)
    - Byrokratické uspořádání
    - Nemám nejmenší páru k čemu to je, ani jak bych to dělal
- Bohužel neplatí, že by C(a) < C(b) implikovalo a -> b

Distribuované synchronizační algoritmy (vzájemné vyloučení, volba koordinátora, distribuovaný konsensus, detekce globálního stavu).

Vzájemné vyloučení

- *Neexistence sdílené paměti => nemáme klasická synchronizační primitiva*
- *Centralizované řešení*
- *Permission-based*
    - *Časové značky*
        - *Lamportův algoritmus*
        - *Ricart-Agrawala*
    - *Volby*
        - *Maekawa*
- *Token-based*
    - *Token ring*

Volba koordinátora

- *Bully algoritmus (pozor na předpoklady)*
- *Invitation*
- *Kruhový algoritmus*

Distribuovaný konsenzus

- *Problém dvou armád (okamžitá domluva, jinak fail)*
- *Problém byzantských generálů (odhalení zrádců = nemocných uzlů)*

Detekce globálního stavu

- *Řezy*
    - *konzistentní, nekonzistentní*
    - *konzistentní stav*
- *Algoritmus detekce (to posílání značek a signálů)*
- *Speciální případ: ukončení distribuovanýho výpočtu*
    - *Algoritmus Dijkstra-Scholten*
        - *strom, DAG, obecný graf*
    - *Značkový algoritmus*
        - *speciální případ detekce globálního stavu*

MutEx

- Základní problém - nemáme sdílenou paměť, tedy ani semafor
- Existují různé způsoby jak zajistit mutex v distribuovaném prostředí
    - Centralizovat - všichni si někomu říkají o povolení
    - Permission-based - časové značky nebo volby
    - Token-based - předáváme si token, ten nám dává právo k práci se sdíleným zdrojem
- Centralizace
    - Máme jeden server s frontou (říkáme mu sekvencér)
    - Klasický formát request + ack / deny + notifikace o uvolnění
    - Ideově nevhodné

    - Single point of failure - v případě smrti serveru je třeba restore do předchozího stavu, volby nového sekvencéru, etc.
    - Musíme řešit i výpadek klienta, jinak hrozí vyhladovění
- Lamport
    - Stojí na časových razítkách z logických hodin
    - Když chci vstoupit do kritické sekce, pošlu žádost se svým časovým razítkem, a pak čekám až budu mít potvrzení (potvrzují, že přijali můj request, ne že můžu do sekce) od všech procesů a všechny žádosti co mám ve frontě budou novější než ta moje
    - Do kritické sekce můžu vstoupit v okamžiku, kdy mám ode všech potvrzení novější než kdy jsem žádal, a neznám žádnou starší žádost na vstup
    - Při přijetí uvolnění smažu starší žádosti uvolňujícího procesu z fronty
    - Komunikační složitost 3(n-1)
- Ricart & Agrawala
    - Modifikace Lamporta
    - Rozlišuje jen requesty a acky
    - V případě snahy vstoupit do CS pošleme všem request, a čekáme na všechna potvrzení
    - V případě přijetí žádosti
        - Pokud nejsem v CS a nechci být, pošlu ack (povolím vstup)
        - Pokud jsem, dám si to do fronty a pokračuju
        - Pokud nejsem v CS a chci být, porovnám razítka žádosti - pokud je cizí žádost starší, pošlu ack (vzdám se), jinak do fronty
    - Po opuštění CS pošlu ack procesům ve frontě
    - 2(n - 1)
- Volby obecně
    - Snažíme se získat hlasy ostatních
    - Každý má jeden hlas
    - Potřebuji víc hlasů než ostatní
    - Základní otázka: jak SPRÁVNĚ hlasovat a počítat hlasy
        - Naivní implementace může vyústit v deadlock
- Maekawa
    - Klíčová idea: Voting district
    - Každý proces potřebuje pro vstup všechny hlasy ze svého okrsku
    - Každý proces ve více okrscích (jako volič)
    - Základní otázkou je jak sestavit okrsky
        - Každé dva procesy musí mít jednoho společného voliče (korektnost)
        - Velikost všech okrsků je stejná
        - Všechny procesy ve stejném počtu okrsků
    - Rozdělení do okrsků - optimální je velikost okrsku = O(sqrt(počet okrsků))
    - Maekawa bere matici všech procesů
        - Okrsek procesu je tvořen sloupcem a řádkem ve kterém se vyskytuje
        - Každé dva procesy mají společného voliče (ve skutečnosti je mají dva)

        - Stále hrozí deadlock! Dva společní voliči se mohou přiklonit každý k jinému procesu
            - Řešením je odebírání hlasů na základě časových razítek
            - Když přijmu žádost o svůj hlas
                - Pokud mám volný hlas, potvrzuji zvolení
                - Pokud jsem již dal hlas někomu jehož žádost je starší, zařadím do fronty a zahlasuju až po uvolnění
                - Pokud jsem již dal hlas někomu jehož žádost je novější, pošlu mu REJECT, tedy snahu odebrat mu hlas
                - Reakce na REJECT je různá podle toho jestli už jsem začal provádět CS, nebo ne - pokud ano, už to dodělám, pokud ne, vrátím hlas
- Stromové algoritmy
    - Agrawal & El Abbadi - zakořeněný strom, volby, potřebuju hlasy z cesty od kořene k listu, nahrazujeme podstromy při výpadku
    - Raymond - token, kostra, pointer směrem k token holderovi
- Suzuki & Kasami - přenos zprávy s povolením, žádost odpovídá broadcastu, zpráva obsahuje frontu, umožňuje prioritní řazení
- Token ring - přenos token, je třeba řešit výpadek uzlu
- POZOROVÁNÍ: Algoritmy jsou to sice cool, ale ve skutečnosti mají výrazně větší komunikační složitost i problémy než centralizované řešení
    - A to proto, že se snažíme řešit inherentně CENTRALIZOVANÝ problém DISTRIBUOVANÝM algoritmem -> nemá to cenu !!!!!

