1 Operační systém, proces, vlákno, algoritmy plánování procesů a vláken. Komunikace mezi procesy a vlákny, časově závislé chyby, kritické sekce, synchronizační nástroje, problém uváznutí, možnosti řešení. Správa paměti. Virtuální paměť - stránkování, algoritmy pro náhradu stránek, segmentace. Souborové systémy, organizace dat na vnějších pamětích, principy, řešení, ochrany. (A4B33OSS)

1.1 Operační systém

- zakrývá detaily hardware, aby mohly programy PC snáze ovládat (poskytuje API)
- přiděluje prostředky programům (paměť, procesor...)
- Unixové systémy splňují standard POSIX, který sjednocuje API, systémová volání jsou stejná napříč systémy (open, read, write, fork, mkdir, link...)

1.2 Proces

- je to program označený číslem procesu PID, info o něm je uložené v registru Process Control Block
- tvoří ho obsahy registrů procesoru a uživatelských registrů, otevřené soubory, použitá paměť...
- proces je chápán jako obal vláken, vlákna jsou součástí procesu

- procesy mohou vytvářet nové procesy - potomky - pomocí volání fork/clone. V POSIX jsou všechny procesy potomkem procesu init s PID 0
- proces se ukončí pomocí exit, nebo chybou, nebo na žádost rodiče, nebo při zániku rodiče
- proces se může odpojit od rodiče, pak se mu říká démon
- program je seznam instrukcí

1.3 Vlákno

- existuje v rámci procesu, často má proces jen jedno vlákno
- zdroje jsou sdílené mezi procesem a všemi jeho vlákny
- ukončení procesu zničí všechna jeho vlákna
- správu vláken řeší v POSIX knihovna pthreads

1.4 Algoritmy plánování procesů a vláken

- **Nepreemptivní plánování (N):** Jakmile se procesu přidělí procesor, nelze mu ho vzít. Používá se v uzavřených systémech, kde jsou všechny procesy "slušné" a pustí ostatní včas k procesoru.
- **Preemptivní plánování (P):** Procesu schopnému běhu se procesor odebere a spustí se jiný proces

1.4.1 Různé algoritmy

Plánování se znázorňuje Ganttovým diagramem. Většinou se "doba procesu" musí odhadovat. Je potřeba to řešit chytře, protože časté přepínání je náročné. Když se vybírá proces podle priority, tak se zase může stát, že nějaký proces se vůbec nedostane na řadu (stárnutí).

- First come first serve (N): Procesy se obsluhují v pořadí, v jakém přišly. Nevýhodou je, že všechny krátké procesy budou čekat, než skončí jeden dlouhý.
- Shortest process next (N): Bere se vždy nejkratší proces.
- Shortest remaining time (P): Pokud se objeví proces, který je kratší, než zbývající doba aktuálního procesu, tak se aktuální proces přeruší a spustí se nový proces.
- Round robin (P): Všechny procesy se střídají rovným dílem
- feedback: Procesy, které trvaly nezvykle dlouho, se při příštím běhu dá horší priorita

- RMS: Používá se v systémech, kde se procesy spouští pravidelně s různými periodami. Přednost se dává procesu s kratší periodou

1.5 Kritická sekce

- úsek programu, kde se přistupuju ke sdílenému prostředku
- pokud je v kritické sekci vždy jen jeden proces, vše je ok. Když jich je víc, můžou si navzájem přepisovat sdílený prostředek (třeba proměnnou), protože procesy mohou být v procesoru vyměněny v nevhodnou chvíli. Těsně po tom, co jeden z nich si načte sdílený prostředek k sobě, jsou procesy vyměněny, a ten druhý hned potom změní sdílený prostředek, ale první dál pracuje se starou hodnotou.

1.5.1 Řešení kritické sekce

- Pomocná proměnná: Procesy označují v proměnné, jestli už do kritické sekce vstoupily. Pokud vidí, že v kritické sekci už nějaký proces je, tak čekají (sleep). To ale jen posune kritickou sekci na tuto proměnnou, takže to nefunguje.
- Tři pravidla
  - V kritické sekci smí být v každém okamžiku jen jeden program
  - Výběr procesu k běhu se nesmí nekonečně odkládat
  - Čekání na vstup do kritické sekce musí být omezená
- Petersonovo řešení na aplikační úrovni: Udělají se dvě pomocné proměnné. Proces postupně označí obě, a pak vstoupí do kritické sekce. Pokud je jedna z proměnných aktivovaná, tak čeká a neoznačuje tu druhou.
- HW řešení: Některé procesory mají instrukci TestAndSet, kterou se čte a mění pomocná proměnná. Během této instrukce nejde proces přerušit.
- Semafor: Konstrukce některých programovacích jazyků, která zajišťuje, aby ke zdroji vždy přistoupil jen jeden proces. Kromě semaforu se taky používá Monitor nebo SpinLock.

1.6 Deadlock

- Máme dva procesy, každý z nich si zabere jeden ze dvou zdrojů a pak čekají, než se uvolní ten druhý zdroj, tím se vzájemně zablokují. Jako příklad se dává 5 večeřících filozofů, kteří sedí u kruhového stolu a mají 5 talířů a 5 vidliček, ale k jídlu potřebují dvě vidličky. Pokud se budou všichni filozofové chovat stejně, nikdy se nenají.
- Deadlock se vizualizuje pomocí grafu Resource Allocation Graph (RAG). Procesy a zdroje jsou vrcholy. Pokud proces P čeká na zdroj Z, vede hrana z P do Z. Pokud

je zdroj Z zabrán procesem P, pak vede hrana ze Z do P. Pokud je v grafu cyklus, tak je systém v Deadlocku

- Bankéřův algoritmus: Dovolí spustit jen ty procesy, u kterých ví, že jim za současného stavu dokáže půjčit zdroje. Nevýhodou je, že předem chce vědět seznam zdrojů, které bude proces potřebovat, což je ne vždy znát.
- Deadlock nelze efektivně řešit, lze mu předcházet (Bankéř), nebo ho detekovat a řešit zpětně.

1.7 Správa paměti

- Adresy z Fyzického Adresního Prostoru (FAP) popisují konkrétní místa v paměti (Fyzické adresy, FA)
- Adresy z Logiského Adresního Prostoru (LAP) popisují adresy, se kterými pracují aplikace (Logické adres, LA)
- Adresy z LAP se překládají do FAP
- Adresa, která má délku 32 b popisuje prostor o velikosti 2^32 = cca 4GB

1.7.1 Sementace

- Paměť se rozdělí na víc segmentů, které mohou mít různou velikost
- Paměť se adresuje jako číslo segmentu a pak pozice v segmentu, to se pak pomocí tabulky ST převede na FA
- Přístup do špatného segmentu vyvolá chybu segmentace (segfault)
- Segmenty lze využít, když je paměť větší než dovoluje adresa popsat (adresa je delší, než povoluje vstup do procesoru), nebo když chceme oddělit paměti procesů od sebe

1.7.2 Virtualizace

- Virtualizace se hodí, když je paměť menší než to, co by šlo popsat adresou. Proto se část paměti uloží jinde

1.7.3 Stránkování

- FAP je rozdělený na rámce stejné velikosti, a těm odpovídají stejně velké stránky v LAP
- Překlad mezi rámci a stránkami je prováděn pomocí page table (PT)

- Adresa je určena číslem stránky p a offsetem ve stránce. Překládá se jen číslo stránky na rámec, offset je stejný.
- Aby se překlad adres zrychlil, ukládají se stránky do cache TLB
- Když se cache naplní, tak je potřeba najít stránku, která se přesune na disk, aby bylo v tabulce místo pro novou
  - First in First Out (FIFO): smažeme nejstarší stránku
  - Last Recently Used (LRU): smažeme stránku, která byla nejdéle nepoužita
- Cache může být víceúrovňová, taky se dá použít hashování

1 Počítačové sítě, hierarchický model ISO/OSI. LAN, jejich topologie, technologie a adresování, propojování LAN do internetových struktur, adresování, směrování. IP protokoly (UDP, TCP, ICMP) a jejich aplikace v konkrétních protokolech (HTTP, SMTP, DNS). Výpočty v síťovém prostředí, klient/server, sockety. (A4B33OSS)

1.1 Histrorie internetu

- 1969 Arpanet
- 1982 TCP/IP
- 1991 WWW

1.2 Topologie sítí

- Síť, hvězda, sběrnice, kruh, strom

1.3 ISO/OSI model (International Organization for Standardization/Open Systems Interconnection model)

ISO OSI model je rozdělený na vrstvy. Teorie je taková, že v každé vrstvě se řeší něco jiného a není potřeba znát fungování jiných vrstev. Třeba emailový klient nepozná, jestli mu data přišla přes wifi nebo kabelem. Data vytvořená v aplikační vrstvě (packet) se obalí hlavičkou nižší vrstvy, postupně se uzavírá do dalších a dalších hlaviček, poslední je hlavička ethernetového rámce. Zařízení se vždy dívají jen na hlavičku z vlastní vrstvy

Vrstva	Vrstva anglicky	Protokoly	Název v TCP/IP	Zařízení	Pomůcka
Aplikační	Application	FTP, HTTP, SSH	Aplikační		All
Prezentační	Presentation	CSS, HTML, GIF	x		People
Relační	Session	SQL, SSL, PAP	x		Sleeping
Transportní	Transport	TCP, UDP	Trasportní		Through
Síťová	Network	IP	Síťová	Router	Networking
Spojová	Link	MAC	Spojová	Switch	Lectures
Fyzická	Physical	Ethernet	Spojová	Hub	Fail

Table 1.1: Vrstvy ISO/OSI modelu

- třeba router posílá packety dál podle IP adresy a je mu jedno, jestli to je TCP nebo UDP.

V praxi to ale tak vždy není. Třeba FTP posílá IP adresy uvnitř svého packetu, takže když prochází FTP packet routerem na hranici sítě (bránou), ve kterém se adresy překládají (NAT níže), je potřeba packet rozbalit a upravit IP adresy v něm.

TCP/IP je obecnější model, který má vrstvy jen čtyři.

1.4 Protokoly

1.4.1 Ethernet

- Na fyzické úrovni je nutné řešit, aby nevysílalo víc zařízení najednou, jinak se signál zaruší a nelze ho poslouchat (tomu se říká kolize)
- Kolizím lze předcházet, třeba tím, že každý připojený stroj má přidělený čas k vysílání (tak to řešil protokol token ring)
- Ethernet kolizím nepředchází, ethernet je detekuje. Každé zařízení během vysílání poslouchá, jestli nevysílá někdo jiný. Pokud ano, obě zařízení přestanou hned vysílat a budou opakovat vysílání po náhodné době.
- V ethernetu se data ověřují pomocí Cyclic Redundancy Check (CRC)

1.4.2 ARP (Address Resolution Protocol)

- Převádí IP adresu na MAC adresu
- Odesílatel pošle požadavek “Who has 192.168.37.178”, aby zjistil, kdo má danou IP. Dostane odpověď a MAC adresu si uloží do tabulky pro příště.

1.4.3 IP

- Slouží k adresaci strojů v síti

- IPv4: Čtyři čísla po 8 bitech, celkem 32 bitů, má 4 miliardy adres, ale dnes už nestačí
- Adresy jsou rozděleny do menších sítí, ty se udávají maskou
- Maska v IP verzi 4 označuje, kolik bitů na začátku adresy má celá síť stejně, ostatní bity identifikují zařízení uvnitř sítě (např. 192.168.0.0/16)
- IPv6: Osm čísel po 16 bitech, celkem 128 bitů, tj 3*10^38 adres, to už by stačit mělo

1.4.4 NAT

IP adresy uvnitř sítě (třeba 192.168.*.*) nejsou v celém internetu unikátní, taky se jim říká neveřejné. Routery v internetu packety z neveřejných adres mažou. Aby mohly počítače z vnitřních sítí komunikovat se serverem v internetu, musí jim být přidělena veřejná IP adresa, na kterou jim pak server pošle odpověď. Překlad neveřejných IP adres na veřejné obvykle zajišťuje hraniční router sítě (brána) protokolem Network Address Translation (NAT) s pomocí tabulky. Takový router (většinou patří poskytovateli internetu) má několik veřejných IP adres, a když přijde z vnitřní sítě požadavek něco poslat do internetu, uloží si do tabulky neveřejnou IP počítače a k tomu přidělí veřejnou IP. Jakmile přijde odpověď, podívá se do tabulky, aby zjistil, kterému počítači má packet poslat. Při pozdější komunikaci může být veřejná adresa přidělená počítači jiná. Pokud je adres málo, lze pracovat i s porty.

1.4.5 UDP

User Datagram Protocol slouží k posílání packetů přes síť. UDP neumí garantovat, že všechny packety budou doručeny, a že budou doručeny ve správném pořadí. Občas se prostě stane, že se nějaký packet úplně ztratí. UDP se používá u služeb, kde je prioritou čas, a není důležité mít kompletní informace - třeba u online streamů

1.4.6 TCP

Transmission Control Protocol také řídí posílání dat, ale garantuje, že dorazí v pořádku. Spojení je navázáno "handshakem", kdy nejprve stroj A vyšle žádost o spojení (SYN), spojení je potvrzeno strojem B (SYN ACK), a stroj A potvrdí, že bylo spojení úspěšně navázáno (ACK). Každý packet má své číslo, a přijemce musí potvrdit, že packet obdržel odesláním zprávy ACK. Pokud přijetí nepotvrdí (nebo se ACK ztratí), tak bude packet odeslán znovu. TCP se používá u služeb, kde chceme zaručeně správné a kompletní informace a nepotřebujeme je okamžitě - třeba email.

1.4.7 ICMP

Internet control message protocol je nástroj, který informuje o stavu sítě. Jeho součástí je třeba ping nebo traceroute. Zprávy "Destination unreachable" nebo "Time limit exceeded" jsou taky z ICMP - odesílá je router, když maže packet, který se pohyboval

internetem moc dlouho. Stáří IP packetu (Time To Live - TTL v jeho hlavičce) se počítá podle počtu navštívených routerů, číslo se postupně snižuje, na nule je packet smazán, protože zřejmě zabloudil.

1.4.8 HTTP

HyperText Transfer Protocol funguje na principu klient server. Klient posílá serveru požadavky, třeba GET, POST, PUT, DELETE, server je vykonává a odpovídá třímístným číselným kódem:

- 1..: Informativní
- 2..: Vše proběhlo ok (200 OK)
- 3..: Přesměrování
- 4..: Chyba na straně klienta (403 forbidden, 404 not found)
- 5..: Chyba na straně serveru (500 internal server error)

1.4.9 SMTP

Simple mail transfer protocol, stará se o emaily.
A: MAIL FROM <...@...>
B: ok
A: RCPT TO <...@...>
B: ok
A: DATA
B: ok, ukonči data pomocí \n . \n
A: ..... \n . \n
B: ok
A: QUIT
B: bye

1.4.10 DNS

Domain name system překládá doménová jména (google.com) na IP adresy. Domény mají různé řády, třeba v cyber.felk.cvut.cz je cz doména prvního řádu a cyber doména čtvrtého řádu. Domény se registrují u autoritativního serveru. Při požadavku na identifikaci domény se nejprve server podívá do své paměti, a pokud doménu nemá, ptá se dalších serverů. Autoritativní servery sdílí své domény ostatním serverům společně s jejich dobou platnosti.

1.5 Sockety

Socket je způsob komunikace mezi procesy, v rámci pc i přes síť. Jeví se jako soubor a zapisuje se do něj jako do ostatních souborů v POSIX (read, write).

1.6 Bezpečnost

Bezpečnost na síti se řeší dvěma způsoby

- Symetrické klíče: Odesílatel má symetrický klíč, tímto klíčem zašifruje zprávu. Příjemce má stejný klíč a pomocí něj zprávu přečte. Je to výpočetně jednodušší než asymetrické klíče, ale je potřeba vyřešit bezpečné předání klíčů. Taky nejde vůbec poznat, kdo zprávu zašifroval. Jakmile klíč unikne, lze pomocí něj přečíst všechny zprávy.