Volba koordinátora

- Obecně - snažíme se zvolit nějaký uzel pro nějakou unikátní roli
- Bully algoritmus
    - Předpokládáme omezenou dobu přenosu a zpracování zprávy
    - Stejně jako existenci spolehlivého detektoru havárie (2Tm + Tp; nejspíše čas přenosu a zpracování)
    - Což je poměrně silný požadavek (v reálu neplatí)
    - Procesy jsou očíslované, pokud se rozhodnu volit nového koordinátora, zašlu zprávu všem vyšším kolegům
    - Když mi někdo odpoví, vzdám se
    - Když nepřijde nic, vyhrál jsem a stávám se kordinátorem
    - Přijde mi výzva k volbě, odpovím a vyšlu žádost všem vyšším
    - Volba se provede ve dvou kolech (prvním se vytvoří kandidáti), v případě že proces odpoví na první výzvu, ale po nějakou dobu se neprohlásí za vítěze, vyvoláme nové volby
    - V případě že proces slyší o výhře procesu s nižším ID, okamžitě vyvolá nové volby -> bully
    - Při porušení předpokladů hrozí více koordinátorů
- Invitiation

    - Reálně použitelný - založen na idee že nejde spolehlivě detekovat havárii
    - Koordinátor je dán pro skupinu, skupiny lze štěpit a slučovat (typicky jako důsledek výpadku / restartu)
    - Koordinátor pravidelně posílá výzvu AYC (Are you coordinator?)
    - Pokud AYC dojde koordinátorovi - sjednotí skupiny do skupiny vyššího koordinátora
    - Pokud člen dlouho nedostane AYC od svého koordinátora, prohlásí se za koordinátora (skupiny velikosti 1) a pošle pozvání ostatním skupinám
    - Konzistence pouze vzhledem ke skupině
- Kruháč (Chang & Roberts)
    - Uzly organizovány pointery do kruhu
    - Když se rozhodnu volit, pošlu zprávu následníkovi (se svým ID)
    - Procesy zapisují do zprávy nejvyšší ID živého procesu
        - A podle Wiki tam zapisují i svou účast (aby poznali zda už na tuto výzvu reagovali)
    - Když se mi zpráva vrátí, vím kdo bude koordinátor
    - Pak se oznamuje, opět kruhem

Distribuovaný konsensus

- Distribuované procesy se musí na něčem shodnout
- Dva ilustrační problémy
    - Problém dvou armád
        - Dva generálové se MUSÍ shodnout na času útoku
        - Nespolehlivá komunikace
        - Neexistuje striktně správně řešení za použití pouze nespolehlivého kanálu
        - Prakticky se dá řešit agresivně (pošlu mnoho kurýrů a doufám) nebo pomocí pravděpodobnostní strategie (pošlu N zpráv, N1 jich projde, N2 odpovědí dostanu zpátky - stavíme na znalosti pravděpodobnosti průchodu)
    - Problém byzantských generálů
        - Uzly mohou havarovat a chovat se zákeřně -> zrádci
        - Máme spolehlivou komunikaci
        - Všichni loajální (nehavarovaní) generálové se musí rozhodnout stejně
        - Rozhoduju na základě informací od ostatních
        - BÚNO - jeden generál, zbytek důstojníci (proč asi? jak to převést na konsensus?)
        - Rozkaz vydán na základě většiny
        - Cílem je
            - Aby všichni loajální rozhodli stejně
            - V případě loajálního generála předá loajální důstojník přesně jeho rozkaz

        - Řešení ne vždy existuje - v případě tří procesů nejde rozlišit zrada generála a lieutenanta (k loajálnímu důstojníkovi se vždy dostane konfliktní informace)
            - Obecně neexistuje řešení pro n <= 3m, kde n = počet uzlů a m = počet zrádců
        - Pro 4 uzly řešení existuje - zradu generála buď poznáme, nebo aspoň rozhodneme většinově, zrada lieutenanta nám neovlivní většinu
- Obecně klasifikujeme konsensus do tří kategorií, které jsou přibližně ekvivalentní
    - Byzantský konsensus (1 iniciální hodnota -> 1 hodnota) - generálové, tedy máme iniciální hodnotu jednoho uzlu a ta je rozeslána, loajální uzly se musí shodnout na stejné hodnotě, při loajálním iniciátorovi musí být výsledná hodnota identická s iniciální
    - Konsensu (n iniciálních hodnot -> 1 hodnota) - všichni loajální se musí shodnout na společné hodnotě, pokud jsou všechny loajální iniciální hodnoty shodné, musí být výsledkem
    - Interaktivní konzistence (n iniciálních hodnot -> n hodnot) - všichni loajální se musí shodnout na společném vektoru, položky vektoru odpovídající loajálním uzlům musí obsahovat jejich hodnotu

Detekce globálního stavu

- Snažíme se zjistit celkový stav distribuovaného systému z hlediska nějakého parametru (např. ukončení výpočtu, existence deadlocku, atp.)
- Speciálním případem je detekce ukončení distribuovaného výpočtu
    - Výpočet vizualizujeme grafem s hranami představujícími komunikační kanály
    - Kromě standardních kanálů máme i tzv. signální hrany vedoucí opačným směrem - indikují ukončení
    - Algoritmus Dijkstra-Scholten (DS)
        - V případě stromu je situace jednoduchá - listy posílají při ukončení signál otci, otec signalizuje když ukončí on a všichni syni
        - DAG - signály nejsou bool ale int, rozdíl #zpráv a #signálů = deficit, při ukončení pošleme každou vstupní hranou nahoru tolik signálů aby deficit byl 0
        - Graf - postavíme dynamicky kostru, uzel považuje za otce toho, od koho mu přišla první zpráva (realistické), končí se tak, že pošlu signál všude krom otce (signalizuju že na mě nikdo nemusí čekat), počkám na signály od potomků (nemusím na nikoho čekat), pošlu signál otci (moje část hotova)
    - Značkový (TM) algoritmus
        - Značka = speciální zpráva (někdy myslím označovaná taky jako jed - poison)
        - Iniciátor pošle značku do výstupních hran
        - Při přijetí první značky buď propaguju značku (když jsem ochoten končit), nebo zamítnu

        - Při přijetí další značky buď signalizuju (už jsem jednou propagoval), nebo zamítnu
        - Při přijetí zamítnutí zamítnu vejš
        - Když mám všechny signály od potomků, signalizuju vejš
        - Speciální případ detekce GS
- Rozdíl při detekci globálního stavu od DS je podle mě to, že u DS je detekce stavu iniciována uzly samotnými, ne externě… navíc je dost možný, že DS není stavěnej na to, že by se uzly mohly vrátit do stavu výpočtu potom co zasignalizujou (nebo je?)
- Detekce GS
    - Proč to není triviální - na všech procesech víme stav pro určitý čas, jenže "čas" testování stavu je různý napříč procesy - může se nám stát, že stavy dohromady reprezentují konstelaci, která ve skutečnosti nikdy nenastala (false positive, detekce falešného deadlocku)
    - Musíme tedy nějak zajistit, abychom testovali stav konzistentní
    - Řez c definujeme tak, že nám dělí všechny události v systému na Pc (past) a Fc (future), Pc a Fc pochopitelně disjunktní
    - Řez je konzistentní ⇔ a -> b && a z Fc implikuje b z Fc (minulost nelze ovlivnit budoucností, cokoliv kauzálně závisí na budoucnosti je taky budoucnost)
    - Stav procesu definujeme jako nějakou množinu událostí které se v procesu udály
    - Konzistentní stav S je takový stav, že S = Pc pro nějaký konzistentní řez c
        - Pokud S je konzistentní stav, pak S + e (událost) je konzistentní stav (proč?), značím $S \rightarrow^{e} S'$ (S' je dosažitelný z S)
        - s = (e1, e2…) sekvence událostí, $S \rightarrow^{e1} S1 \rightarrow^{e2} \dots \rightarrow^{en} Sn$, s říkáme rozvrh, značíme $S \rightarrow^{s} Sn$
        - $S \rightarrow^{s} Sn$ => S podmnožinou Sn
    - Algoritmus detekce GS
        - stav uzlu - množina přijatých a odeslaných zpráv, stav kanálu = množina odeslaných ale nedoručených zpráv
        - Iniciátor pošle značku
        - Při příjmu první značky
            - Zapamatuju poslední přijaté a odeslané zprávy → stav uzlu v okamžiku řezu
            - Stav příchozích kanálů nastaví na prázdný
            - Výstupními kanály pošle značku
        - Při přijmů zpráv od uzlů, od kterých ještě nepřišla značka (nejsou ještě v řezu) si zapamatuju čísla zpráv → to budou zprávy přes hranici řezu
        - Při přijmů značky od dalšího uzlu - nastavím stav kanálu jako to co přišlo od přijetí iniciální značky (to by měly být ty zprávy z past do future)
        - Konec algoritmu - odevšud mi přišla značka
    - Stavy uzlů a kanálů definují konzistentní stav
        - Stavy kanálu definují přesně zprávy které kříží řez (z past do future)

Procesy v distribuovaném prostředí, migrace, zablokování. - PDS

Procesy v distribuovaném prostředí

- *Smysl (sdílení, load ballancing)*
  - *možná použít nějaký okecávačky z virtualizace?*
- *Klasifikace*
- *Vzdálené spouštění - transparentní, přenos kódu a dat a vytvoření prostředí*
- *Alokační algoritmy*
  - *Hierarchický*
  - *Bidding - nákup a prodej výpočetní síly*
  - *Up-down - vybírám proces z fronty, jehož vysílající uzel má nejméně trestných bodů*
- *Co když uzel přestane být volný?*
  - *doběhnutí / zabití / čas na uložení / **migrace***

Migrace

- *Korektní a transparentní!*
- *Load ballancing*
- *Přenos stavu*
  - *Zmražení, přenos, spuštění*
  - *Pre-copy, Copy-on-reference*
- *Reziduální dependence*
  - *Nechat na původní uzlu něco, co je obslouží*
- *Nějaký ukázkový migrační systém*
  - *MOSIX*
- *Load ballancing = volba kdy koho kam zmigrovat, aby bylo zatížení rovnoměrné*
  - *Párový algoritmus*
  - *Vektorový algoritmus*
  - *Centralizované řešení*
  - *Lokální algoritmy (při velkém zatížení zkouším udat své úlohy jinde)*

Zablokování

- *Obecně správa prostředků*
- *Centralizované řešení - resource manager*
- *Deadlock*
  - *Wait-for-graph*
  - *Korektnost (dedlock umíme vždy detekovat, nedetekujeme falešný deadlock)*
  - *Centralizované řešení (s kauzálním doručováním)*
  - *Path-pushing*
  - *Edge-chasing*

- Smyslem správy procesu v DS je
  - Sdílení výpočetní síly
  - Load balancing
  - Vzdálená operace

    - Synchronizace a evidence stavu
- Vzdálené spouštění procesů
    - Potřebujeme najít uzel na kterém úloha poběží, způsob měření zátěže, přenést na cíl kód a vytvořit prostředí odpovídající domovskému… potenciálně i musíme řešit co dělat když uzel přestane být volný
    - Iniciální komunikace jde typicky přes centralizovanou komponentu - registr - ke které se volné uzly hlásí, a domovské prosí o přidělení
- Alokační algoritmy - klasifikace
    - Deterministické / heuristické - máme údaje o procesech?
    - Optimální / suboptimální
    - Co optimalizujeme (CPU / čas odpovědi / komunikační složitost…)
    - Migrační / nemigrační (můžu přenést rozběhlý proces?)
    - Homogenní / heterogenní prostředí (různý HW, speciální požadavky)
    - Centralizované / dist (typ algoritmu)
    - Lokální / globální (podle jakých informací se rozhoduje o spuštění)
    - Kdo iniciuje vzdálené spuštění (odesilatel, příjemce, oba?)
- Alokace procesorů
    - Hierarchický alg - rozděluje se ve skupinách, výše se jde jen při neúspěchu
    - Distribuovaný heuristický - ?
    - Deterministický grafový - snaha o minimalizaci komunikace, toky v sítích
    - Bidding - procesy kupují výpočetní sílu
    - Up-down
        - Cílem je stejnoměrné sdílení výkonu
        - Máme koordinátora, který spravuje uzlům trestné body
        - Trestné body se udílí při významné akci (vytvoření a ukončení procesu, tik)
        - Za každý proces co mi běží na jiném uzlu dostanu plus body (tedy jsem znevýhodněn když pro mě už někdo pracuje)
        - Za každý můj neuspokojený požadavek mínus body (tedy čím déle a s více úlohama čekám, tím dřív se dostanu na řadu)
        - Nic z toho -> posun směrem k nule
        - Když se uvolní uzel, vybereme proces jehož majitel má nejméně trestných bodů
- Migrace procesů
    - Mělo by být korektní a transparentní
        - Korektnost - ostatní nejsou migrací (příliš) ovlivněni, po ukončení je stav stejný jako bez migrace
        - Transparentnost - procesy o migraci netuší
    - Smysl je v load balancingu optimalizaci, přemístění
    - Problémy
        - Přenesení rozdělaného stavu (výpočtu)
        - Přenesení adresového prostoru
        - Vazby a komunikace s ostatními procesy

        - Reziduální dependence
        - Vícenásobná migrace
    - Implementace přenosu
        - Vyjmutí ze stavu, zmražení, speciální stav?
        - Oznámení příjemci, alokace
        - Přenos stavu
        - Přenos kódu a adresového prostoru
        - Přesměrování a doručení zpráv
        - Dealokace procesu, vyčištění
        - Vazby na nové jádro, nastartování
            - Přesunutí části stavu s procesem (např. paměti)
            - Forwardování požadavků (např. IO)
            - Použití odpovídajícího prostředku u cíle - fyzické paměti, disku, atp.
        - Dokončení přenosu vazeb
            - Dočasná reziduální dependence, přesměrování zpráv
    - Implementace přenosu VM
        - Buď přeneseme všechno při migraci
            - Eliminace reziduálních dependencí
            - Trvá, mnohdy zbytečné
        - Pre-copy - přenášíme ještě před migrací
            - Zmražení netrvá dlouho
            - Aktivní data můžeme muset přenést víckrát
        - Copy on reference
            - Přeneseme jen klíčová data potřebná pro běh (registry, kanály)
            - Ostatní stránky označíme jako neplatné, musí se znovu načíst ze zdroje
            - Nutnost evidovat zdroj stránky
            - Reziduální dependence?
    - Implementace zprávové komunikace
        - Přesměrování dočasné vs. trvalé
        - Migrace kanálů - různá podle toho odkud kam migrujeme
            - Rozlišujeme lokální (meziprocesová komunikace?) a vzdálené (síť) kanály
            - Sekvence vytvoření - přepojení - zrušení původního
- Ukázkové migrační systémy
    - DEMOS/MP
        - Linky spojené s procesy, migrace obsažena v jádře OS
        - Transparentní, jednotná komunikace procesů
        - Standardní výše popsaný postup s pár odlišnostmi
            - Při přesunu stavu si cílový uzel “tahá proces k sobě”
            - 3 typy zpráv - před migrací přeneseny současně s migrováním, zprávy se starou adresou -> forward, zprávy s novou