- Asymetrické klíče: Klíč má dvě části, veřejnou a soukromou. Veřejná část je volně přístupná, a je výpočetně nemožné zjistit z ní soukromý klíč. Server, řekněme banka, má u sebe uložený svůj soukromý klíč. Ten dokáže přečíst všechny zprávy, které byly zašifrovány odpovídajícím veřejným klíčem. Veřejný klíč je registrovaný u certifikační autority danou bankou, certifikační autorita potvrzuje, že klíč opravdu patří bance. Klienti posílají svá data bance zašifrovaná pomocí veřejného klíče, a jediná banka je dokáže pomocí svého soukromého klíče číst. Je ale důležité, aby certifikační autorita (klidně i lokální databáze certifikátů) byla důvěryhodná - viz chyba Superfish u Lenova, hodně hezky to vysvětluje computerphile na youtube.

1.6.1 SSL a TLS

Transport Layer Security je novější verze SSL. Používá symetrickou kryptografii, ale k bezpečnému předání klíčů používá asymetrické klíče. TLS se používá v HTTPS (bezpečné verzi HTTP), komunikace mezi klientem (K) a serverem (S) se navazuje takto:

K: posílá úvodní packet se seznamem podporovaných šifer a verzí TLS

S: odpovídá, posílá šifru a verzi TLS, která se bude používat a svůj veřejný klíč

K: ověří veřejný klíč u sertifikační autority. Pokud je platný, tak vygeneruje symetrický klíč (ve skutečnosti vygeneruje Master secret, to je složitější), ten zašifruje pomocí veřejného klíče a pošle serveru

S: přečte zprávu pomocí svého soukromého klíče, tím získal symetrický klíč. Teď mají obě strany symetrický klíč a mohou komunikovat.

1.6.2 RSA

Asymetrická šifra, která se používá třeba u SSH. Máme náhodná hodně velká prvočísla p a q, a číslo N=p*q, které je veřejně známé. Najdeme e, které je nesoudělné s fí(N) = (p-1)(q-1). Najdeme d, aby e*d = 1 mod fí(N).

Veřejný klíč je (N, e), soukromý klíč je (N, d). Zprávu Z zašifrujeme C = Z^e mod N, a dešifrujeme Z = C^d mod N.

1.6.3 Diffie-Hellmann

Dá se použít k vytvoření symetrického klíče mezi dvěma stranami (Alicí a Bobem) bez pomoci certifikační autority. Máme veřejná čísla p, q, a tajná čísla a od Alice a b od

Boba. Alice pošle Bobovi q^a mod p, a Bob pošle Alici q^b mod p. Z toho mohou oba spočítat klíč K = g^(a*b) mod p, ale po odposlechnutí komunikace nelze kód zjistit.

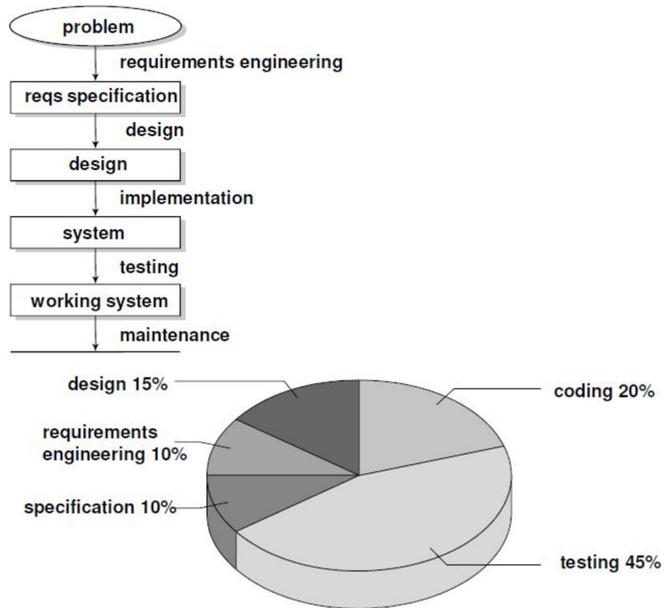
1.6.4 Man in the middle

Man in the middle (MIM) je útok, během kterého útočník přeruší komunikaci mezi oběma stranami a vede ji přes sebe, u toho pak mění obsah komunikace. V příkladu Diffie-Hellmannovy výměny klíčů může MIM vygenerovat vlastní číslo x, a komunikovat s Alicí pomocí klíče g^(a*x) a s Bobem pomocí klíče g^(x*b), uprostřed zprávy rozbalí, přečte a zašifruje pomocí druhého klíče. Ani jeden z komunikujících se nemůže dozvědět, že komunikace byla narušena.

3 Proces vývoje software, jeho struktura a fáze vývoje, klasické a moderní agilní metodiky vývoje, řízení rizika. (A4B33SI)

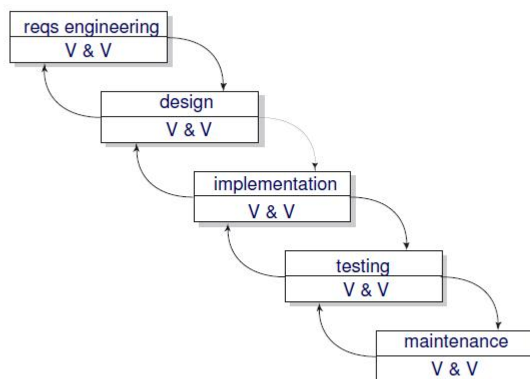
3.1 Proces vývoje software

1. **Requirement engeneering** (Stanovení požadavků) => Dokument 10%
    - jaké funkce, možnost rozšíření,...
    - získání nutné dokumentace, co musí umět (funkční/nefunkční požadavky, akceptační podmínky, náročnost na výkon...
2. **Design** (Návrh) 10%(spec) + 15%(design)
    - klade se důraz na to, co musí umět, ne jak
    - výsledkem je přesná specifikace
3. **Implementation (**Implementace) 20%
    - zaměření na jednotlivé komponenty
    - cílem je funkční SW
4. **Testing** (Testování) 45%
    - dělá systém to, co má?
    - mělo by se testovat v průběhu celého vývoje (až 45% času vývoje!)
    - validace: Děláme správný systém? (dle požadavků)
    - verifikace: Děláme systém správně? (bez chyb)
5. **Maintenance** (Údržba)
    - oprava chyb po předání zadavateli (neměly by být, ale jsou...) 21% + 4%(prevence)
    - úprava SW (25%), přidávání/úprava funkčnosti (25%)

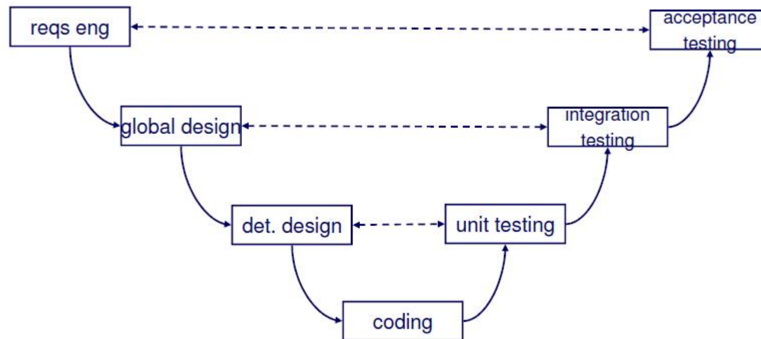


3.2 Metodiky vývoje

1. **Tradiční model**
    - vhodný na velké projekty
    - problémy: špatná (žádná) zpětná vazba, údržba nezahrnuje vývoj
2. **Waterfall model**
    - mezi každým krokem se provádí verifikace a validace a případně se vrací k dodělání
    - stále příliš nepružný (stále pevná dokumentace)



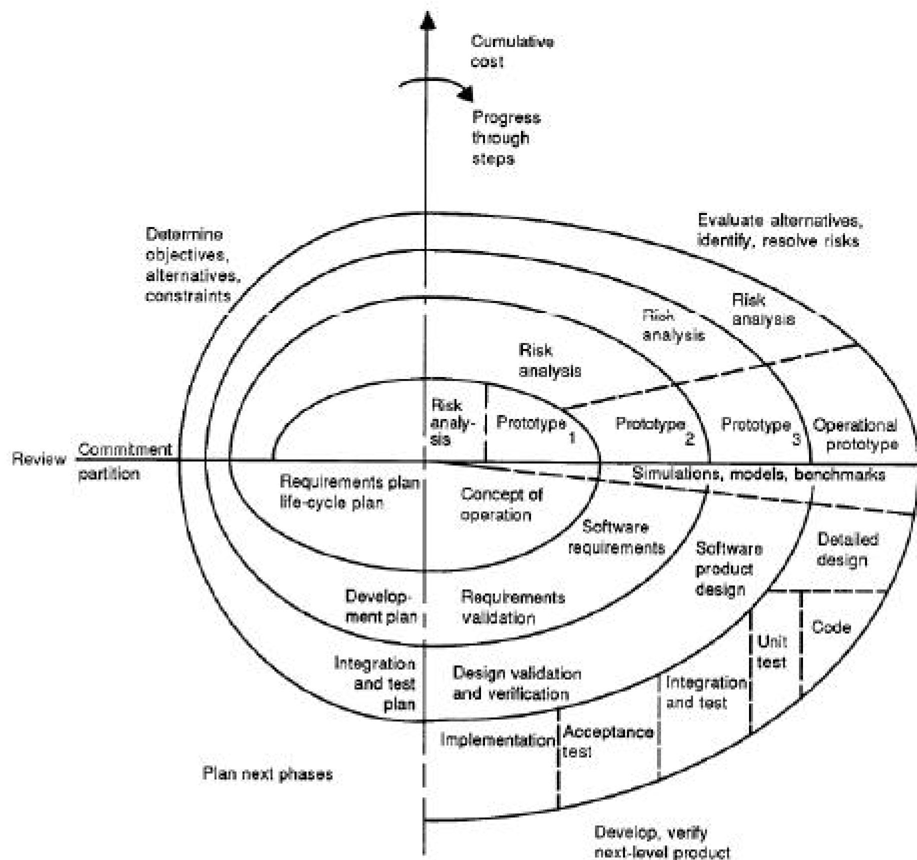
3. **V-model**
    - provádí se verifikace a validace vůči fázím
    - stále nepružný



4. **Prototypy**
    - celkem levné, nemusí umět vše -> je ale nutné ukáza, jak bude fungovat
    - 2 typy:
        - *throwaway* - na jedno použití -> jako *waterfall*, až N-tý bude finální, jinak se zahodí a začne se znovu
        - *evolutionary* - vývojový -> až několikátý bude dodán
    - klady: jednodušší a rychlejší vývoj, rychleji se objeví chyby, jednodušší údržba (někdy)
    - zápory: více funkcí než je potřeba, méně výkonný, horší design, těžší údržba (někdy), nutná větší zkušenost vývojářů

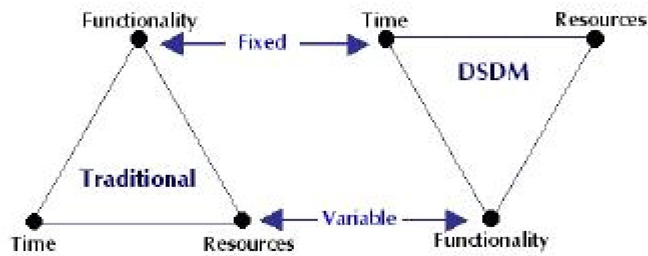
5. **Incremental development**
    - dodáván po kouskách, v každém kousku waterfall model
    - uživatel více zatažen do vývoje (pro každý kousek)
    - nebude mít více funkcí než je nutné

6. **Spiral model**



7. **RAD** (Rapid Application Development)
   - evoluční vývoj s časovými rámci (do kdy co musí být hotovo)
   - nejprve časový rámec -> snaha v něm udělat co nejvíce dle priorit
   - SWAT teams - skilled workers with advanced tools
8. **DSDM** (Dynamic System Development Method) - nejvíce v UK
   - pevně daný čas a prostředky -> odvíjí se funkčnost (u tradičních naopak)
   - nutná spolupráce uživatele
   - testování během vývoje
   - inkrementální vývoj, možnost vrátit změny
   - fáze:
     a) report a osnova o proveditelnosti, rychlý prototyp
     b) bussiness study - analýza, dodání architektury
     c) funkční model - časové rámce, inkrementační fáze

d) design a build iterace
e) implementace



9. **RUP** (Rational Unified Process)
   - doplňuje UML, používán na objektové systémy
   - fáze:
     a) Inception (začátek) - určení cílů, kritické use-cases, časový plán, odhad ceny
     b) Elaborate - založení architektury, včechny use-cases
     c) Construction - manufactory process (= prokládání dohromady)
     d) Transition - uvolnění pro uživatele, často několik releases
10. **MDA** (Model Driven Architecture)
    - nezávislý na architektuře -> výsledkem je PSM = platform specific model
    - 2 typy:
      - MODEL ———> CODE↻ (*údržba*)
      - (*údržba*) ↻MODEL ———> CODE

3.3 Řízení a rizika

- řízení:
  - *času* - počet člověko-hodin a plánování; těžko měřitelné; více lidí != méně času
  - *informací* - především dokumentace (Agilní méně, ale má lepší lidi); aktuální stav
  - organizace - týmy, lidé,...; organizace práce
  - kvality - žádná funkčnost navíc, přesně to, co má mít; nutná komunikace se stakeholdery
  - peněz - různé, hodně o osobách
- řízení pomocí CCB (Configuration Control Board)

  - drží aktuální stav

- méně lepších lidí, vyvážený tým

4 Technické aspekty softwarového projektu, projektová dokumentace: uživatelská specifikace, technická specifikace a návrh, testování, validace a integrace. Systémy pro správu konfigurace a podporu vývoje. (A4B33SI)

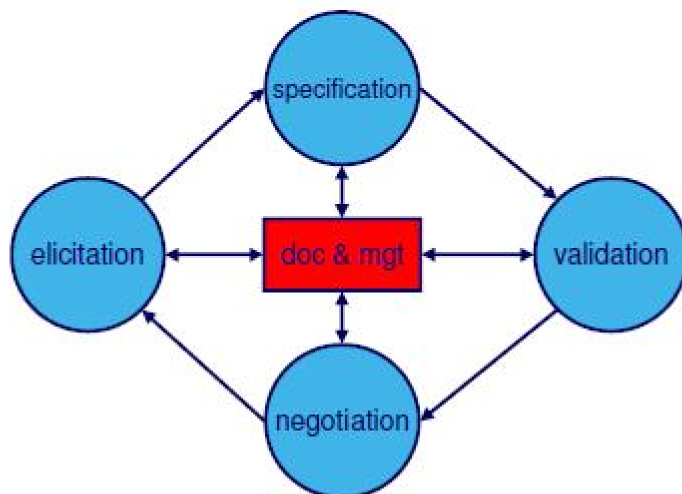
4.1 Technické aspekty softwarového projektu

- řízení projektu - viz otázka 03
- dodržování ISO - spolehlivost, efektivita, užitečnost, udržovatelnost, flexibilita...

4.2 Projektová dokumentace

4.2.1 Uživatelská specifikace

- zjištění aktuálního stavu, jak s ním jsou uživatelé spokojeni, co jím vadí, co by pořebovali
- výsledkem je *Specifikace požadavků*
- nutnost:
  - vše správně pochopit (*elicitation*)
  - správně popsat (*specification*)
  - sjednat podstatu problému (*validation*)
  - sjednat hranice problému (*negotiation*)
  - => opakuji, dokud není vše jasné!



4.2.2 Technická specifikace a návrh

- vychází z uživatelské specifikace a přesně stanovuje, co se má udělat
- ! vyvarova se obecnému jazyku => kniha ≠svazek knihy; přítomna × nepřítomna (ztracena, zapůjčena, zničena,...)
- vypracování všech use-cases - hiearchické uspořádání => rozložení do menších celků
  - ideálně přiřazení konkrétních stakeholderů
- priority: MoSCoW
  - Must have (nejvyšší priorita)
  - Should have (chtěné)
  - Could have (když zbyde čas/peníze)
  - Won't have (dnes ne)
- stanovení funkčních a nefunkčních (rychlost, spolehlivost, jednoduchost,...) požadavků

CMM (Capability Maturity Model)

- na začátku opakovatelný level (analýza, requirements,...) => definitivní level (techické řešení, V+V,...)

SPI (Software Process Increment)

- stanovení hypotézy => sesbírání dat informací => interpretace dat ↻znovu, dokud nemáme vše

Odhad ceny

1. kvantitavní modely
2. kvalitativní modely
3. **COCOMO** (Constructive Cost Model) - dobře dokumentované na výpočty, podle typu a velikosti projektu
4. **FPA** (Function Point Analysis) - stanovení ceny dle struktury, vstupů, funkcí,...