            - Na zdroji se likviduje všechno kromě nové adresy (kvůli fw)
    - Charlotte
        - Linky nezávislé na umístění procesů, přesouvání linků
        - Migrační politika přímo v procesu, migrace zajištěna jádrem
        - Směřuje k maximální fault toleranci (eliminuje reziduální dependence)
        - Migrační strategie na základě statistiky o počítači a procesu
        - Migrace probíhá ve fázích
            - Negotiation - Výměna informací mezi procesy, proces na zdroji volá Migrate out, na cílový uzel Migrate in (?)
            - Přesun - přesun obrazu, nastavení linků (přesměrování u počátku), přesun stavu
            - Cleanup
    - V
        - Migrují tzv. Logical host - adresový prostor, potenciálně několik procesů
        - Pre-copy
        - Při migraci se dělá pre-copying a pak odmražení a přesměrování odkazů (pomocí broadcastu adresy logical hostu)
    - MOSIX
        - Dist. UNIX, dnes linux (*jedna instalace linuxu rozložená na více PC*)
        - Netřeba nějak speciálně upravovat aplikace, funguje samo, jako na SMP
        - Poskytuje jednotný systémový obraz (transparentnost)
        - Load-balancing
        - Jako jeden z mála je reálně nasazován
        - Migrace
            - Cílový uzel zjistí jestli si může migraci dovolit
            - Předělá komunikační kanály
            - Nastaví paměť
            - Čeká na doručení procesu, pak ho nastartuje
    - Sprite
        - Předpokládá nevyužité uzly
        - Cílem rychlost a transparence
        - Migrace
            - Migraci provádí jádro, sběr statistiky a řízení proces (nemělo to bejt kurva transparentní??)
            - Původně zámšleno tak, že volání jádra jsou forwardovány domů
            - Transparent vs. reziduální dep.
            - VM - kombinace copy all a copy on reference (JAK Zavorale???)
            - Jednotná migrace různých entit - každá součást stavu má 4 rutiny - pre-mig, encapsulation, de-encapsulation, post-mig
    - T4
        - Snaha o efektivitu normálního OS, prostředí pro použití jako DS

Load balancing

- Vytěžování procesorů a jiných zdrojů, dynamicá migrace při zachování konzistence

- Řízení - vybrat kdy migrovat, co a kam
- Při zachování konzistence
- Algoritmy
    - Párový (Bryant & Finkel)
        - Vzájemně se vyvažující páry (dynamicky vznikající)
        - Request o vytvoření páru - odmítnutí / vytvoření páru / migrace (?)
        - Migraci řídí zatíženější uzel, rozhoduje na základě míry vylepšení $k_i = A_i / (B_i + AB_i)$
        - Migruje jen když je zlepšení “významné”
    - Vektorový
        - Vektor zátěže L, L(0) jsem já
        - Periodicky zjišťuju vlastní zátěž a náhodně zašlu půlku vektoru (horní?)
        - Při příjmu zkombinuju obě horní půlky (střídavě je proložím)
        - Což vyustí v zahození starších dat
        - K zátěži se přičte komunikační režie (jak??)
    - “Inteligentní” algoritmy
        - Bidding, stochastic learning automata, bayesian decision theory
        - Příliš komplikované
    - Praktické algoritmy
        - Centralizovaný - koordinátor, může být i rozdělen na menší skupiny + hierarchie
        - Lokální - znám jen lokální zátěž a prahovou hodnotu, požádám okolí o pomoc, když jsou nějací pod prahem, vyberu nejlepšího a tam migruju

Deadlocky

- Vzniká jako problém správy prostředků
    - Zavoral tvrdí, že distribuovaná správa je nepoužitelní kvůli distribuovanému vyloučení procesů (proč?)
- Musí být korektní - každý deadlock jednou detekujeme, nesmíme mít false positives (včetně phantom deadlocků - vnořené)
- Základem klasicky wait for graph (WFG)
- Metody v DS se liší hlavně způsobem vytváření WFG
- Centralizované řešení
    - Uzly přenášení informace o využití zdrojů po změně stavu / pravidelně / na požádání
    - Může mít falešný deadlock kvůli špatnému pořadí zpráv
        - Dá se řešit kauzálním doručováním (vážně?)
    - Dá se rozšířit na hierarchii
- Path-pushing
    - WFG distribuovaný, uzly mají jen lokální část + proxy uzly
    - Spousta algoritmů nekorektních kvůli phantomovi
- Edge-chasing
    - Posíláme podél hran WFG zprávičky (probes)
    - Pokud se zprávička vrátí, jsem v prdeli

    - Nemá cenu všechno vzdát, stačí najít kandidáta na eliminaci - zpráva dělá při oběhu
- Detekce globálního stavu
    - Když je deadlock, je i v konzistentním řezu

Distribuované sdílení paměti, konzistenční modely (a konkrétní technologie DDS, JavaSpaces). - PDS+Wiki?

Distribuované sdílení paměti

- *Pro paralelní výpočty*
    - *multiprocesory, multicomputery*
- *Úrovně*
    - *MMU (cache blok, přes bus)*
    - *OS (stránky)*
        - *Sekvenčně konzistentní (vlastník writeable, ostatní readonly)*
        - *Kauzálně konzistentní*
    - *Programovací jazyk / Middleware (objekty - sdílené proměnné)*
        - *Taková distribuovaná objektová databáze*

Konzistenční modely

- *Striktní (=linearizability) - všechny zápisy okamžitě viditelné všude*
    - *na uniprocesorech jde*
    - *v distrib. systémech prakticky nedosažitelné*
- *Sekvenční - jednotlivá vlákna jsou sekvenční, proložená jakkoliv, všichni vidí změny ve stejném pořadi*
- *Kauzální - kauzální zápisy nejsou prohozeny*
- *PRAM - zápisy od jednoho procesu nejsou nikde prohozeny*
- *Slow memory - zápisy od jednoho procesu do jednoho místa nejsou prohozeny*
- *Synchronizační proměnná - co to vlastně je? nezdá se to být úplně proměnnou…*
- *Slabá konzistence (přístup do sync. proměnné = “flush”)*
- *Výstupní konzistence (acq, rel) - eager / lazy release*
- *Vstupní konzistence*

- Distribuované sdílení paměti = sdílený adresní prostor přes několik uzlů
- Podpora z OS / knihovna - lze chápat jako rozšíření architektury virtuální paměti
- Jednotka sdílení může být:
    - blok
    - stránka (pevné velikosti)
    - objekt (různých velikostí)
    - tuple?? (tzv. tuple space, př. JavaSpaces)
- Různé způsoby a aspekty řešení
    - HW - Na úrovni MMU (Memory Management Unit) nebo OS (NUMA)
    - SW - Na úrovni OS (Distribuované stránkování) nebo konkrétního jazyka / knihovny (Sdílené proměnné či objekty)

- Větší abstrakce nám vždy poskytuje volnější vazby (loosely coupled), z čehož plynou standardní vlastnosti abstraktnějších systémů, zejména škálovatelnost
- Zajištění koherence v závislosti na konzistenčním modelu
- Konzistenční model nám dává určité garance při čtení a zápisu, které musí implementace poskytovat
- Značení W(x)a - write hodnoty a do proměnné x
- Konzistenční modely
    - Striktní (strict, atomic) konzistence - čtení vrací poslední uloženou hodnotu, writes okamžitě viditelné, ideální, v DS nedosažitelné (není globální čas)
    - Sequential consistency - výsledek stejný jako při nějakém sekvenčním proložení všech instrukcí, všechny procesy vidí stejné pořadí změn, dvě spuštění jednoho programu mohou dát různé výsledky
        - Nejsem si jsit jeslti chápu dobře, ale představuju si že je potřeba zaručit správné pořadí instrukcí z jednoho procesu, ale jinak libovolně prokládat
        - Signatura = výstupy z procesů v pevně daném pořadí
        - Nepříliš dobrá výkonnost, r + w >= t (read, write, čas přenosu paketu)
    - Kauzální závislost
        - Kauzálně vázané zápisy musí být vidět ve stejném pořadí ve všech procesech
        - Analogicky ke zprávám, w = send, r = receive
        - Pokud W(x)a a jinde R(x) W(y)b, W(y)b je kauzálně závislé na W(x)a
        - Složitá implementace (graf závislostí)
    - PRAM (Pipelined RAM)
        - Zápisy prováděné jedním procesem jsou viděny ve správném pořadí všude
        - Jinak libovolné prokládání
        - Snadno implementovatelná
    - Slow Memory
        - Zápisy jedním procesem do jednoho místa ve stejném pořadí všude
        - Jinak whatever
        - Nulová synchronizace
- Konzistenční modely se synchronizační proměnnou - bez SP propagujeme vždy vše, není nutné, málo efektivní, typicky zapisujeme ve smyčce, atp. Necháme tedy řízení propagace na procesu
- Konzistenční modely se synhronizační proměnnou
    - Slabá konzistence - přístup k SP je sekvenčně konzistentní, není povolen dokud neskončí předchozí zápisy, před přístupem k datům se musí dokončit předchozí přístupy k SP
        - Synchronizace těsně před čtením zajistí aktuální verzi dat
        - Nerozlišuje mezi vstupem a výstupem z CS
        - Značíme S - bariéra

    - Výstupní konzistence - Rozlišuje Acq a Rel, před přístupem ke sdílené proměnné musí být ukončeny předchozí Acq procesu, před Rel musí být ukončeny čtení a zápisy do proměnné, Acq a Rel PRAM konzistentní
        - Při správném párování je výsledek ekvivalentní sekvenčně konzistentní paměti
        - Lazy vs. Eager - Lazy propaguje změny až při jiném Acq
    - Vstupní konzistence - Acq není povolen dokud nejsou hotovy aktualizace všech chráněných dat procesu, zápis do SP je povolen jen pokud nikdo jiný nepřistupuje, po exkluzivním přístupu si příští neex přístup k SP musí vyžádat aktuální kopii dat od vlastníka SP (posledního zapisovatele)
        - Data jsou vázána na kritickou sekci a slibuje se, že budou aktuální při vstupu do této kritické sekce
        - Rozlišuje typ přístupu (RW / RO)
        - Nejsem si uplně jistej jak to vlastně funguje :-/
- Konzistence bez SP (transparentní, virtuální paměť) vs. se SP (speciální korektní kód,vyšší výkonnost)

Distribuované stránkování

- Podobné jako virtuální paměť, když přistoupíme k nenamapované stránce, musíme ji načíst (akorát ze sítě)
- Problémy
    - Jak udržovat dat konzistentní
    - Jak je lokalizovat (jak najít stránku)
    - Jak pracovat s kopiemi
    - Jak vybrat uvolňovanou stránku
    - Falešné sdílení - na jedné stránce máme nezávislá data
- Sekvenčně konzistentní dist. stránkování
    - Rozlišujeme různé situace u každé stránky - kdo je vlastník, zda je stránka namapovaná, a s jakými přístupovými právy
    - Pro read nám stačí mít namapovanou readonly kopii stránky, cokoliv silnějšího taky stačí
    - Pro write musím být vlastník a stránka namapovaná pro zápis, různé akce v závislosti na tom jaká je situace
        - Nemám stránku namapovánu - získání kopie
        - Nejsem vlastník - změna vlastnictví
        - Někdo jiný má stránku taky namapovánu - invalidace, donutím o ni znovu zažádat
        - Vlastním readonly namapování - potřebuju povýšit mapování na zapisovací
- Kauzálně konzistentní DS
    - Opět stojí na vektorových hodinách - evidujeme na kterých stránkách může záležet obsah stránky, a ze kterých stránek zná proces data
    - Při zápisu (namapování pro zápis) zvedneme hodnotu VT[i]

    - Při výpadku stránky si ji musím stáhnout od vlastníka a aktualizovat VT procesu podle VT stránky
    - Aktualizace VT procesu mi může zneplatnit stránky - ty pro které mám mapování s menším TS než má proces

Distribuované sdílené proměnné

- Na úrovni knihoven
- ~= distribuovaná databáze
- Typicky model na bázi synch. proměnných
- Typicky výkonnější, nemáme falešné sdílení
- Netransparentní
- Nutnost různých implementací
- Speciální variantou jsou distribuované objekty
    - Flexibilnější, o synchronizaci se můžeme postarat v metodách
    - Automatické generování na základě IDL

DDS

- Publish/subscribe network middleware
    - Produktéři informací dělaj topiky a do nich publikují data
    - Subscribeři se zapisují k topikům
- Events
- Commands
- <TODO víc>