- těžké správně odhadnout

4.2.3 Testování

- nutné testovat, dle metodiky je nejvhodnější testovat už v průběhu vývoje (levnější)
- poměr 2:3 (tester:vývojář)
- testování je měření kvality SW - ISO 9000
    - funkčnost (správnost, spolehlivost), inženýrské řešení (efektivita, dokumentace), adaptibilita (opětovné použitá, údržba)
- kritéria pokrytí testů:
    1. řádky - každý řádek se vykoná alespoň jednou => nedostatečné
    2. větve - každá musí být alespoň 1 pravdivá a 1 nepravdivá
    3. podmínky - zkontroluje všechny možnosti nastalé podmínky a vyhodnotí je (vyžaduje armáda a letectví)
    4. úplné pokrytí cest - v praxi neproveditelné
- definování pomocí grafů:
    - *uzly* = objekty, o které se zajímáme
    - *hrany* = vztah objektů a relací mezi uzly
    - postup:
        * definuj graf
        * definuj relace
        * navrhni testy pro pokrytí uzlů a hran
        * otestuj a porovnej s očekávanými výsledky
        * nahvrhni a otestuj testy smyček
- po testování je nutné mít přesné a pdrobné specifikace! (jinak se nedá testovat, není co)

- automatizace testování:
  - klady:
    * častější testování
    * ověření na nové verzi programu
    * opakovatelnost testů
  - zápory:
    * nereálná očekávání
    * slabší testovací praxe
    * údržba automatizovaných testů

4.3 Systémy pro správu konfigurace a podporu vývoje

WBS (Work Breakdown Structure) - rozložení projektu na menší struktury

PERT chart (Program Evaluation and Review Technique) - analýza úkolu a rozvržení času na dokončení jednotlivých částí rozložení projektu na menší struktury

GANTT chart - zobrazuje rozložení a stav jednotlivých částí v čase

SVN - synchronizace kódu, verzování,...

1 Základy modelování dat, E-R diagramy, relační model. Integritní omezení, normální formy. Základy jazyka SQL, referenční integrita, agregační funkce, vnořené dotazy. Transakce, jejich serializovatelnost, zamykání, stupně izolovanosti, uváznutí transakcí, jeho prevence a řešení. Objektově-relační mapování, persistence objektů. (A4B33DS)

1.1 Základy modelování dat

- **Konceptuální :** Na této úrovni se snažíme popsat předmětnou oblast (obsah) datové základny. V žádném případě nebereme v úvahu jakékoli pozdější způsoby implementace. Konceptuální návrh určuje co je obsahem systému. Nezávisí na použité DB technologii
- **Logické :** Na této úrovni se v relačních databázích používá tzv. relační schéma. Toto relační schéma obsahuje tabulky, a to včetně jejích sloupců (názvům sloupců odpovídají názvy atributů každé entity). Jsou zde vyznačeny primární a cizí klíče. Logický model stále nesmí být zatížen implementačními specifiky řešení. Logický návrh určuje jak je obsah systémů v dané technologii realizován. Závisí na technologii, ale nezávisí na typu DB.
- **Fyzické :** Popisuje, jak je záznam uložen (např.: o zákazníkovi). Zde vybíráme konkrétní databázovou platformu, ve které bude navrhovaná datová základna vytvořena. Využívají se zde specifika použitého vývojového prostředí (programovací jazyk, konkrétní databázové či vývojové prostředí GUI).

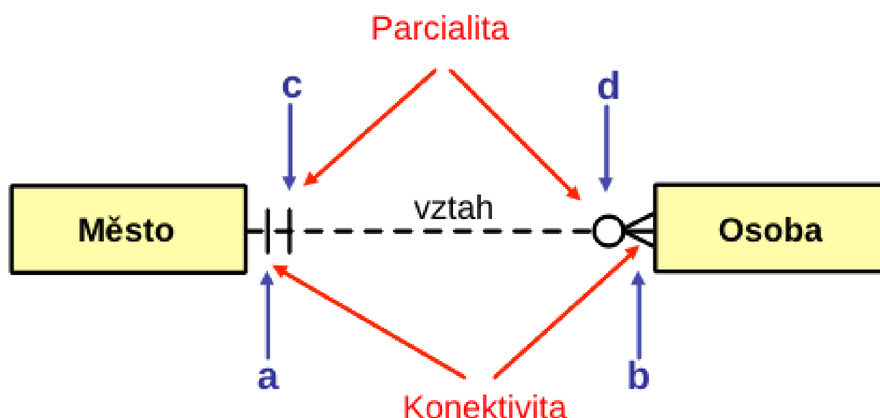
Vzhledem k převaze relačních databází se často nerozlišuje fáze tvorby konceptuálního a logického modelu.

1.2 Relační model

1.2.1 Základní pojmy

- **relace** (relation)**:** Jsou dvourozměrné struktury tvořené záhlavím a tělem (databázové tabulky).
- **entitní typ :** je nějaká “věc” nebo “objekt” jednoznačně odlišitelná od ostatních (entitní typ je dán množinou svých atributů). Obvykle je vyjádřen podstatným jménem.
- **entita :** instance entitního typu (konkrétní řádek v tabulce, obsahuje nějaké hodnoty)
- **atribut :** vlastnost entitního typu. *např.:* Entitní typ student může obsahovat atributy: *jmeno, prijmeni,*
- **doména atributu :** přípustné hodnoty pro atribut
- **vztah** (relationship)**:** zachycuje, jakým způsobem jsou dvě nebo více entit vztažené mezi sebou. Nezaměňovat s relací. Existují 3 typy vztahu mezi relacemi: **1:1, 1:N** (cizí klíč na straně N), **M:N** (využívá vazební tabulku). Obvykle vyjádřen slovesem.
- **klíč (primární klíč) :** je atribut (nejčastěji id) nebo množina atributů (např. autor, název v tabulce knížek). Klíč jednoznačně určuje entitu.
- **super klíč :** Super klíč množiny entit je množina jednoho nebo více atributů, jejichž hodnoty jednoznačně určují entitu (tedy klíč je podmnožina atributů – např. všechny atributy).
- **kandidátní klíč :** Kandidátní klíč množiny entit je minimální super klíč. Rodné číslo je kandidátní klíč entity zákazník, číslo účtu je kandidátní klíč klíč entity účet.
- **cizí klíč :** je atribut, který korensponduje s primárním klíčem v jiné relaci (tabulce). Hodnotami cizího klíče v referencující (odkazující) relaci smí být jen ty hodnoty, které se vyskytují jako primární klíč v relaci referencované (odkazované).
- **slabá množina entit :** při modelování reality se někdy vytváří entitní typy, které nemají samy o sobě význam, Existence slabé množiny entit závisí na množině definujících entit.

1.2.2 Parcialita a Konektivita



1.2.3 Kardinalita vs. Konektivita

Kardinalita (Chen):

Konektivita (také UML):

Kardinalita určuje počet prvků asociované množiny entit (entitního typu) prostřednictvím množiny vztahů.

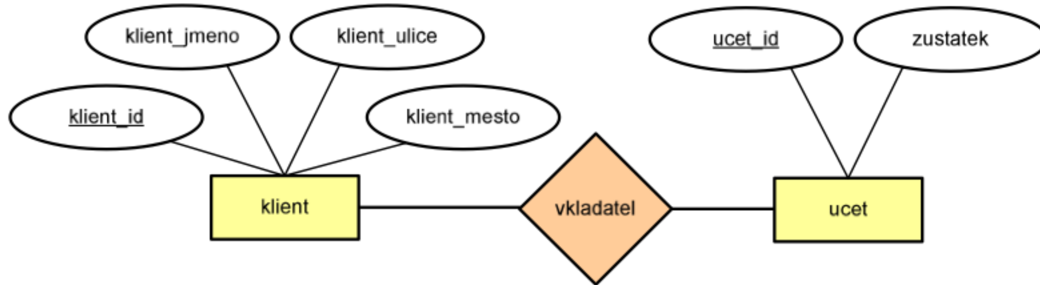
Pro binární vztahy existují 4 typy kardinality:

- **1:1 :** přiřazuje jednomu záznamu jeden jediný záznam v jiné tabulce. Tento vztah se užívá jen ojediněle, protože vetšinou není důvod proč takové záznamy neumístit do jedné tabulky.
- **1:N :** přiřadí jednomu záznamu více záznamů v tabulce jiné. Nejpoužívanější typ relace, odpovídá mnoha situacím v reálném životě.

- **N:1 :** obdobně jako **1:N**.
- **M:N :** méně častý. Umožňuje několika záznamům v jedné tabulce přiřadit několik záznamů v tabulce jiné. V databázové praxi bývá tento vztah z praktických důvodů nejčastěji realizován kombinací dvou vztahů 1:N a 1:M.

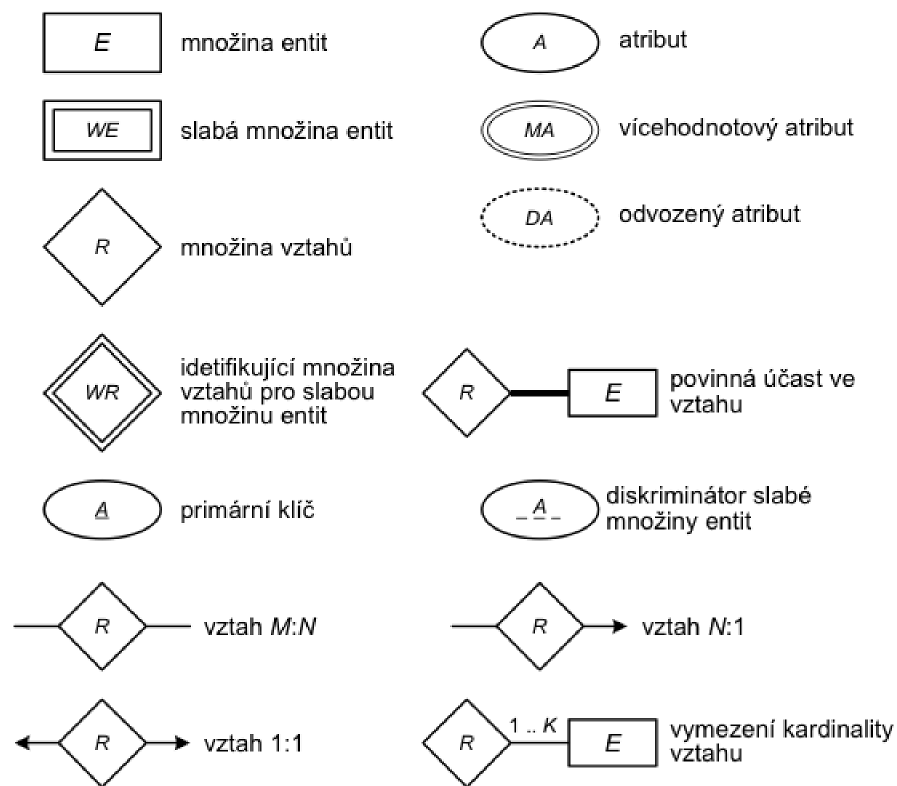
1.3 E-R diagramy

E-R digram je grafická reprezentace E-R (entity-relationship) modelu.



- **obdelníky** - množiny entit (entitní typy)
- **kosočtverce** - množiny vztahů
- **ovály** - atributy
  - zdvojené ovály se používají pro více hodnotové atributy
  - čárkované ovály značí odvozené (počítané) atributy
- **podtržené atributy** - značí primární klíče

1.3.1 Hlavní symboly



1.4 Normální formy

1.4.1 První norální forma (1NF)

Relace je v **1NF** právě tehdy, když platí současně:

- atributy jsou atomické (dále nedělitelné)
- k řádkům relace lze přistupovat podle obsahu (klíčových) atributů
- řádky tabulky jsou jedinečné

Příklad:

Relace nesplňující 1NF:

jmeno	prijmeni	adresa
Josef	Novák	Technická 2, Praha 16627
Petr	Pan	Karlovo náměstí 13, Praha 12135

Relace v 1NF:

jmeno	prijmeni	ulice	cislo	mesto	psc
Josef	Novák	Technivká	2	Praha	16627
Petr	Pan	Karlovo náměstí	13	Praha	12135

1.4.2 Druhá normální forma (2NF)

Relace je v **2NF** právě tehdy, když platí zároveň:

- relace je v 1NF
- každý atribut, který není primárním klíčem je na primárním klíči úplně závislý

Příklad:
Mějme relaci *{IdStudentu, IdPredmetu, JmenoStudenta, Semestr}*, kde *IdStudenta* a *IdPredmetu* tvoří primární klíč. Tato relace není v 2NF, protože *JmenoStudenta* je závislé pouze na *IdStudenta a Semestr* je závislé pouze na *IdPredmetu.*
Řešení:
Rozdělení relace do tří tabulek

- *{IdStudenta, IdPredmetu}*
- *{IdStudenta, JmenoStudenta}*
- *{IdPredmetu, Semestr}*

1.4.3 Třetí normální forma (3NF)

Relace je v **3NF** právě tehdy, když platí:

- relace je v **2NF**
- žádný atribut, který není primárním klíčem, není tranzitivně závislý na žádném klíči

Příklad:
Mějme relaci *{IdStudenta, JmenoStudenta, Fakulta, Dekan} ta* není ve **3NF**. Sice je ve **2NF**, ale atribut Dekan je funčně závislý na *Fakulta* a *Fakulta* je funkčně závislá na *IdStudenta* (předpokládáme, že student nemůže být současně studentem více fakult téže university). *IdStudenta* není funkčně závislá na *Fakulta.* Atribut *Dekat* je tedy transitivně závislý na klíči.
Řešení:
Rozdělíme relaci do tabulek:

- *{IdStudenta, JmenoStudenta, Fakulta}*
- *{Fakulta, Dekan}*

1.5 Integritní omezení

- **Entitní** – povinné integritní omezení, které zajišťuje úplnost primárního klíče tabulky; zamezí uložení dat, která neobsahují všechna pole sdružená do primárního klíče, nebo data, jež by v těchto polích byla stejná jako v nějakém jiném, již zapsaném, řádku tabulky. To znamená, že sloupce zvolené jako primární klíč by měly být unikátní a nenulové.

- **Doménová** – zajišťují dodržování datových typů/domén definovaných u sloupců databázové tabulky
- **Referenční** – zabývají se vztahy dvou tabulek, kde jejich relace je určena vazbou primárního a cizího klíče

1.6 Základy SQL

Structured Query Language (SQL) je jazyk pro kladení dotazů do databáze. Obsahuje jak příkazy DML (Data manipulation Language), tak i DDL příkazy (Data Definition Language). SQL je case insensitive (nerozlišuje mezi velkými a malými písmeny).

1.6.1 Příkazy pro manipulaci s daty

Jsou to příkazy pro získání dat z databáze a pro jejich úpravy. Označují se zkráceně DML – Data Manipulation Language („jazyk pro manipulaci s daty").

- SELECT – vybírá data z databáze, umožňuje výběr podmnožiny a řazení dat.
- INSERT – vkládá do databáze nová data.
- UPDATE – mění data v databázi (editace).
- MERGE – kombinace INSERT a UPDATE – data buď vloží (pokud neexistuje odpovídající klíč), pokud existuje, pak je upraví ve stylu UPDATE.
- DELETE – odstraňuje data (záznamy) z databáze.
- EXPLAIN – speciální příkaz, který zobrazuje postup zpracování SQL příkazu. Pomáhá uživateli optimalizovat příkazy tak, aby byly rychlejší.
- SHOW - méně častý příkaz, umožňující zobrazit databáze, tabulky nebo jejich definice

1.6.2 Příkazy pro definici dat

Těmito příkazy se vytvářejí struktury databáze – tabulky, indexy, pohledy a další objekty. Vytvořené struktury lze také upravovat, doplňovat a mazat. Tato skupina příkazů se nazývá zkráceně DDL – Data Definition Language („jazyk pro definici dat").

- CREATE – vytváření nových objektů.
- ALTER – změny existujících objektů.
- DROP – odstraňování objektů.

1.6.2.1 Příkazy pro řízení dat

Do této skupiny patří příkazy pro nastavování přístupových práv a řízení transakcí. Označují se jako DCL – Data Control Language („jazyk pro ovládání dat“), někdy také TCC – Transaction Control Commands („jazyk pro ovládání transakcí“).