JavaSpaces

- Komunikace sdílením stavu
- JavaSpace je vlastně distribuované úložiště pracující nad záznamy (Entry) s následujícíma metodama
    - write (Entry) - vloží prvek do JavaSpace
    - read (Entry) - přečte prvek z JavaSpace (a nechá ho tam, tedy udělá kopii)
        - parametrem je template (null políčka jsou wildcardy)
    - take (Entry) - přečte prvek z JavaSpace (a odebere ho odtamtud)
    - notify: notifikuje, pokud je objekt splňující zadané podmínky zapsán do spacu
    - S objekty nejde pracovat přímo ve spacu, je třeba take, pak akce, pak write

Distribuované souborové systémy (NFS, AFS, CODA). - OSy+Wiki

- Základní vlastností (a odlišností od file-sharing protokolů) je transparence přístupu, aplikace přistupují k filesystému stejně, jako by byl lokální

NFS (network file system)

- Funguje nad RPC a je bezestavový
- Klient nejprve pošle na server mount příkaz s identifikací složky, jako odpověď je mi file handle na daný adresář (pro něho neinterpretovatelný, ale typicky obsahuje např. číslo filesystému a i-nodu dané složky)

- Poté klient používá stateless příkazy: get/set attrib. read/write, lookup, rm/mk dir, s nimi posílá file handle a relativní cestu
- Důsledky bezstavovosti:
    - Kontrola permissions se musí dělat opakovaně
    - Problém s mazáním otevřených souborů
    - Výhoda při power failure serveru, jakmile nabootuje, můžou klienti dál pokračovat a nemusí podnikat žádné kroky (můžou ale zmrznout, protože RPC)
- Od verze 3 přidává zámky (NLM protokol)
    - Zámek má lease & grace periodu (grace > lease), pokud by došlo k power failure serveru, ten si po bootu nebude zámky pamatovat, nicméně kvůli omezené lease periodě si musí klienti pravidelně žádat o renewal. Server to vyřeší tak, že po nabootování v tzv. grace periodě uzná žádost o renewal každému, kdo si řekne první (počítá se tedy s tím, že klienti toho nezneužijí).
- Od verze 4 nové fíčury, přidána stavovost (inspirované AFSkem, takže poskytuje například šifrování, autentizaci,...)

AFS (andrew file system)

- Homogenní filesystém fyzicky se rozkládající klidně i na větším množství serverů
- Bezpečný (používá kerberos pro přihlášení)
- Lépe škálovatelný - klienti si kešují soubory a protokol je umí upozornit, že byly na serveru změněny pomocí callbacku; při libovolné chybě je potřeba callbacky reestablishovat
- Používá slabou konzistenci (viz konzistenční modely) - read/write operace se provádí nad lokální kopií souboru, po zavření se změněné části pošlou zpět na server
- Podpora logických jednotek (volumes)
    - Dává možnost transparentní migrace dat
    - Volume může být replikována na jiná místa jako read-only

Coda

- Možnost pracovat v disconnected módu (nejdřív natahat soubory do keše, pak se klidně odpojit a pracovat se soubory - změny se logují a po znovupřihlášení se pošlou a reintegrují)
- U AFS jsou všechny kopie volume read-only (zapisuje se tedy jen na 1 server), Coda umí více writeable kopií. To může při reintegraci způsobit i server/server konflikty.
    - Coda je na to ale dělaná - má automatický a manuální nástroje pro recovery
- Shrnul bych to tak, že je to systém hodně zaměřenej na mobilitu

Distribuovaná správa prostorů jmen, identifikace objektů a přístup k nim, služby (LDAP, JNDI, CORBA Namig, CORBA Trading). - Wiki?

- Řekl bych, že to bude něco jako distribuovaná adresářová služba (podle wiki LDAP i JNDI jsou adresářové služby)
    - Základní operace: set a lookup
- Adresář = množina položek <jméno, hodnota>, kde hodnota může být

    - Nějaká data
    - Reference na nějaký objekt
    - Odkaz na jiný adresář (lokální / vzdálený)
- Lookup("a/b/c") = Lookup("a") -> lookup("b") -> lookup("c")
- Lokační a replikační transparentnost
- Jména = identifikace objektu, typicky binární řetězec pevné délky i struktůry
- Access Control Matrix - user × resource → effective right
    - Kapability - uživatel drží oprávnění k prostředkům
        - podle Zavorala také jednoznačně identifikují persistentní objekt (takže vlastně vzdálená reference, včetně oprávnění)
        - musí mu být znemožněno vlastní generování a změny kapabilit (šifrování, podpis)
        - výhody: snadno se testuje, flexibilní - každý prostředek si může nadefinovat sadu práv
        - nevýhody: složitá revokace, nutná kontrola propagace
    - ACL - prostředek má seznam oprávněných uživatelů
- Navíc to někdy umí zajišťovat autentizaci
- Obecně adresářový služby nezajišťují integritu (oproti databázím), jsou vhodnější spíše pro data, co se hodně čtou, ale málo mění
- Dalo by se zmínit taky DNS

LDAP

- Protokol pro přístup k datům na adresářovém serveru
- Klient-server, součástí je autentizace klienta
- Unikátní identifier = DN (distinguished name) = cesta, př. "a/b/c.txt"
- Relativní identifier = RDN (relative distinguished name), př. "c.txt"
- <TODO>

JNDI

- Furt to samý, prostě je to rozhraní, do kterýho pod stringovym jménem ukládám objekty, můžu nad tim dělat nějaký query a vytvářet handlery aktivovaný při změně obsahu
    - Jména jsou hierarchická, např: com.mydomain.MyBean
- Je jen univerzální nadstavbou nad nějakou adresářovou službou (LDAP, DNS, Corba naming, file system)
    - Takže to lze použít třeba k propojení Java aplikace s adresářovou službou (nějakou databází, LDAPem, nebo CORBím nameserverem)
- Používá to třeba Java RMI
- <TODO>

CORBA Naming

- Abychom nemuseli ručně předávat IOR, vytvoříme naming service, který bude obecně známý
- Namespace obsahuje dvojice <name, reference>, kde reference je nějaký vzdálený objekt - může to být klidně další nameserver

- Lze sestavit hierarchickou strukturu
- Operace bind(name, ref) a resolve(name) : ref

CORBA Trading

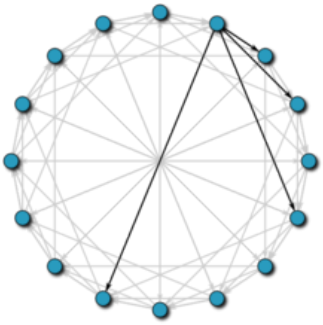
- Objekty se u Trading serveru registrují popisem své funkcionality (př. mail server, tiskárna)
- Klient poté místo podle jména hledá podle popisu - typu a “constraintů” (tj. chci tiskárnu, co tiskne alespoň 50 stránek za minutu, která je v prvním patře)
- Trading server odpoví na dotaz nějakou nabídkou (což je reference na objekt, který splňuje jeho kritéria)

Distribuované hašovací tabulky (Chord, Pastry). - Wiki?

- Decentralizovaný distribuovaný systém, který poskytuje vyhledávání podle klíče, podobně jako hash tabulka
- Páry (klíč, hodnota) jsou rozmístěny v systému a každý node umí efektivně zjistit hodnotu pro daný klíč
- Možné využití: P2P file-sharing service
- Sítě typicky umí efektivně vyhledávat podle klíče, hledání podle keywordů je ale typicky složité
- Vlastnosti DHT:
    - Autonomie a decentralizace - uzly nejsou centrálně koordinovány
    - Fault tolerance - systém by měl zvládat připojování/odpojování uzlů
    - Scalability - systém by měl efektivně fungovat i s velkým množstvím uzlů
- Pojmy
    - Keyspace = prostor klíčů
    - Keyspace partitioning = schéma rozdělující keyspace mezi uzly
    - Overlay network = síť, nad kterou DHT běží umožňující spojení dvojic uzlů
- Přiřazení klíčů nodům - na klíčích zavedeme metriku “vzdálenost”, každému uzlu přiřadíme nějakou hodnotu, záznam s hashem H pak náleží tomu uzlu, jehož hodnotě je H nejblíž - říká se tomu konzistentní hashování
    - Výhoda při přidání/odebrání uzlu - data migrují jen mezi přímými sousedy, ne skrz celou síť
    - Co když uzel prostě zmizí - ztratíme data???
- Vzájemná komunikace - uzly typicky nemají reference “všichni na všechny”, přesto ale musí být schopny pro každý hash H najít majitele - to se zajistí tak, že každý uzel buď záznam sám má, nebo zná uzel, který je k tomu hledanému blíž

Chord

- Uzly jsou v logickém kruhu, každý má nějaké ID (doména IDček je shodná s doménou hashovací funkce, IDčka se vytváří hashováním IP adres)
    - Uspořádání na IDčkách udává ten kruh, zároveň říká, kam se má připojit případný nový člen
    - Uzel má referenci na předchůdce a následovníka v kruhu a k tomu tzv. “finger table” na nody vzdálené o mocninu dvojky (viz obrázek)
        - díky tomu hledání uzlu pro hash H trvá O(log(N)) - podle mě je zřejmé proč, ale pro zájemce na wiki je k tomu poměrně dlouhý důkaz..



Pastry

- Podobný jako Chord, s několika improvementy
  - Redundance dat (tak, že odejde-li jeden uzel, nevadí to)
  - Podpora routování na základě z vnějšku zadané metriky
- Logický kruh, 128-bit ID, náhodně a uniformně zvolená
- Každý uzel udržuje a vyměňuje s ostatními následující informace
  - leaf node list - seznam L uzlů, které jsou nejblíž podle ID (L/2 v každém směru)
  - neighbourhood list - seznam M uzlů, které jsou nejblíž podle routovací metriky
  - routing table (viz dále)
- Routování
  - To se nedá popsat prostě..:-D

Replikace a mobilita v distribuovaném prostředí (konzistence replik, přenos stavu). - PDS?

- Replikace = udržujeme více kopií na více místech
- Smyslem je spolehlivost - nepřijdeme o data, není SPOF
- Dostupnost - výpadek serveru nezruší dostupnost dat
- Výkon - lokální data jsou rychlejší
- Replikace je různého typu
    - Explicitní (vyvolaná userem)
    - Odložená - primární replika na zápis, aktualizace postupně do sekundárních
    - Skupinová - simultání zápisy
- Při aktualizaci kopie buď zápis jen do primární repliky, nebo hlasování
    - Většinové - potřebuju větší polovinu (Nr > N/2, Nw > N/2)
    - Vážené - potřebuju různé kvórum pro zápis a čtení (Nr + Nw > N)
        - Možno optimalizovat pro čtení/zápis
    - S duchy - pasivní uzly se účastní hlasování o zápisu (asi jen kvůli optimalizaci?)
    - Dynamická kvóra
- Klientocentrické konzistenční modely - pohled na sdílená data z jednoho procesu (klienta)
    - V porovnání s DSM modely, které to berou z různých procesů
    - A replikovanými DB, kde se prakticky nepíše, ale brutálně čte
    - Stavěný na dodržení konzistence při přesunu klienta na jiný uzel (mobilní zařízení)
- Typy klientocentrické konzistence
    - Eventuální - Po dokončení zápisů se jednou (v konečném čase) všechny repliky aktualizují
        - Problém je, že klient může po přepojení k jiné replice nevidět změny které už viděl
    - Monotonní čtení
        - Po přečtení hodnoty už vždy uvidíme stejnou nebo novější hodnoty
    - Monotonní zápis
        - Zápis je proveden před jakýmkoliv dalším zápisem
        - CVS commit
    - Čtení vlastních zápisů
        - Zápis proměnné je proveden před jakýmkoliv následným čtením této proměnné
        - Aktualizace -> neprohlížím cache
    - Zápisy následují čtení
        - Zápis po předchozím čtení je proveden na stejné nebo novější hodnotě
        - Zápis do newsgroups se provede tam, kde je i to na co reaguju
        - Představa: inkrementace
- Implementace
    - Naivní

        - Zápisy mají identifikátory WID - určeny replikou kde klient zapisoval
            - WID obsahuje identifikaci repliky
        - Klient má u sebe read-set a write-set (to co už četl, co zapsal)
            - Mohou neomezeně růst! (asi proto že neznáme vztahy mezi zápisy)
    - Efektivnější
        - Řeší neomezený růst
        - Buď pomocí sessions
        - Nebo pomocí reprezentace množin pomocí vektorových hodin
            - Zápisu se přiřazují i timestampy
            - Replika Si si udržuje RCV(i)[j] - TS poslední operace zápisu od Sj
            - Porovnáváme VT klienta pro read-set a write-set - replika musí být aktuálnější
            - Aktualizujeme po operaci klasicky maximem
    - Epidemické
        - Eventuální konzistence
        - Neřeší kolize
        - VELMI rozsáhlé sytémy
        - Uzly si náhodně vyberou někoho k výměně (Antientropie), typy push (vecpu mu svoje data), pull (vezmu si jeho data), obojí
        - Kdy přestat infikovat?
            - Gossiping - při nákaze už nakaženého se s nějakou pravděpodobností uklidníme (a už nebudeme pushovat)
        - Super rozšiřitelné, rychlost šíření se dá optimalizovat hierarchickým uspořádáním
        - Problém s mazáním dat - můžou se nám dál šířit od jinud
            - Death certificate - řeší zablokování šíření, ale musíme řešit JEJICH mazaní
            - Certifikát má omezenou platnost - funguje, ale stojí na konečné době rozšiřování infekce
        - Dá se použít i pro různé aplikační úlohy
            - Spočítat uzly
            - Dá se převést na průměr - epidemicky měníme data jejich průměrováním - po čase se dostaneme přibližně k průměru
            - Pro spočítání iniciujeme hodnoty na 0 kromě vyzývajícího uzlu, jde bude 1 -> jde k 1/N

Architektury distribuovaných aplikací, SOA, ESB, P2P. - MWy+???