- GRANT – příkaz pro přidělení oprávnění uživateli k určitým objektům.
- REVOKE – příkaz pro odnětí práv uživateli.
- START TRANSACTION – zahájení transakce.
- COMMIT – potvrzení transakce.
- ROLLBACK – zrušení transakce, návrat do původního stavu.

1.7 Referenční integrita

Referenční integrita je nástroj databázového stroje, který pomáhá udržovat vztahy v relačně propojených databázových tabulkách. Referenční integrita se definuje cizím klíčem, a to pro dvojici tabulek, nebo nad jednou tabulkou, která obsahuje na sobě závislá data (například stromové struktury). Tabulka, v niž je pravidlo uvedeno, se nazývá podřízená tabulka (používá se také anglický termín slave). Tabulka, jejíž jméno je v omezení uvedeno, je nadřízená tabulka (master). Pravidlo referenční integrity vyžaduje, aby pro každý záznam v podřízené tabulce, pokud tento obsahuje data vztahující se k nadřízené tabulce, odpovídající záznam v nadřízené tabulce existoval. To znamená, že každý záznam v podřízené tabulce musí v cizím klíči obsahovat hodnoty odpovídající primárnímu klíči nějakého záznamu v nadřízené tabulce, nebo NULL.

1.8 Agregační funkce

Agregační funkce jsou v SQL statistické funkce, pomocí kterých systém řízení báze dat umožňuje seskupit vybrané řádky dotazu (získané příkazem SELECT) a spočítat nad nimi výsledek určité aritmetické nebo statistické funkce. Agregační funkce se v SQL používají s konstrukcí GROUP BY. Agregační funkce pracují s kolekcí hodnot a vrací jedinou výslednou hodnotu.

- **avg :** průměrná hodnota
- **min :** minimum
- **max :** maximum
- **sum :** součet hodnot
- **count :** počet hodnot

1.9 Vnořené dotazy (poddotaz)

Poddotaz je takový dotaz na databázi, který je umístěn uvnitř jiného „vnějšího“ dotazu a výsledky z něj se používají v nějaké podmínce v tom vnějším dotazu. Poddotaz je nejčastěji příkaz SELECT a poskytuje hodnoty do porovnávací podmínky (klauzuli WHERE) pro nadřazený dotaz (jiné části dotazu jen velmi zřídka). Používá se tam, kde není vhodné nebo možné použít agregační funkce nebo (pro dodržení kompatibility) uložené procedury.

Příklad:

*SELECT * FROM tabulka1 WHERE sloupec1=(SELECT sloupec2 FROM tabulka2 WHERE podmínka);*

Příklad:

DELETE FROM tabulka1 WHERE sloupec1 IN (SELECT sloupec2 FROM tabulka2 WHERE podmínka);

Příklad:

UPDATE tabulka1 SET sloupec1=hodnota1 WHERE EXISTS(SELECT sloupec2 FROM tabulka2 WHERE podmínka);

1.10 Transakce

- transakce je posloupnost operací (část programu), která přistupuje a aktualizuje (mění) data.
- Transakce pracuje s konzistentní databází.
- Během spouštění transakce může být databáze v nekonzistentním stavu.
- Ve chvíli, kdy je transakce úspěšně ukončena, databáze musí být konzistentní.
- Dva hlavní problémy:
  - Různé výpadky, např. chyba hardware nebo pád systému
  - Souběžné spouštění více transakci

1.10.1 ACID vlastnosti

K zachování konzistence a integrity databáze, transakční mechanismus musí zajistit:

- **A**tomicity : transakce atomická - buď se podaří a provede se celá nebo nic. Nelze vykonat jen část transakce.
- **C**onsistency : transakce - konkrétní transformace stavu, zachování invariant - integritní omezení.
- **I**solation : (isolace = serializovatelnost). I když jsou transakce vykonány zároveň, tak výsledek je stejný. jako by byly vykonány jedna po druhé.

- **D**urability : po úspěšném vykonání transakce (commit) jsou změny stavu databáze trvalé a to i v případě poruchy systému - zotavení chyb.

1.10.2 Akce

- **akce na objektech :** READ, WRITE, XLOCK, SLOCK, UNLOCK
- **akce globální :** BEGIN, COMMIT, ROLLBACK

1.10.3 Stavy transakce

- Aktivní – počáteční stav; transakce zůstává v tomto stavu, dokud běží
- Částečně potvrzená (Partially Committed) – jakmile byla provedena poslední operace transakce
- Chybující (Failed) – po zjištění, že normální běh transakce nemůže pokračovat
- Zrušená (Aborted) – poté, co byla transakce vrácena (rolled back) a databáze byla vrácena do stavu před spuštěním transakce. Dvě možnosti po zrušení transakce:
  - Znovu spustit transakci – pouze pokud nedošlo k logické chybě
  - Zamítnout transakci
- Potvrzená (Committed) – po úspěšném dokončení

1.10.4 Transakční historie (rozvrh transakcí)

Posloupnost akcí několika transakcí, jež zachovává pořadí akcí, v němž byly prováděny.

Historie (rozvrh) se nazývá sériová, pokud jsou všechny kroky jedné transakce provedeny před všemi kroky druhé transakce.

	Serializovatelá historie	
Krok	T_1	T_2
1	**BOT**	
2	READ(A)	
3		**BOT**
4		READ(C)
5	WRITE(A)	
6		WRITE(C)
7	READ(B)	
8	WRITE(B)	
9	**COMMIT**	
10		READ(A)
11		WRITE(A)
12		**COMMIT**

Sériová historie	
T_1	T_2
BOT	
READ(A)	
WRITE(A)	
READ(B)	
WRITE(B)	
COMMIT	
	BOT
	READ(C)
	WRITE(C)
	READ(A)
	WRITE(A)
	COMMIT

	Neserializovatelá historie	
Krok	T_1	T_2
1	**BOT**	
2	READ(A)	
3	WRITE(A)	
4		**BOT**
5		READ(A)
6		WRITE(A)
7		READ(B)
8		WRITE(B)
9		**COMMIT**
10	READ(B)	
11	WRITE(B)	
12	**COMMIT**	

1.10.5 Serializovatelnost

1.10.5.1 Teorie

Nechť se transakce T_i skládá z následujících elementárních akcí:

- $\mathbf{READ_i(A)}$ - čtení objektu A v rámci transakce T_i

- **WRITE$_i$(A)** - zápis (přepis) objektu A v rámci transakce T_i
- **ROLLBACK$_i$** - přerušení transakce T_i
- **COMMIT$_i$** - potvrzení transakce T_i

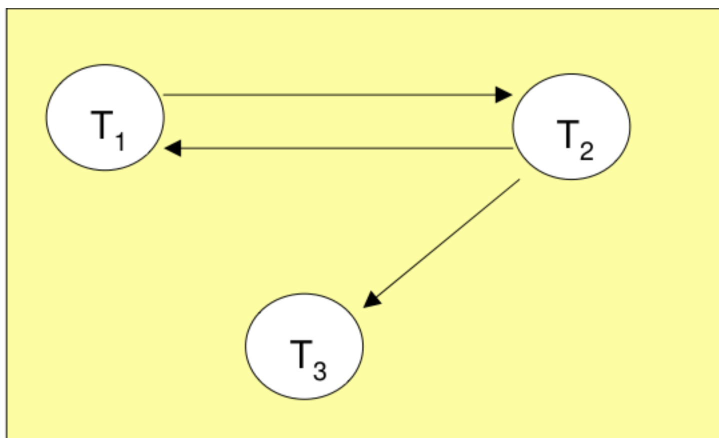
Jsou možné 4 případy:

READI(A) - READJ(A)	Není konflikt	Na pořadí nezávisí
READI(A) - WRITEJ(A)	**Konflikt**	Pořadí má význam
WRITEI(A) - READJ(A)	**Konflikt**	Pořadí má význam
WRITEI(A) - WRITEJ(A)	**Konflikt**	Pořadí má význam

Zajímavé jsou navzájem konfliktní operace: Dvě **historie H1** a **H2** (na téže množině transakcí) jsou **ekvivalentní**, pokud jsou **všechny konfliktní operace** (nepřerušených) transakcí **provedeny v témže pořadí**.

To znamená, že pro dvě ekvivalentní historie a uspořádání <H1 indukované historií H1 a <H2 indukované historií H2 platí: pokud pi a qj jsou konfliktní operace takové, že pi $<_{H1}$ qj , musí platit také pi $<_{H2}$ qj . Pořadí nekonfliktních operací není zajímavé.

- Základní předpoklady – každá transakce zachovává konzistenci databáze
- Tedy sériový plán zachovává konzistenci databáze
- Plán je serializovatelný, když je ekvivalentní sériovému plánu. Různé formy ekvivalence plánů vedou k následujícím pojmům:
  - Konfliktní serializovatelnost
  - Pohledová serializovatelnost
- Ignorujeme všechny instrukce kromě **čtení** a **zápisu** a předpokládáme, že transakce mohou provádět libovolné výpočty na datech v lokálních vyrovnávacích pamětech mezi čteními a zápisy. Naše zjednodušené plány se skládají pouze z operací **čtení** a **zápisu**.



Historie H je serializovatelná právě tehdy, když její závislostní graf nemá cykly.

Transakce je serializovatelná (s výjimkou fantom problémů), práve když:

- je dobře formulovaná (všechny akce prokyty zámky)
- zamykat výhradně všechny data. jejichž obsah modifikuje (legální)
- je dvoufázová - neměla by uvolňovat zámky dříve než budou všechny zámky aplikovány
- výhradní zámky drží až do COMMIT/ROLLBACK

Serializovatelnost se řeší pomocí:

- zamykání (locking) na různé úrovni granularity:
- zamykání celého systému (=>sériovost)
- jednotlivých tabulek
- Jednotlivých záznamů (vět)
- časové značky
- MVCC (multiversion concurrency control)
- predikátové zámky

1.10.6 Zamykání

Jedním ze způsobů, jak zajistit požadavek sériovosti, je zpřístupnit data vždy jen jediné transakci. Když jedna transakce získá k údaji výlučný (exklusivní) přístup, pak tento údaj nemůže modifikovat jiná transakce dříve, než první transakce skončí a uvolní přístup k údaji - a to i v případě, že byla při paralelním zpracování několikrát přerušena. Říkáme, že údaje jsou zamčeny. Jediný klíč ke každému zámku (při modifikaci) přiděluje systém pro řízení paralelního zpracování těm transakcím, které o něj požádají.

Existuje několik úrovní zamykání údajů (CO se zamyká):

1. Na úrovni operačního systému definujeme soubor typu read-only a tak zakážeme zápis a modifikaci všem.
2. Na úrovni SŘBD (Systém řízení báze dat, DBMS) v aplikačním programu definujeme svůj pracovní soubor jako soubor s výlučným přístupem (exclusive). Tak zamezíme přístup všem ostatním procesům, dokud náš program neskončí a neuvolní soubor. Použijeme příkaz k uzamčení a uvolnění souboru, říkáme, že soubor zamykáme explicitně. V SŘBD existují příkazy pro práci se souborem, které vyžadují výlučný přístup k souboru a tak si uzamykají soubor automaticky.
3. V aplikačním programu stačí často zamknout jen jeden nebo několik záznamů, ne celý soubor, aby tak byly ostatní záznamy přístupné ostatním uživatelům. Opět zamykání záznamů může být explicitní nebo automatické.
4. Některé SŘBD umožňují zamykat dokonce jen jednotlivé položky.

Rozlišujeme **zámky dvou** základních **druhů** (JAK se zamyká):

1. zámky pro sdílený přístup (shared) umožňují údaje jen číst více transakcím současně, ne však do nich zapisovat (**SLOCK**)
2. zámky výlučné (exclusive) umožní čtení i zápis vždy pouze jediné transakci (**XLOCK**).

Pokud má jedna transakce údaj (soubor, záznam) uzamčený a další transakce jej chce uzamknout také, může dojít ke kolizi. Proto v SŘBD existují funkce testující, zda je údaj volný. Pokud není, je nutno situaci programově řešit (počkat na uvolnění, zrušit transakci ap.).

Způsob zamykání (KDO zamyká):

1. Aplikační program (programátor) explicitním příkazem
2. SŘBD automaticky (implicitně) současně s některým příkazem pro manipulaci s daty

Použití zámků však není jednoduché, nesprávné použití může vést k nesprávným výsledkům. Důvodem může být například uvolnění zámku příliš brzy (může dojít k nekonzistenci)

Dobře definovaná transakce:

- Před každou operací READ se na daném DB objkektu uplatní zámek SLOCK,
- před každou operací WRITE se na daném DB objektu uplatní zámek XLOCK
- operace UNLOCK se na daném DB objektu může provést pouze tehdy, když je na daném DB objektu uplatněn zámek SLOCK/XLOCK
- každá operace SLOCK/XLOCK je v někdy v následujícím běhu transakce následována příslušnou akcí UNLOCK.

1.10.7 Jednoduchá transakce

1. Obsahuje akce READ, WRITE, XLOCK, SLOCK a UNLOCK
2. COMMIT se nahradí sekvencí příkazů UNLOCK A, pro každý objekt A, na který bylo v průběhu transakce aplikováno SLOCK A nebo XLOCK
3. ROLLBACK se nahradí sekvencí:
   - WRITE A pro každý objekt A, na nějž T aplikovala akci WRITE A
   - UNLOCK A pro každý objekt A, na nějž T aplikovala akci SLOCK A nebo XLOCK A

1.10.8 Dvoufázová transakce

Dvoufázové transakce Všechny akce LOCK jsou provedeny před všemi akcemi UNLOCK. Fáze vzrůstu (growing phase) - během ní se provedou všechny akce LOCK Fáze poklesu (shrinking phase) - během ní se provedou všechny akce UNLOCK. U dvoufázové transakce se fáze vzrůstu a fáze poklesu nepřekrývají.

1.10.9 Stupně izolace

	Transakce	**Názvy**	**Protokol zamykání**
0°	0° T nepřepisuje dirty data jiné transakce, je-li tato stupně 1° a více	anarchie	dobře formulován pro WRITE
1°	1° T nemá lost updates	browse	dvoufázový pro XLOCK a dobře formulovaný pro WRITE
2°	2° nemá lost updates a dirty reads		dvoufázový pro XLOCK a dobře formulovaný pro WRITE a READ
3°	3° nemá lost updates, dirty reads a má repeatable reads	isolovaná transakce serializovatelná opakovatelné čtení	dvoufázový pro XLOCK i SLOCK a dobře formulovaný pro WRITE a READ

1.10.10 Uváznutí transakcí a jeho prevence

proces T3	**proces T4**	
LX(B)		
read(B)		
B:=B-50		
write(B)		

	LX(A)	
	read(A)	
	LX(B)	... marně čeká na uvolnění položky B

LX(A)		... marně čeká na uvolnění položky A
...		

Takovéto situaci, kdy obě transakce čekají, nelze žádný požadavek uspokojit a celý proces uvázne v mrtvém bodě nazýváme uváznutím (deadlock). Problém tedy je v tom, že pokud používáme zámků málo, hrozí nekonzistence, používáme-li zámků mnoho, hrozí uváznutí.

Máme nyní dva problémy: splnění požadavku sériovosti a řešení uváznutí v mrtvém bodě.

1 Požadavek sériovosti

K řešení prvního problému, požadavku sériovosti, se používá tzv. protokolu o zámcích. Je to řada pravidel udávajících, kdy může transakce zamknout a uvolnit objekty.

Pro prevenci uváznutí existuje více technik.

Nejjednodušší metodou prevence uváznutí je uzamčení všech položek, které transakce používá, hned na začátku transakce ještě před operacemi a jejich uvolnění až na konci transakce. Tak se transakce nezahájí dříve, dokud nemá k dispozici všechny potřebné údaje a nemůže dojít k uváznutí uprostřed transakce. Tato metoda však má dvě velké nevýhody:

1. využití přístupu k položkám je nízké, protože jsou dlouhou dobu zbytečně zamčené
2. transakce musí čekat až budou volné současně všechny údaje, které chce na začátku zamknout, a to může trvat velmi dlouho.

Jiná metoda prevence uváznutí využívá faktu, že k uváznutí nedojde, jestliže transakce zamykají objekty v pořadí respektujícím nějaké lineární uspořádání, definované nad těmito objekty (např. abecední ap.). Z hlediska uživatelského však takový požadavek je příliš omezující.