- <TODO nějaký obecný kecy?>
- Souvislost s middlewarem (různý mw vhodný pro různé architektury)
- Middleware = “to, co je mezi OS a tím, co se chce člověku programovat”; zejména poskytnutí prostředků pro komunikaci mezi distrib. komponentama
    - Zmínit vrstvy, dole hw, přenosový protokoly, výše pak spolehlivost, formátování, nahoře použitelný rozhraní (zprávy, RPC, …)

- Terminálová architektura (inteligence v serveru, hloupý klient)
- Chytří klienti, server zprostředkovávající jen komunikaci
- Tiered architecture
    - Tier = oddělená část systému, která něco implementuje (logiku, prezentaci)
    - Příklad two-tier: klient-server
    - Příklad three-tier: klient-server + rozdělit klienta na logiku a prezentaci
- Klient-server
    - Uzly mají jiné role
    - Typicky poskytování nějakých služeb
- Distributed object architecture
- Service-oriented architecture
    - <a href=”Vývoj SW systémů/Webové služby a servisně-orientovaná řešení, webově orientovaná řešení.” />
- ESB - Enterprise service bus
    - Cílem je unifikovaná komunikace různých service-based (hlavně SOA) technologií a platforem
    - Důraz na modularitu - můžeme služby přidávat, odebírat atp., typicky i za běhu
    - Protože se jedná o defacto standardní způsob implementace SOA, sdílí s ní mnoho cílů, zejména transparentnost a orientaci na služby
    - Je to architektonický model
    - Snaha mít standard popisující implementaci softwaru na bázi rozličných nezávislých komunikujících komponent (služeb)
    - Prý podobný server-klient modelu, představuju si, že ESB plní roli “distribuovaného” serveru
    - Mělo by se starat o
        - Zajišťování a monitorování komunikace mezi službama (včetně konverze protokolů)
        - Řešit konflikty
        - Řídit deployment a verzování služeb
        - Řídit použití redundantních služeb (předpokládám že to znamená že službu nahradí při výpadku)
        - Zajišťuje určité standardní služby - handlování eventů, transformaci dat a protokolů (předpokládám věci jako endiany, různé reprezentace stringů,

kódování, převod ze SOAPu na JSON...), handlování exceptions, zajištění kvality komunikace
    - Typicky postavený nad nějakým messaging middlewarem
    - Vlastně zajišťuje environment a prostředek pro komunikaci služeb
    - Web se dá vnímat jako ESB :)
    - Nesmí existovat jinej způsob komunikace mezi službama než skrz ESB - to dává ESB šanci provádět se zprávama další čachry (monitoring, statistiky, logy...)
    - Často taky umí publisher-subscribe
    - Typicky zajišťuje i buffering
- P2P
    - Uzly jsou rovnocenné (equipotentní :-D), bez centrální arbitrace
    - Uzly většinou tvoří nějakou logickou strukturu (nezávislá na topologii)
        - unstructured - “náhodná” struktura, prostě se nějak spojí (jednoduchý, robustní, ale složitý hledání - typicky nějaký flooding)
        - structured - zaručuje efektivní hledání prostředků v síti (viz DHT)
- EJB - Enterprise Java Beans
    - Komponentová architektura (tedy klasické vlastnosti - modularita, atp.)
    - Zamýšlená na server
    - Cílem je realizace a zapouzdření byznys logiky
    - Bean = komponenta poskytující služby
    - Beany se deployují do kontejnerů :D
    - Klienti závisející na beaně ji neinstancuje, ale poprosí EJB kontejner o referenci
        - Vzniklá reference je ve skutečnosti proxy objekt, který ve spolupráci s EJB kontejnerem řídí komunikaci s druhou komponentou
        - Dva typy rozhraní s beany - local (na stejném stroji) a remote (skrz CORBU, nebo něco podobného)
        - Klient má tedy na bean tzv. “view” - view podle typu rozhraní local, remote, no-interface
        - Beany nemusí nutně implementovat nějaký interface, i když je to asi většinou rozumné
            - V takovém případě je proxy objekt subclass beany
    - Kontejner zajišťuje transparentně podobné služby jako ESB, akorát v heterogenním prostředí a ne totálně loose-coupled
    - Poskytuje transakční zpracování - proxy objekt při vykonání metody čeká a na konec vykoná commit nebo rollback
    - Umí se integrovat s Java Persistence API (JPA) a JNDI
    - V nové verzi (3.0+) přebírá inspiraci od Springu a Hibernate (ORM)
        - Mnohem jednodušší API
        - Umí používat dependency injection pro integraci a konfiguraci
    - Typy beans
        - Session
            - Stateful - stavové byznys objekty, stav beanu odpovídá klientovi, přístup vždy sériový, např. uchovávání stavu objednávky

        - Stateless - bezstavové, stále jen jeden klient v každý okamžik, při paralelním přístupu přesměruje na jinou instanci
        - Singleton - globální stav, paralelní přístup ovládá buď kontejner, nebo bean sám
    - Message-driven - spouštěny zprávami místo volání metod (JMS)

MISC

Distribuovaný systém

- Systém vzniklý propojením uzlů
- Transparentní pro uživatele
- Typicky chápáno tak, že uzly jsou kompy spojené sítí, ale obecně jde na rúzných úrovních
- Evoluce: Distribuovaný OS (fail), distribuovaný framework a aplikace (CORBA), Cloud computing, XaaS (whatever as a Service)
- Mnoho využití: ditribuovaná data, mobilní systémy, ambient intelligence, agenti, content management (torrent)...
- Výhody
    - Ekonomika (levnější přidat deset nových PC než mainframe)
    - Rozšiřitelnost (^)
    - Spolehlivost (není single point of failure)
    - Výkon (překonává tech. limity)
    - Distribuovanost
- Cíle návrhu
    - Transparentnost (přístupová, lokační, migrační…)
    - Přizpůsobivost (decentralizace, autonomie)
    - Spolehlivost, Výkonnost, Rozšiřitelnost (Scalability)
- Pro rozšiřitelnost je zásadní eliminovat centralizované koncepty
- Nárůst výkonnosti není lineární, je nižší

Zajištění spolehlivosti

- Duplicita zpráv - číslování / zahození
- Předbíhání zpráv - číslování
- Ztráta zprávy: potvrzení / timeout
- Přenos dlouhých zpráv: burst - potvrzuje se až poslední (větší režie při chybě)
- Idempotentní služba = opakované provedení neva