Plánovače

Některé systémy řeší problém uváznutí synchronizací paralelních transakcí pomocí plánovače. V SŘBD jsou zabudovány tyto programové moduly

- Modul řízení transakcí (RT); je to fronta, na kterou se transakce obracejí se žádostí o vykonání operací READ(X) a WRITE(X). Každá transakce je doplněna příkazy BEGIN TRANSACTION a END TRANSACTION.
- Modul řízení dat (RD) realizuje čtení a zápis objektů dle požadavků plánovače a dává plánovači zprávu o výsledku a ukončení.
- Plánovač zabezpečuje synchronizaci požadavků z fronty dle realizované strategie a řadí požadavky do schémat.
- Schéma pro množinu transakcí je pořadí, ve kterém se operace těchto transakcí realizují.

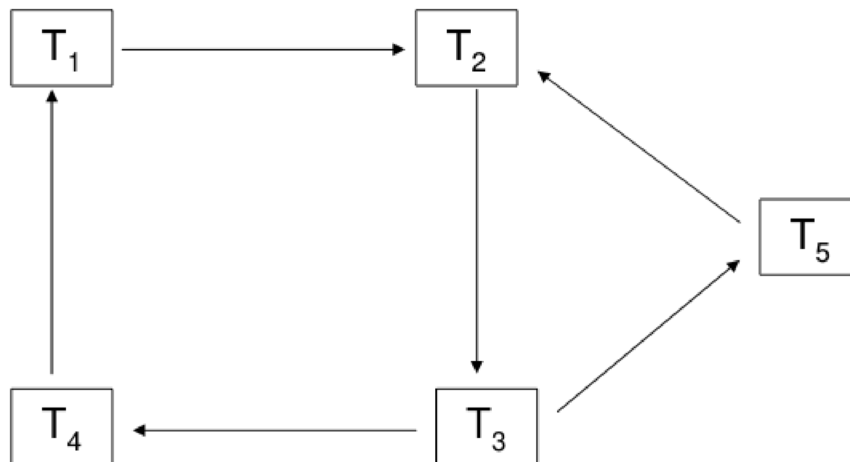
Nejjednodušší schéma je sériové (vždy proběhne celá transakce, pak další), ovšem je málo průchodné. Cílem celé strategie je větší průchodnost systému.

Plánovač při dvoufázovém zamykání vykonává tyto operace

- řídí zamykání objektů

- operace čtení a modifikace objektů povoluje jen těm transakcím, které mají příslušné objekty zamknuté
- sleduje, jestli transakce dodržují protokol dvoufázového zamykání; pokud zjistí jeho porušení, transakci zruší
- předchází uváznutí nebo ho detekují a řeší zrušením transakce.

Jestliže systém nepoužívá prevenci uváznutí, musí mít prostředky pro detekci (rozpoznání) uváznutí a obnovu činnosti umrtvených transakcí. Detekce se provádí obvykle použitím grafu relace "kdo na koho čeká". Je to graf, jehož uzly jsou transakce a orientované hrany představují uvedenou závislost. Záznamem a analýzou grafu čekání se rozpoznává uváznutí. Je-li v grafu cyklus, systém uvázl v mrtvém bodě.



Odstranění cyklů – strategie:

- Přerušovat co nejmladší transakci (ovlivnit co nejméně dalších transakcí)
- Přerušovat transakci s max. počtem zámků
- Nepřerušovat transakci, která byla již vícekrát přerušena
- Přerušit transakci, která se účastní více cyklů

Jestliže taková situace nastane, systém musí jednu nebo více transakcí vrátit zpět (pomocí souboru log), čímž se zablokovaný přístup k datům (pro tuto transakci) odblokuje a umožní provést ostatní transakce. Připomíná to situaci, kdy se dva automobily potkají na úzké cestě a jeden musí vycouvat.

Obnovení činnosti se provádí pomocí souboru log, popsaného v předchozí kapitole. V případě potřeby je možno kteroukoliv transakci vrátit. Jde jen o to, kdy a které transakce se mají provést znovu. Systém vybírá takové transakce, aby s celým postupem byly spojeny co nejmenší náklady, k tomu bere v úvahu:

- jaká část transakce již byla provedena,
- kolik dat transakce použila a kolik jich ještě potřebuje pro dokončení,

- kolik transakcí bude třeba celkem vrátit.

Podle těchto kriterií by se mohlo dále stát, že bude vracena stále tatáž transakce a její dokončení by bylo stále odkládáno. Je vhodné, aby systém měl evidenci o vracených transakcích a při výběru bral v úvahu i tuto skutečnost.

1.11 Objektově-relační mapování (ORM)

Onjektově-relační mapování je programovací technika v softwarovém inženýrství, která zajišťuje automatickou konverzi dat mezi relační databází a objektově orientovaným programovacím jazykem.

Hlavním cílem ORM je synchronizace mezi používanými objekty v aplikaci a jejich reprezentací v databázovém systému tak, aby byla zajištěna persistence dat.

Řada implementací ORM se snaží v co největší míře odstínit vývojáře od nutnosti psaní SQL dotazů a pro selekci objektů z databáze používá raději objektový přístup. Takovýto postup však zpravidla umožňuje vyhledávat objekty jen podle databázového primárního klíče, což zpravidla nestačí. Proto některé implementace ORM využívají pro selekci objektů objektový dotazovací jazyk. Jedna z výhod odstínění od práce s SQL může být i určitá nezávislost aplikace na konkrétním databázovém systému, resp. možnost zvolit databázový systém či jiné datové úložiště tak, aby vyhovovalo konkrétním podmínkám a požadavkům. Nezávislost na konkrétním databázovém systému a skrývání SQL dotazů jsou však již jen příjemné důsledky použití ORM, není to ale primárním cílem.

1.11.1 Pesistence objektů

Java Persistence API (JPA) je framework programovacího jazyka Java, který umožňuje objektově relační mapování (ORM). To usnadňuje práci s ukládáním objektů do databáze a naopak. Je určen jak pro Java SE, tak pro Java EE.

7 Koncept jazyka na bázi virtuálniho stroje, JVM memory management, datové struktury, výjimky, objektové programování, vlákna a synchronizace. (A4B77ASS)

7.1 Typy programovacích jazyků

Z hlediska vykonávání zdrojového kódu lze programovací jazyky dělit do dvou kategorií.

- **Interpretované** - přímo zdrojový kód (skript či překompilovaný byte-code) je interpretován virtuálním strojem, který běží na cílovém zařízení
- **Kompilované** - zdrojový kód je nutné zkompilovat do strojového kódu cílového zařízení a poté lze program spustit přímo

Vlastnosti interpretovaných jazyků:

- $\oplus$ nezávislost na platformě - architektura (RISC/CISC), operační systém
- $\oplus$ reflexe - sledování a modifikace kódu za běhu
- $\oplus$ dynamické typování
- $\oplus$ malá velikost zdrojových souborů
- $\ominus$ pomalejší vykonávání kódu v interpretovaném módu

7.2 Java Virtual Machine (JVM)

Je zásobníkově orientovaný virtuální stroj Javy, který interpretuje Java byte-code. Zdrojové kódy (`.java`) je proto nutné zkompilovat do byte-code (`.class`), při kompilaci nedochází k žádným optimalizacím kódu. Před spuštěním je byte-code verifikován (skoky jsou pouze na validní umístění, správná inicializace dat, type-safe reference, kontrola private a protected přístupů). Při běhu se používá JIT.

Zásobníkový způsob předávání parametrů: $(2+3) \times 11 + 1$

Input	2	3	add	11	mul	1	add
Stack		3		11		1	
	2	2	5	5	55	55	56

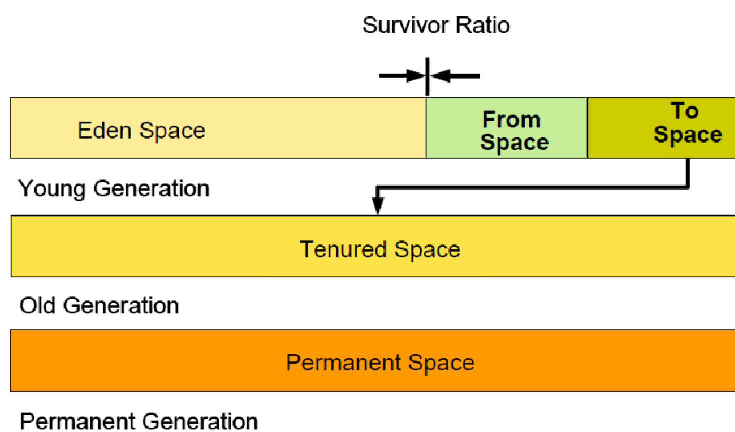
7.2.1 Just-in-time compiler (JIT)

Za běhu konvertuje byte-code na nativní strojový kód. Existují různé verze pro serverové a klientské aplikace (různý stupeň optimalizace). Některé prováděné optimalizace: inlining of functions, loop unrolling, dead code elimination, loop invariant hoisting, common subexpression elimination, constant propagation, optimize branches.

7.3 JVM memory management

V JVM je použita automatická správa paměti pomocí **Garbage Collectoru (GC)** - živé (dosažitelné) objekty jsou ponechány v paměti a mrtvé (nedosažitelné) jsou smazány. Halda (**Heap**) je oblast pro dynamickou alokaci paměti pro všechny objekty a je rozdělena do secí, tzv. generací - Young a Old.

Generační koncept Objekty jsou v haldě rozděleny do generací podle svého stáří, zpočátku jsou alokovány v Young generaci, pokud přežijí několik cyklů GC jsou povýšeny (Tenuring) do Old generace. Koncept je postaven na hypotéze, že většina objektů je krátko-žijících, tj. nedostanou se do Old generace.



Young malá velikost, časté a rychlé cykly GC

Old velká velikost, málo časté a pomalé cykly GC

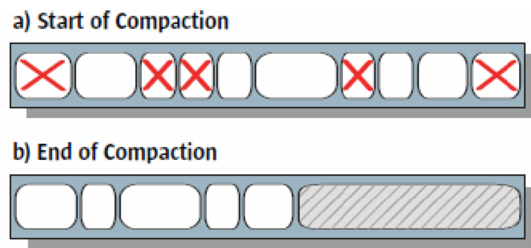
Eden místo pro alokaci nových objektů

From/To místo kam jsou zkopírovány přeživší objekty po běhu GC

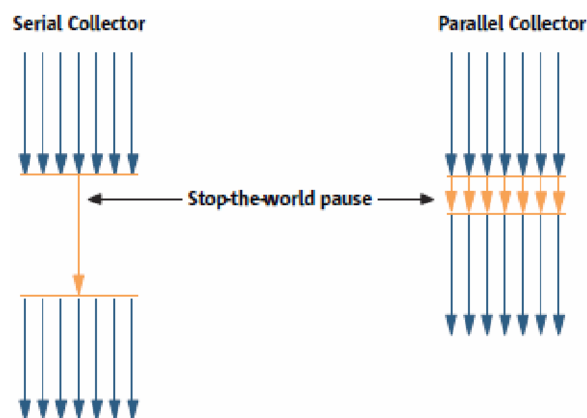
Permanent je mimo haldu, obsahuje data pro JVM jako definice tříd, metod a další

7.3.1 Druhy Garbage Collectorů

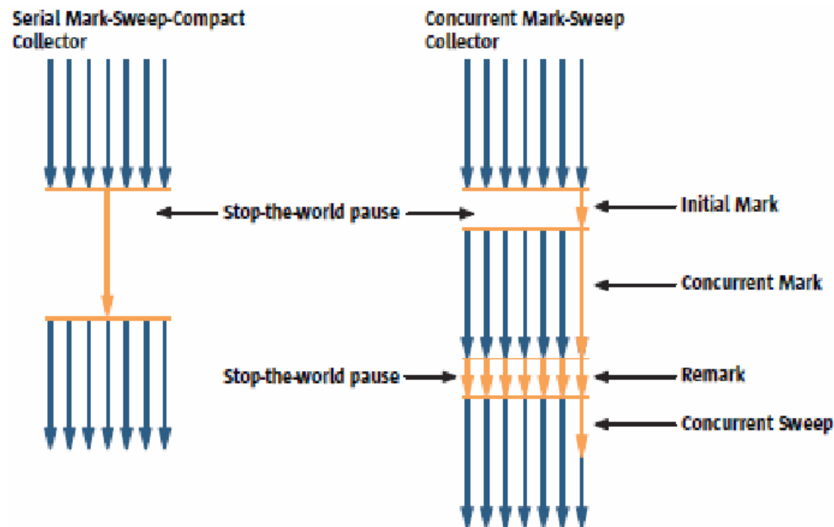
- **Sériový collector** - používá algoritmus *mark-sweep-compact* způsobem *stop-the-world*, výchozí pro klientské aplikace, efektivní na 64MB haldě



- **Paralelní collector** - pro Old generaci obyčejný sériový collector, pro Young generaci navíc využívá více vláken/CPU, výchozí pro serverové aplikace a na vícejádrových systémech



- **Souběžný (Concurrent) collector** - pro Young generaci jako paralelní collector, pro Old generaci běží souběžně s aplikací, má nízkou latenci a snižuje počet zastavení aplikace, vyžaduje ale větší haldu



7.4 Java datové struktury

- **primitiva** - bez implicitní alokace, uložené na zásobníku, typy: boolean, byte, char, int, long, float, double
- **objekty** - každý je potomek třídy Object, uloženy na haldě, existují objekty pro primitiva: Boolean, Byte, Char, Integer, Long, Float, Double
- **pole (arrays)** - speciální datová struktura pro uchovávání více primitiv/objektů stejného typu v lineárním pořadí, mají definovaný limit, který je automaticky za běhu kontrolován, jsou uložené na haldě, více-dimenzionální pole = pole polí

7.4.1 Autoboxing, Unboxing, Widening

Autoboxing automatická konverze primitivních datových typů na jejich objektové reprezentace

Unboxing opačný postup, objekty na primitiva

Widening automatická konverze menších primitiv na větší:
byte $\prec$ char $\prec$ int $\prec$ long a float $\prec$ double

Používání boxingu a unboxingu přináší neefektivitu. V případě více možností má widening přednost před autoboxingem. Při volání přetížených metod Java vybere tu s nejvíce specifickými typy parametrů.

7.4.2 Výjimky

Všechny extendují třídu Throwable (Error, Exception) a slouží k reprezentaci různých chybových stavů aplikace. Výjimky lze dělit do dvou kategorií - kontrolované (Exception) a

nekontrolované (Error a RuntimeException). Kontrolované výjimky je nutné odchytávat pomocí **try-catch** bloku. Pro vyhození vlastní výjimky v programu souží příkaz **throw**. Výjimky také obsahují záznam obsahu zásobníku ve chvíli kdy byla tato výjimka vyhozena a případně další informace o vzniklé chybě.

7.5 Vlákna a synchronizace - Java

7.5.1 Vlákna (Threads)

Vlákna umožňují souběžné vykonávání více úkolů najednou, vytváří se buď extendováním třídy Thread nebo implementací rozhraní Runnable. Hlavní je metoda **run()** obsahující kód, který vlákno vykonává. Jeden proces může mít více vláken, která sdílí společný adresní prostor, ale všechny mají vlastní zásobník a lokální proměnné. Každé vlákno má: ID, jméno, prioritu, thread group, uncaught exception handler, deamon flag, class loader, interrupted flag, status.

7.5.2 Synchronizace

Při práci s více vlákny je často nutná jejich synchronizace a umožnění bezpečného přístupu ke sdíleným prostředkům. K tomu existuje několik technik.

- **Synchronized** - S každým objektem je asociován tzv. *monitor*, který umožňuje synchronizaci vláken nad tímto objektem použitím bloku **synchronized**. Pro vstup do tohoto bloku je nutné aby příslušné vlákno vlastnilo tento monitor, v jednu chvíli ale může monitor vlastnit pouze jedno vlákno. Pokud není monitor pro vlákno dostupný, jeho běh je zastaven dokud se monitor neuvolní. Pro synchronizaci celé metody je možné použít klíčové slovo **synchronized**.
- **Reentrant Locks** - Obdoba předchozí synchronizační techniky s manuálním získáváním a uvolňováním zámků - metody: **lock()**, **unlock()**, **tryLock()**. Zámky je nutné manuálně vytvářet jako objekty ReentrantLock, zámky je možné vytvořit *fair*, tj. první čekající vlákno bude první kdo získá zámek. Používání Reentrant zámků je v praxi efektivnější než použití synchronized, ale zámky jsou běžné objekty na haldě.

  ```
  lock.lock();
  try {
      // do some staff ...
  } finally {
      lock.unlock()
  }
  ```

- **Volatile proměnné** - Jsou obyčejné proměnné, které ale mají zaručen atomický read/write přístup. Tyto proměnné nejsou nikdy uloženy lokálně pro jednotlivá vlákna, ale je vždy přistupováno přímo do hlavní paměti. Nevhodné pro *read-update-write* operace. Použití pomocí klíčového slova **volatile**.

- **Atomické proměnné** - Umožňují atomický read/write přístup i *read-update-write* operace. Používají se speciální atomické instrukce procesoru - **CMPXCHG** (compare and exchange) nebo **CAS** (compare and swap). Pro základní datové typy existují AtomicBoolean, AtomicInteger, AtomicLong a AtomicReference s operacemi **get()**, **set()**, **compareAndSet()**, **addAndGet()**, **incrementAndGet()**, **decrementAndGet()**.
- **Neblokující algoritmy** - Synchronizační algoritmy, které nepoužívají zámky ani čekání vláken, ale jsou postavené na atomických CAS operacích. Princip je, že se v nekonečné smyčce kontroluje stav atomické proměnné, dokud není možné bezpečně pokračovat dále. Tento princip výkonnostně převyšuje blokující algoritmy protože většina CAS operací uspěje na první pokus. Další výhodou je, že odstraňuje nadbytečné uspávání vláken a přepínání kontextu.

7.6 Návrhové vzory pro objektové programování

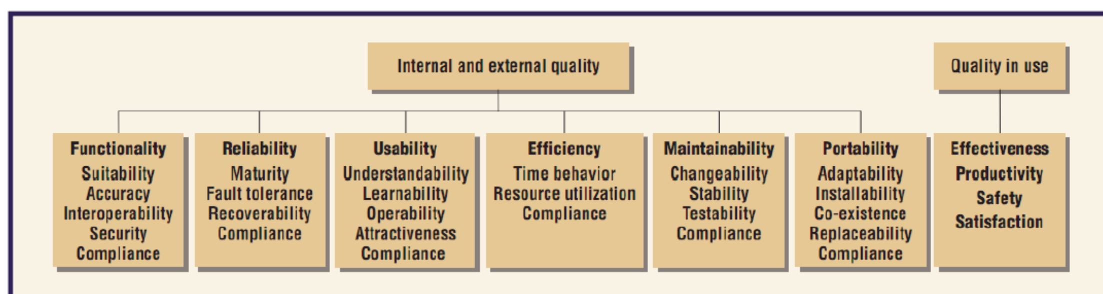
- **Immutable object** - objekt, který po dobu svého života nemění žádné své vlastnosti, all fields are final, no setters
- **Factory method, Abstract factory** - k vytváření objektů se používají speciální metody či tovární třídy
- **Lazy initialization** - k výpočtům, inicializacím objektů a dalších dat dochází až v případě prvního použití
- **Singleton** - pro danou třídu existuje pouze jedna instance v rámci celé aplikace
- **Multiton** - pro danou třídu existuje několik instancí s různými parametry v rámci celé aplikace
- **Strategy** - definice společného rozhraní pro skupinu podobných algoritmů, umožňuje zaměňovat různé implementace
- **Composite** - skládání objektů do stromových struktur, ke skupině objektů je přistupováno jako k jediné instanci (Nodes and Leafs)
- **Iterator** - umožňuje sekvenční procházení kolekce po jednotlivých prvcích bez odhalení vnitřní struktury této kolekce
- **Command** - zastřešuje veškeré informace potřebné pro volání metod později - client, invoker, receiver

8 Architektury softwarových systémů, komponentové architektury, vzdálená invokace, distribuované komponentové architektury (CORBA), redundance a návrh spolehlivých systémů. (A4B77ASS)

8.1 Nefunkční požadavky softwarových systémů

Specifikace **ISO/IEC 9126**:

- **Functionality** – existence of a set of functions and their specified properties
- **Reliability** - capability of software to *maintain* its level of performance under stated conditions for a stated period of time
- **Usability** - effort needed for use
- **Efficiency** - relationship between the *level of performance* of the software and the amount of *resources* used, under stated conditions
- **Maintainability** - effort needed to *make specified modifications*
- **Portability** - transferred from one *environment* to another



8.2 Návrhové vzory pro distribuované systémy

- **Façade** - zakrývá komplexitu a heterogenitu systému či knihovny za jednoduché rozhraní, často zakrývá více objektů

- **Adapter/Wrapper** - poskytuje mapování mezi dvěma rozhraními se stejnou funkcionalitou, zaručuje kompatibilitu volání
- **Wrapper Façade** - zapouzdřuje funkce a data poskytované ne-objektově orientovaným API pod objektově orientované rozhraní
- **Proxy** - je lokální reprezentace vzdálených objektů, rozhraní nebo knihoven
- **Active Object** - odděluje spouštění metody od její invokace, tím zlepšuje souběžnost a zjednodušuje synchronní přístup k objektům
- **Reactor** - umožňuje událostmi řízeným aplikacím zpracovávání požadavků od jednoho či více klientů
- **Proactor** - umožňuje událostmi řízeným aplikacím zpracovávání požadavků, které jsou vyvolány dokončením asynchronních operací
- **HalfSync/HalfAsync** - odděluje synchronní a asynchronní zpracovávání v souběžných systémech

8.3 Vzdálená invokace - RMI

RMI (Remote Method Invocation) je způsob používání vzdálených objektů jako lokálních, dostupné jen pro Javu. Architektura klient-server: server vytvoří vzdálené objekty, klient tyto objekty získá a může na nich volat metody. Umožňuje načítat definice tříd za běhu.

Vzdálené objekty je možné předat jiné (klientské) VM pomocí reference, poté je na nich možné volat metody. Výpočet pak probíhá na VM serveru. Vzdálené objekty musí extendovat rozhraní Remote a všechny metody musí vyhazovat RemoteException. Rozhraní Task společné pro klienta i server definuje jejich vzájemnou komunikaci. Takto může server vykonávat jakékoliv úkoly, které implementují rozhraní Task, RMI pak za běhu načte definice tříd. Data přesouvané mezi klientem a servem musí být buď primitivní datové typy nebo objekty implementující Serializable nebo Remote.

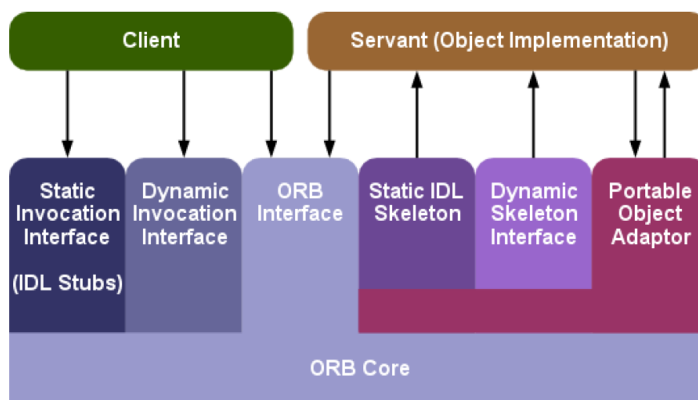
```
public interface Task<T> {
    T execute();
}

public interface Compute extends Remote {
    <T> T perform(Task<T> t) throws RemoteException;
}
```

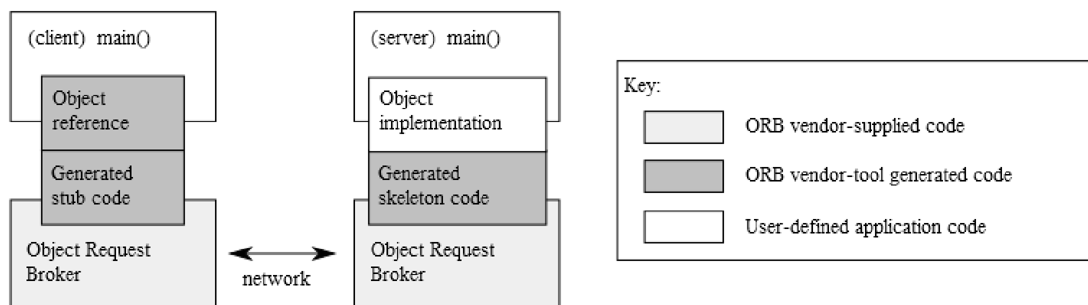
8.4 Distribuované komponenty - CORBA

CORBA (Common Object Request Broker Architecture) je platformně a jazykově nezávislý standard (architektura) pro distribuované výpočty. Umožňuje transparentní invokaci objektů a metod přes síť.

ORB (Object Request Broker) přenáší žádosti od klienta na server a volá metody na vzdálených objektech, potom co server žádost zpracuje, tak ORB přenese odpověď zpět ke klientovi. Klient pak volá metody na lokální proxy. Pro komunikaci mezi jednotlivými ORB a pro přenos dat se používá **IIOP** (Internet Inter-ORB Protocol).

IDL (Interface Definition Language) je platformně a jazykově nezávislý objektově orientovaný jazyk určený ke specifikaci business level služeb a objektů. Používá se pro popis dostupných lokálních a serverových metod.





8.5 Masivně distribuované architektury

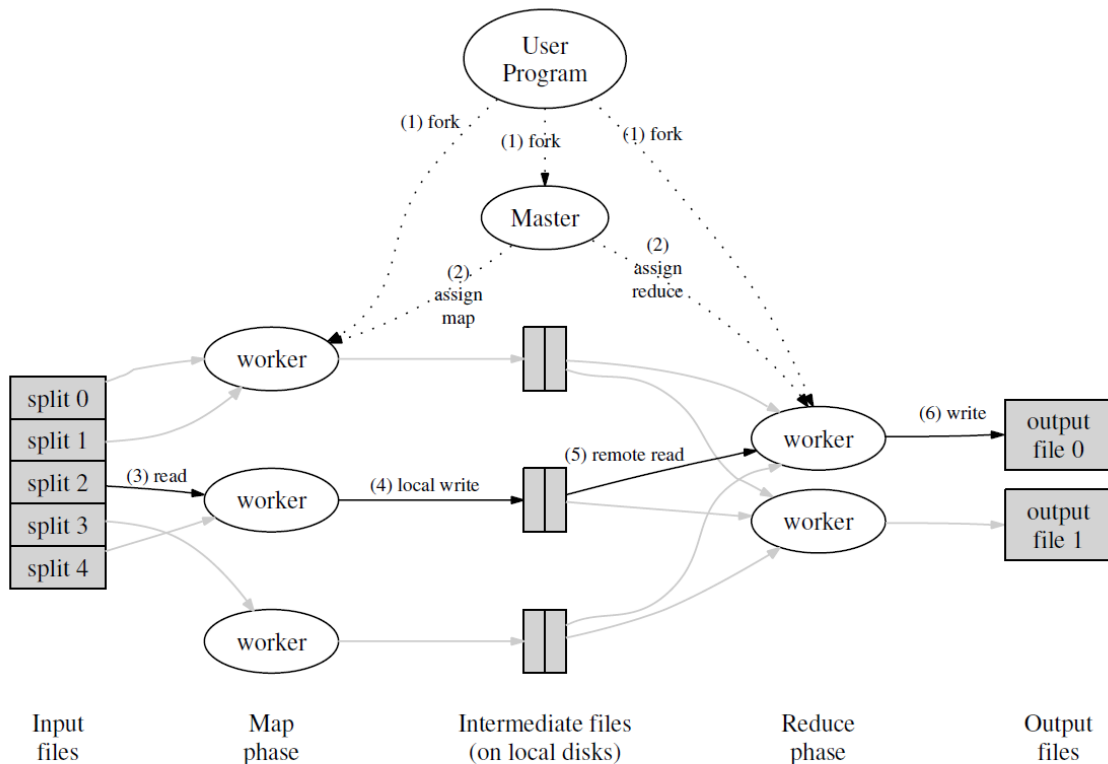
8.5.1 Map-Reduce

Navržen a používán Googlem, masivně paralelní přístup založený a inspirovaný na funkcionálním programování. Používá se k ukládání, manipulaci a hledání v datech s jednoduchou strukturou, kterých je ale velké množství. Operuje nad dvojicemi klíč-hodnota. Metoda Map-Reduce má dvě opakující se fáze:

- **1) Map** - funkce (filtr) se aplikuje na množinu záznamů/elementů v listu
- **2) Reduce** - zkrácení listu za použití nějaké agregační funkce

Z výpočetních jednotek je vybrán jeden řídící *master node* a ostatní jsou *worker nodes*. Masivní paralelizace je dosažena tím, že se data rozdělí do několika částí a nechají se

zpracovat různým *worker nodes*. Příklady použití: distribuované zpracování regulárních výrazu, počítání URL referencí, vytváření reverzního grafu webu, sémantické vyhledávání, invertovaní indexování, distribuované řazení.



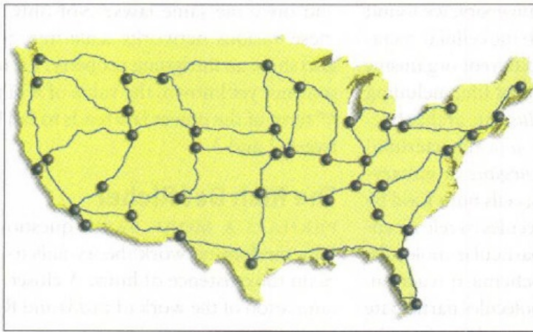
8.5.2 KaZaA

Je skupina protokolů a technologií používaných pro *peer-to-peer* komunikaci.

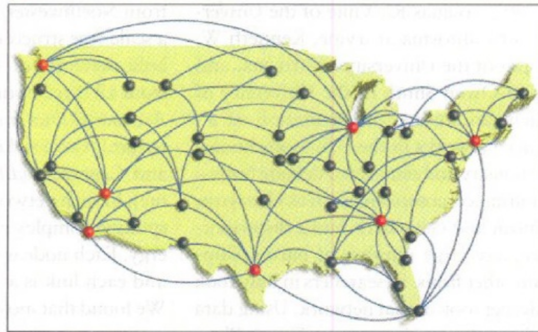
- **Napster** - *peer-to-peer* síť s centralizovaným řídícím prvkem
- **Gnutella** - Je *flat peer-to-peer* síť kde jsou si všichni peery rovni. Při startu se vytvoří daný počet náhodných spojení s peery, které jsou v tu dobu aktivní. Peery sdílí informace o ostatních peerech. V novějších verzích představen koncept *ultrapeerů* (peery s větším množstvím spojení, huby) - design ovlivněný *scale-free* sítěmi.

Scale-Free Networks Sítě reflektující povahu reálného světa. Existují dva typy nodů - *Ordinary Nodes* a *Super Nodes*. Při startu se ON připojí k některému z dostupných SN. Každý SN si udržuje databázi připojených ON a aktuální topologii dalších SN. Při hledání se ON spojí s SN, ale jsou možné spojení přímo ON k ON (Skype).

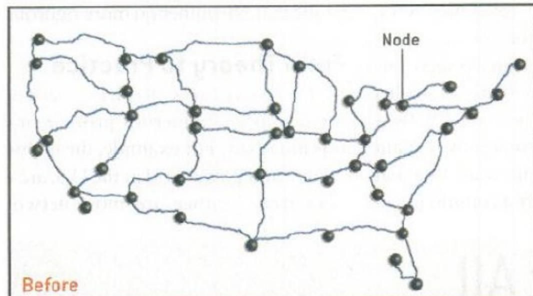
Random Network

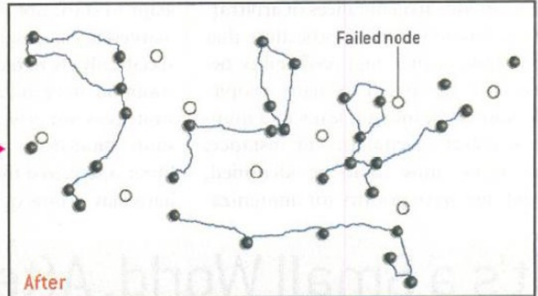


Scale-Free Network

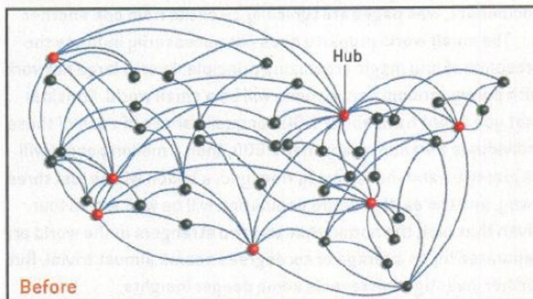


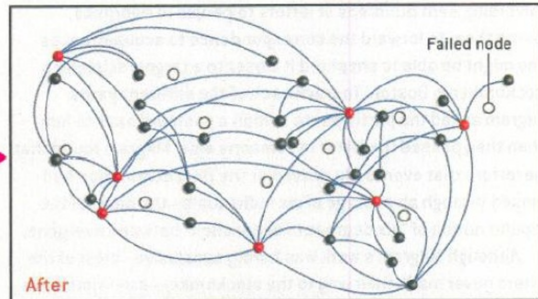
Random Network, Accidental Node Failure



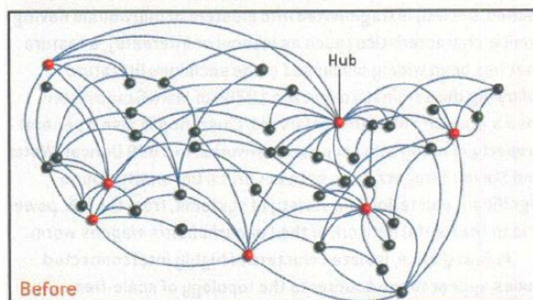


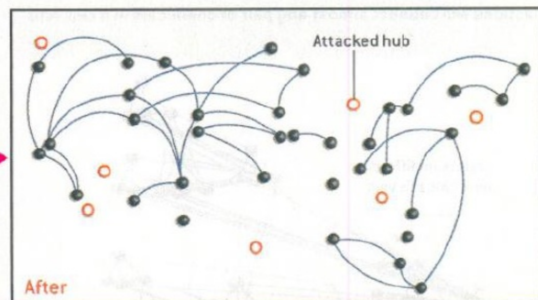
Scale-Free Network, Accidental Node Failure





Scale-Free Network, Attack on Hubs





Sofwarové systémy- otázka č. 9

Webové služby a service-oriented architektury, asynchronní architektury komunikace, producer-consumer model, aktivní objekty a agentní systémy. (A4B77ASS)

June 5, 2012

1 Webové služby

Software navržen tak aby podporoval výpočty mezi stroji za použití sítě. Používá WSDL (Web Service Definition Language) pro specifikaci rozhraní. Komunikuje s ostatními službami za pomoci zpráv popsaných formátem SOAP. Typické použití protokolu HTTP za použití XML serializace dat. Použití UDDI (Universal Description, Discovery and Integration) pro nalezení služeb.

Webové služby typy

RESTfull Web Services

- Hlavním cílem je manipulovat s XML reprezentací webových zdrojů
- Jednotný soubor operací nezávyslý na žadateli

"Big" Web Services

- Výstava libovolného souboru operací
- Může být stavový

Realné příklady

Google analytics, blogger, books, calendar, custom search, latitude, maps, translate API, calendar...

E-bay API pro obchochodování

Amazon reklamní API

Busines-to-Busines nejvýznamější prostor pro užívání webových služeb a service-oriented architekture

Vývoj webovývh služeb

Client-Server Jeden centrální server, který obdržuje požadavky od klientů. Například web.

RPC/DCE Framework pro softwarový vývoj, uveden v letech 1990. Microsoft vytvořil svou vlastní alternativu MSRPC. První distribuované objektové systémy byly založeny na RPC/DCE (CORBA, Microsoft DCOM, RMI). Beží taktéž na client-server architektuře (klient pošlo požadavek obdrží výsledek)

XML-RPC objeveno v letech 1990, podpora pro základní datové typy, zprávava byla kódována v XML.

2 Service-oriented architektury (SOA)

Jedná se o soubor zásad a metodiku pro návrh a vývoj softwaru v podobě spolupracujících služeb. Tyto služby jsou přesně definované obchodní funkce, které jsou postaveny jako softwarové komponenty. Mohou být opakovaně použity pro různé účely. SOA principy platí při fázi návrhu a integraci.

Služby jsou strukturováný jako třídy (kombinace informací, zakrývají vnitřní funkčnost, poskytují jednoduché rozhraní pro zbytek aplikace). Služby mohou tvořit hierarchii.

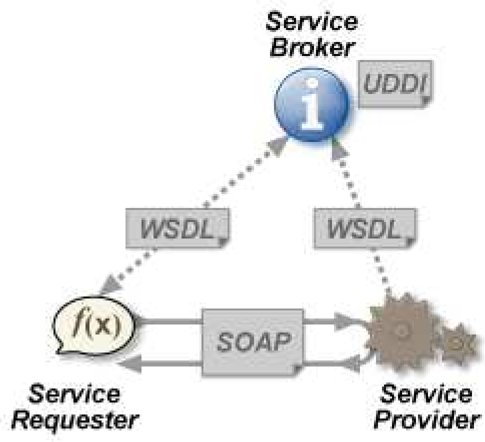
- Poskytuje sadu služeb, která může být použita v rámci více obchodních záměrů.
- Definuje, jak integrovat nesourodé aplikace distribuované na vice systémech
- Definuje rozhraní z pohledu funkcí a protokolů
- Vyžaduje volné spojení se systémem a dalšími technologiemi, které jsou základem aplikace
- Separuje funkce do různých jednotek, nebo služeb, které poskytuje vývojářům po síti

Proč SOA

- Znovupoužití
- Zjednodušené používání starších aplikací
- Platformová nezávyslost
- Vysoká škálovatelnost

Proč ne SOA

- Pokud je důležitý skutečný čas výpočtu


Service
Broker
UDDI
WSDL
WSDL
f(x)
SOAP
Service
Requester
Service
Provider

3 Asynchronní architektury komunikace

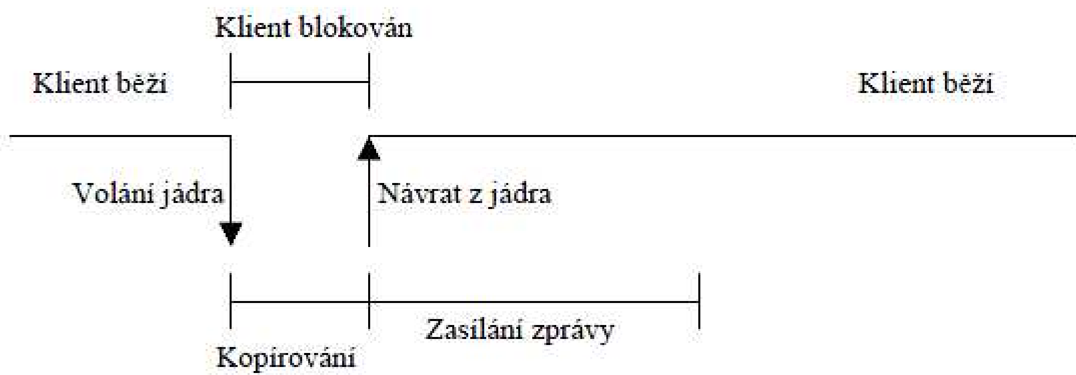
Asynchorní primitiva:

send vrátí kontrolu procesu ihned bez čekání na odeslání zprávy. Výhodou tohoto schématu je vyšší stupeň paralelismu. Proces může pokračovat ve svých výpočtech i po dobu přenosu zprávy. Během přenosu zprávy však proces nesmí používat buffer, ze kterého je zpráva odesílána. To lze vyřešit dvěma způsoby:

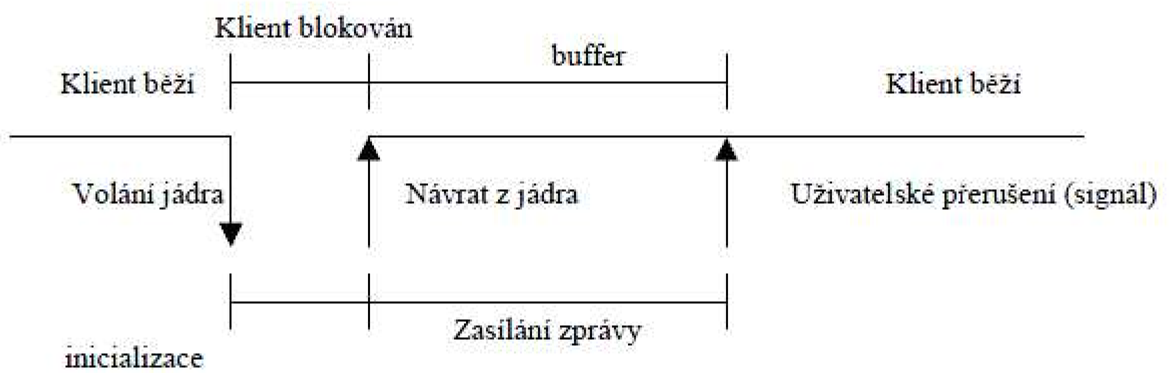
1. Jádro systému si okopíruje zprávu do svých vnitřních bufferů, čímž umožní procesu další volný běh.

2. Po odeslání zprávy je proces přerušen.

receive pouze říká jádru, kam má umístit došlou zprávu. I zde můžeme využít např. tzv. primitiva wait, které pozastaví proces do té doby, než zpráva dorazí. Druhou možností je zavedení primitiva test, které otestuje, zda je nějaká zpráva připravena nebo conditional_retrieve, které buď vybere došlou zprávu nebo se vrátí s informací, že žádná zpráva není k dispozici. Poslední možností je zavedení uživatelského přerušení, které bude informovat o došlé zprávě.



Asynchronní komunikace s kopírováním



Asynchronní komunikace s přerušením

4 producer-consumer problém

Jedná se o typický problém multiprocesové synchronizace. Problém popisuje 2 procesy, producentů a konzumentů, kteří používají jeden buffer používaný jako frontu s pevnou velikostí. Producent generuje data, ukládá je do bufferu a začíná je znovu generovat. Ve stejný čas konzument používá data z bufferu. Problém je, že producent nemůže ukládat data do bufferu, pokud je plný a konzument nemůže brát data, pokud je buffer prázdný.

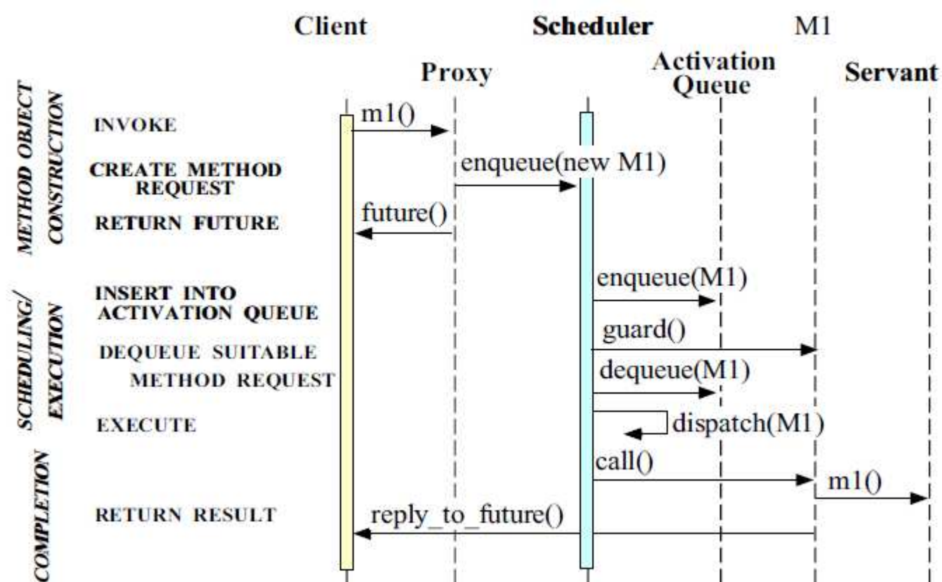
Řešením tohoto problému je, uspat producenta, pokud je zásobník plný a probudit ho, když se uvolní místo. Zároveň pak uspat konzumenta pokud je zásobník prázdný a probudit ho, pokud je v zásobníku něco vloženo.

- Řešení přes standartní sleep, wakeup, kdy se při každé iteraci kontroluje stav zásobníku. Může docházet k deadlocku
- Řešení přes semafory. Procesy si dávají navzájem přednost podle toho v jakém stavu je semafor.
- Řešení přes monitory. Synchornizaci provádí akce add a remove nad zásobníkem. Pokud je zásobník prázdný zamknou konzumenta...

5 Aktivní objekty

Jedná se o návrhový vzor, který odděluje metody provádění výpočtu a volání. Zlepšuje souběžnost a snižuje problémy se synchronizací. Na těchto objektech bývají velmi často založeny Agentní a Multiagentní systémy.

Princip activního objektu je založen na tom, že je vyvolán asynchroní výpočet, kterému jsou předány vstupní parametry a návratová funkce. Objekt který vyvolá asynchroní akci, zahodí na výpočet referenci, a pracuje dál bez ohledu na externím výpočtu (ne tak so slova, pokud je výpočet nezbytný, čeká na něj). Externí výpočet se sám ozve na referenci kterou v sobě uchovává.



6 Agentní systémy

Speciální podmnožina umělých inteligencí. Zkoumá koncept automatického rozhodování, komunikace a koordinace, distribuovaného plánování a učení. Herni aspekty jako chování soutěžícího, nebo logickou formalizaci vyšších struktur na úrovni znalostí.

Agent je zapouzdřený výpočetní systém, který se nachází v nějákém prostředí, a je schopen pružného, samostatného chování za účelem splnění svého cíle. Agent může existovat samostatně, ale často je součástí multi-agentního systému.

Technologie Agentů poskytuje sadu nástrojů, algoritmů a metod pro vývoj a distribuovaných a asynchronních inteligentních softwarových aplikací.

Klíčové vlastnsoti:

- Samostatnost - agent rozhoduje sám za sebe a nemá žádný dozor z venčí
- Reaktivita - agent je schopen rychle reagovat na události v prostředí
- Schopnost zachovat dlouhodobé plány a zvažování dalších kroků k dosáhnutí plánu.
- Sociální schopnosti - je schopen komunikovat a spolupracovat.

Modely užití agenta:

- agent jako metafora - pomáhá vývojářum vývoj ohledně samostatnosti v komunikaci
- zdoj technologii
- simulační - poskytuje simulaci problému realného světa

Úrovně návrhu agenta:

- Organizační úroveň
- Interakční úroveň - komunikace mezi agenty
- Agentní úroveň - týká se samotného agenta (učení...)

Užití agentů:

- Zeměplošné rozložení
- Konkurenční domény

- Časově kritické odevy a vysoká odolnost
- Simulace a modelování scénářů
- Open system scénáře
- Complex system scénáře
- Samostatnostně orientované aspekty

Testování uživatelských rozhraní. Definice pojmu použitelnosti (usability), modely úloh, prototypování uživatelských rozhraní. Testování použitelnosti (organizace a vyhodnocování testů) (A4B39TUR)

June 10, 2012

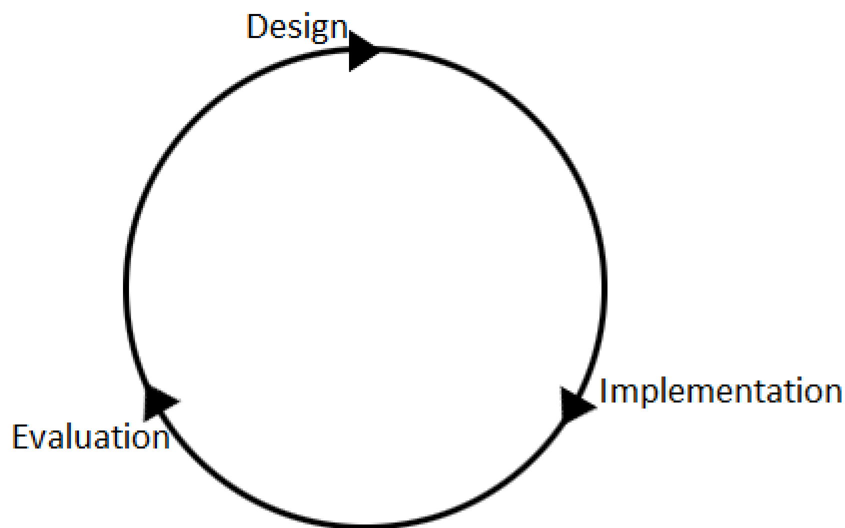
Ověření, jak systém budou používat reálni uživatelé, pochopení jak ho budou používat, identifikace problémů.

1 Usability (použitelnost)

- **jednoduchost použití** – podobné systémy lze ovládat lépe (naučit se rychleji)
- **recall** – rozpomenutí se, jak se taková situace řeší
- **efektivita** – zvládnout úkol rychle a snadno
- **minimum chyb** – v případě chyby korektně informovat a říci další postup
- **spokojenost uživatele** – přesvědčit uživatele, že úkol dokončil

2 Návrh UI – 3 fáze v cyklu

- **design** – návrh dle psychologie a toho, co uživatel potřebuje
- **evaluation** – prototyp a vyhodnocení s uživateli
- **implementace** – vytvoření prototypu a návrhu



- pro vyhodnocování je nutné zavést „přesnou“ metriku hodnocení tak, aby návrháři pochopili problém a mohli ho vylepšit
- záleží na cílové skupině a jejich schopnostech

3 Testování bez uživatelů

3.1 Kognitivní průchod

- Nejprve musíme znát všechny funkce a jak systém funguje. Poté se vrátíme na začátek a postupujeme jako nezkušený uživatel
- vhodné pro ověřování „klasického“ UI, levné a rychlé testování (ale málo vypovídající)

3.1.1 Vstup

- identifikace uživatelů (cílové skupiny) – jejich zkušenosti a znalosti
- stanovení úkolů a cílů, čeho chceme dosáhnout (Např. automat -> chceme koupit jízdenku)
- stanovení postupů, jak dané akce chceme dosáhnout (Např. vybrat destinaci -> vybrat slevu -> vhodit mince -> vzít jízdenku . . .)
- stanovení, co by se mohlo stát (Např. zjištění, že nemáme dostatek peněz -¬> storno -> vrácení peněz)

3.1.2 Výstup

- seznam nálezů a jejich hodnocení
- ve výsledném reportu popíšeme záporné výsledky, ale nenavrhujeme správné řešení (to je na vývojářích -> poté další testování)

Na začátku úkolu zodpovídáme nultou otázku
Q0: Ví uživatel, co chce udělat, čeho chce dosáhnout?

V každém kroku úkolu zodpovídáme otázky:

Q1: Ví uživatel, co má v tomto kroku udělat?
Q2: Propojí si uživatel nadpis, obrázek, vzhled s danou akcí?
Q3: Dostane uživatel rozumnou, jasnou zpětnou vazbu?

3.2 Heuristická evaluace

- identifikace problému a návrhu UI, obecná doporučení pro správný design
- vyhodnocováno 3-5 lidmi -> každý může daný problém vidět jinak – ujistí se, že neporušuje žádné heuristiky (=teorie řešení), jinak jsou sepsány do zprávy podle nálezů v porušení heuristiky
- je méně formální

3.2.1 Nielsenova heuristika (1994)

1. **Viditelnost stavu systému** - uživatel musí být informován informován o stavu (co se právě děje)
2. **Vztah mezi systémem a skutečným světem** – systém musí „mluvit“ jako uživatel v reálném světě
3. **Uživatelská kontrola a svoboda** – musí mít na výběr, možnost proces přerušit
4. **Konzistence a standardy** – dvě stejná označení musí dělat to samé
5. **Prevence chyb** – varování jestli to opravdu chce (Format c: Yes - No)
6. **Lépe připomenout než vzpomínat** – nezatěžovat uživatele zadáváním toho, co mu mohu nabídnout/napovědět
7. **Flexibilita a efektivita využití** – podpora zkušených uživatelů pro urychlení práce – zkratky
8. **Minimalistický design** – podávat jen informace, které jsou nutné
9. **Pomoc při diagnostice a zotavení z chyb** – chybové hlášky v lidské řeči (ne kus kódu), navrhnout řešení
10. **Pomoc a dokumentace** – v případě problému poskytnout dokumentaci a přesný postup

3.2.2 Postup při heurestické evaluaci

1. **Tutorial** – seznámení testu s aplikací (není vždy nutné)
2. **Evaluace** – alespoň 2 průchody – 1. Projít systém, 2. Zaměřit se na specifické problémy
    - hlášení chyb jednotlivě a od každého testu odděleně
    - jakou heuristiku porušuje a lehké doporučení (náznak) řešení, případný komentář
3. **Stanovení priorit** – označení nálezu podle závažnosti (alespoň 5úrovní)

- 1 test nalezne cca 35% problémů, 5 testů zhruba 75% problémů

3.3 Kognitivní průchod vs. Heuristická evaluace

3.3.1 Kognitivní průchod

- informace podle psychologie
- jeden tester/odborník
- více formální
- lepší pro hodně strukturované úkoly

3.3.2 Heuristická evaluace

- informace podle návrhových zvyků (praktice)
- více testerů
- méně formální
- vhodný pro jednodušší úkoly

3.4 KLM (keystroke level model)

- Pro měření časové náročnosti, metoda pro ohodnocení UI
- vstupem je popis úkolů
- založeno na ideálním průchodu, stanoveny časové údaje pro akce (i dle zkušeností)

3.4.1 KPHMR

K-stisk tlačítka
P-zaměření cíle myši
H-přehmat z klávesnice-myš
M-mentální příprava (myšlení)
R-reakce systému

Příklad - uložení souboru:
zaměření menu -> klik -> zaměření uložit -> klik...
total time = P+K+P+K.....+mentální příprava před provedením akce

4 Testování s uživateli

- sledování je v jejich prostředí (zvyklost a „pohodlné prostředí“) – těžké na organizaci, časově náročnější
- kontrolovaný test v usability labu (riziko stresu, atp.) – soustředění jen na test, možnost opakování
- problémem je vybrat správné lidi (nejlépe pomocí screeneru)
- vhodný počet participantů (často 5-10)
- před prvním testem nejlépe vše vyzkoušet sami na sobě (vyvarování se chyb a nedostatků v popisu – **Pilot test**)

4.1 Fáze testování:

- výběr na základě „**screeneru**“ – vhodná cílová skupina, zkušenosti – oslovení k testu
- „**pre-test**“ dotazník (veřejná a neveřejná část) + instrukce k testování – nechat čas na přečtení, ...
- samotný test (procházení úkolů)
- „**post-test**“ dotazník
  - odpovědi tak, aby se daly vyhodnotit (ano, ne, ohodnocení, ...)
  - otevřené otázky – možnost vyjádřit se

Psychologické a ergonomické aspekty používání uživatelských rozhraní. Speciální uživatelská rozhraní a jejich testování (A4B39TUR)

June 10, 2012

1 Psychologické a ergonomické aspekty UI

Zejména při testování v usability labu je nutné o uživatele pečovat – musí se cítit pohodlně, chovat se „normálně“, musíme ho ujistit, že pokud najde chybu/něco mu nepůjde, je to správné = proto testujeme.

1.1 Průběh testování:

1. Ice breaking – překonat jeho počáteční stres, představit se (2-5minut)
2. Briefing, focusing – říct, co chceme, netestujeme jeho ale program, aby „myslel nahlas“ (z důvodu záznamu), uzavření smlouvy, ochrana soukromí, ... (3-8minut)
3. Předtestové interview – dotazník, ... (2-5minut)
4. Sběr dat – testování, připomenutí myšlení nahlas (30-60minut)
5. Potestové interview – dotazník, často chtějí povídat co a proč udělali, ... (5-15minut)
6. Debriefing – rozloučení, poděkování, ujištění, že byl výborný, ... (2-5minut)

Verbální předávání informací zhruba 20%, neverbální pak 50% a více (jak se tváří, hýbe, atp.)

1.2 Typy uživatelů

1. Problémový participant
2. Dominantní – test vede moderátor, ne on!,obvykle za toto chování může úzkost/strach

3. Expert – stres z vlastní zkušenosti (neselhat), neměli by se účastnit testů
4. Insecure – nejistý, mluví málo a potichu (ujistit ho, že vše dělá správně)
5. Inadequate – nedostatečný participant

musí se cítit při testu pohodlně, rozhodně nesmí odejít se špatným pocitem/náladou, ...

2 Speciální uživatelská rozhraní a jejich testování

Testování – nutnost odhalit správnost převodu rozpoznání, subjektivní a objektivní pocity rychlosti, procento porozumnění

2.1 Hlasový vstup a výstup

- ASR (automated speech recognition) – převod WAV do TXT
- TTS (text to speech) – text -> slova -> hlášky -> hlasová řeč (v současnosti několik jazyků a řečníků/přízvuků)
- Speech to text – opačně (nejasnosti řeči – problémy s krátkou (rychlejší) výslovností, řešení kontextu, rozpoznání gramatiky)
- vhodné pro postižené lidi (slepé, bez rukou), řízení auta, ...

2.2 Rozhlasy (nádraží)

- limitováno kapacitou kanálu – množství informací x omezený čas na sdělení
- pouze pasivní příjem bez možnosti zopakování (nezáleží na uživateli)
- dilema mezi kompletností informace a užitečností (hlášení vlaku: řada minulých stanic a až poté cílové)

2.3 Telefonní služby

- technická podpora, reklamy, ... - většinou dočasné využívání
- poskytováno lidmi/automatem

2.4 Multimediální vstup

- kombinace několika vstupů a výstupů a jejich případné zahození v případě postižených
- umožňuje přístupnost systému co nejvíce lidem
- asistivní technologie: rozpoznání hlasu, sledování očí, čtečky obrazovek

- upravení použitelnosti pro staré lidi – větší text, písmena, minimalizace kliknutí a scrollování, méně barev
- poruchy zraku – (částečná) barvoslepost, šedoslepost
  - splynutí 2 normálně odlišných tras metra v jednu (problém například v londýně, NY, . . .)
- kognitivní znevýhodnění – problémy s pamětí, čtení, pozornost
- WEB – zákon o přístupnosti – pomocné/alternativní texty, vhodný kontrast, možnost nastavit přiblížení, atd.

12 Tvorba webových aplikací: Architektura webové aplikace, klientská část webové aplikace, W3C doporučení, webové skriptovací jazyky. Grafická a strukturální stránka prezentace.

12.1 Úvod

Současná architektura webu je založena na vztahu **klient - server**. Jediný způsob komunikace - tenký klient vysílá požadavky a server odpovídá (změna od HTML5).

Je možné dělit web na **statický**, **dynaický** a **webové aplikace** (s ajaxem).

Mnoho standardů je specifikováno v RFC (Request for Comments); např HTTP, HTML, SMTP.

12.2 Obsah

12.2.1 protokol - HTTP (Hypertext Transfer Protocol)

- bezestavový, textový
- metody *[1] - GET*, POST, HEAD* (jako get, ale poskytne pouze metadata), PUT, DELETE, trace*, options* (dotaz na poskytované metody), connect
- hlavička GETu obsahuje - UserAgent , accept-ln, accept-encoding, cookies
- HTTP 1.0 vs HTTP 1.1 - 1.1 obsahuje délku zprávy
- stavy : 2XX - Úspěch x 3XX - Přesměrování x 4XX - Chyba klienta x 5XX - Chyba serveru (200-ok , 403-forbidden, 404-notfound)

12.2.2 HTML (HyperText Markup Language)

- skupina SGML (Standard General Markup Language), XHTML skupina XML skupina SGML...XML vs SGML - párovost tagů v xml
- zlepšování přístupnosti - labely, dlouhé popisy, alternativní texty, thead. . .

[1] * - bezpečné metody = jen pro čtení

- zlepšení použitelnosti - accesskey
- od HTML 4 užívání CSS - oddělení obsahu a formy, kvůli robotům, vyhledávačům
- DOM = Document Object Model - (stromová) reprezentace dokumentu

12.2.3 CSS (Cascading Style Sheets)

- selectors (#-id, .-class)
- dědičnost (dá se porušit important)
- @media (styli pro tisk, handheld)
- je vhodné zadávat v relativních jednotkách (em, ex)

12.2.4 JS (JavaScript)

- skriptovací jazyk na straně klienta, má omezená práva, dá se využít k měnění DOMu, umí reagovat na události
- JSON (JavaScript Object Notation) - textový formát pro výměnu dat, skládá se z kolekcí páru název/hodnota a z tříděného seznamu
- AJAX (Asynchronous JavaScript and XML) - provádění requestů za účelem jiným než je načítání celé stránky
- objektovost a dědičnost - simulace, simulace pomocí prototypů

12.2.5 Doporučení W3C (World Wide Web Consortium)

World Wide Web Consortium (W3C) je mezinárodní konsorcium, jehož členové společně s veřejností vyvíjejí webové standardy. Takže jestli chcete něco dostat do standardů, tak je dobré mít **doporučení od W3C**.

Typy HTML 4.01 DTD (Document Type Definition)

- Strict includes all elements and attributes that have not been deprecated or do not appear in frameset documents
- Transitional = strict + deprecated elements and attributes (most of which concern visual presentation)
- Frameset = transitional + frames

12.3 Zajímavosti

Kdo by si přál konstanty v CSS?

12.3.1 novinky v HTML 5

- přibilo - canvas, audio, video
- Web Sockets - obousměrná komunikace
- Web Storage - key-value databáze, IndexedDB - robustní indexovaná databáze
- Web Workers - obdoba vlákna
- Native Drag & Drop

12.3.2 knihovna jQuery

mimo pěkné funkce umožňuje používat selectory v JS

12.4 Příloha

12.4.1 JS - prototipování + dědičnost

```
function vesmirny_objekt(){
this.soustava = "slunecni";
}
function planeta(pocet) {
this.pocet_mesicu = pocet
}
planeta.prototype = new vesmirny_objekt();
zeme = new planeta(3);
alert(zeme.pocet_mesicu);
alert(zeme.soustava);
```

13 Jazyk PHP, server-klient interakce. Šablony, MVC, frameworky, oddělení prezentační a aplikační logiky. Webové služby, AJAX.

13.1 Úvod

Současná architektura webu je založena na vztahu **klient - server**. Jediný způsob komunikace - tenký klient vysílá požadavky a server odpovídá (změna od HTML5).

Je možné dělit web na **statický**, **dynaický** a **webové aplikace** (s ajaxem).

Mnoho standardů je specifikováno v RFC (Request for Comments); např HTTP, HTML, SMTP.

13.2 Obsah

13.2.1 PHP (Personal Home Page)

- pole jsou asociativní, tedy ve skutečnosti se jedná o (hašovací) tabulky, které ukládají páry klíč -¿ hodnota.
- slabě dynamicky typovaný
- Imperativní (procedurální) a skriptovací jazyk, od PHP 5.0 podpora objektů
- C-like syntax
- Od PHP 5.3.0 - namespaces
- magické metody - __ , __get __set __isset __unset __clone (shallow copy)
- má modifikátory viditelnosti, final
- typi hinting u parametrů funkcí (kontrola typů), neuspokojení může způsobit fatal error
- Magické konstanty - __LINE__, __FILE__, __DIR__, __FUNCTION__, __CLASS__, __METHOD__, __NAMESPACE__
- velikou výhodou ve široká podpora a komunita

13.2.2 bezpečnost s PHP

Cross-side scripting narušení html stránky pomocí uživatelského vstupu (html tagy,JS). Dá se bojovat funkcí *htmlspecialchars*.

SQL injection ovlivňování SQL databáze nezamýšlený způsobem pomocí uživatelského vstupu. Dá se bojovat pomocí *prepareStatement* nebo *escapestring*. Někdo využívá na funkci *magicquotes*, která automaticky escapuje veškeré parametry GET a POST požadavků.

13.2.3 MVC

- výhody - dobrá dělba práce + v aplikaci se častěji mění view + lehce umožňuje předkompilovat části kódu, takže zrychlení (a odstranění komentářů)
- Page Controller(URL má vlastní controller) X Front Controller(jeden hlavní controller) X Composite View(stránka je složená z více view)
- super šablony Smarty
- celé MVC - Zend, Nette

13.2.4 frameworks

- Doctrine - ORM
- ZEND, NETTE - řeší lokalizaci, MVC, strukturu projektu, formuláře, ajax, ...

13.2.5 Webové služby

Toto téma je již zpracováno.

13.2.6 Ajax

Asynchronous JavaScript and XML - provádění requestů za účelem jiným než je načítání celé stránky (úprava DOMu na základě nových dat ze serveru, asynchroní odesílání dat).