Úvod do datového modelování

- **Základní prvky datového modelu (entity, vazby, atributy)**
- **Proč datový model**
  - Dokumentovatelnost
  - Kontrolní mechanismy
  - Verzování a změny
  - Přenositelnost mezi DB, vazby na DB
  - Údržba
- **Základní principy datového modelu**
- **Databáze jako databázový systém (celek)**
- **Další prvky datového modelu (uložené procedury a triggery)**
- **Nástroje pro tvorbu datových modelů**
- **Praktická tvorba datových modelů**

Předtím, než začneme vytvářet nějaký datový model, měli bychom si uvědomit, jakou část reality chceme zobrazovat. Předpokládáme, že v konečné fázi vývoje modelu budeme chtít se získanými daty určitým způsobem manipulovat – např. uchovávat je v databázi. A právě k popisu procesu vývoje tohoto modelu nám může do určité míry pomoci tvz. *princip tří architektur* (P3A). V souvislosti s architekturou P3A se také často hovoří o třech úrovních návrhu informačního systému.

Architektura P3A definuje způsob použití *abstrakce*, což znamená, že nám umožňuje rozčlenit právě zkoumanou problematiku návrhu datové základny (DZ) na mentálně zvládnutelné části. Jak již název vypovídá, skládá se P3A ze třech vrstev (úrovní) - *konceptuální*, *technologické* (logické) a *implementační* (fyzické).

Konceptuální úroveň
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.

Technologická úroveň
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 (o co konkrétně jde si vysvětlíme později). Technologický model stále nesmí být zatížen implementačními specifiky řešení. Technologický návrh určuje *jak* je obsah systémů v dané technologii realizován.

Implementační úroveň
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). Implementační návrh určuje *čím* je technologické řešení realizováno.

Nyní již tedy víme, z čeho se P3A skládá a příště se blíže podíváme na tzv. entitně-relační diagram.

Cílem datového modelování je navrhnout kvalitní datovou strukturu pro konkrétní aplikaci a databázový systém, který bude tato aplikace využívat k uložení dat.

Při datovém modelování obvykle vytváříme nejprve **konceptuální datový model**. Konceptuální datový model představuje určité zobecnění oproti konkrétní implementaci datové struktury v relační databázi. Zobecněním získáme nezávislost modelu na konkrétním databázovém systému, ale zároveň jsme schopni tento model kdykoliv převést do konkrétního implementačního prostředí.

Pokud bychom od začátku analýzy vytvářeli datovou strukturu na míru například databázovému systému MySQL a později zjistili, že zákazník hodlá přejít na Oracle nebo dokonce na nějakou XML databázi, museli bychom pravděpodobně celý návrh opakovat.

Vhodnější je tedy vytvořit nejprve obecnější konceptuální datový model a teprve na jeho základě ve fázi implementace navrhnout datovou strukturu pro konkrétní databázový systém. Pokud používáme nějaký CASE nástroj, máme téměř vždy možnost nechat si z příslušného modelu databázovou strukturu automaticky vygenerovat (například ve formě SQL skriptů). Problematiku datového modelování vzhledem k rozsahu a obtížnosti tématiky pouze ukážeme na příkladu.

Příklad datového modelu

Datový model slouží pro návrh datové struktury. Jedná se o konceptuální model, který je velice podobný klasickým ER diagramům používaným pro návrh struktury relačních databází nebo UML Class diagramům používaných v objektové analýze a návrhu.
Model je tvořen sadou entit, které jsou mezi sebou provázány pomocí vazeb s příslušnou kardinalitou (násobností).

Entity

Entita reprezentuje určitou skupinu objektů reálného světa, které se označují jako **instance entity**. Všechny instance dané entity se nazývají **populací** a vyznačují se stejnou vnitřní datovou strukturou, která se vyjadřuje množinou **atributů**. Pouhá shoda atributů ale samozřejmě nestačí, všechny instance dané entity musí mít i logickou souvislost. Typickou entitou může být například entita "zákazník" a instancemi této entity mohou být například "Jan Novák", "Petra Čechová" a další. Entitou samozřejmě nemusí být jen nějaké reálné fyzické objekty (zboží, zákazník), ale může se jednat i o objekty abstraktní (kategorie zboží, objednávka a podobně).

Každá entita je popsána svým názvem a sadou atributů. Například entita zboží bude mít pravděpodobně atributy (kód, název, popis, cena, záruka a další). Každý z těchto atributů má samozřejmě přiřazen i **datový typ**.

Tyto datové typy jsou navrženy tak, aby nebyly přímo závislé na konkrétním databázovém systému. Jde o základní datové typy, se kterými se pravděpodobně setkáme ve většině databázových systémů. Při generování konkrétní databázové struktury z navrženého modelu jsou tyto datové typy nahrazeny příslušnými ekvivalenty konkrétního databázového systému.

Pro datový model se používá grafické zobrazení.Entitu zobrazujeme pomocí obdélníku, ve kterém jsou vepsány název a pod čarou atributy (viz obrázek č. 1)

Obrázek č. 1 – příklad entity "zbozi"

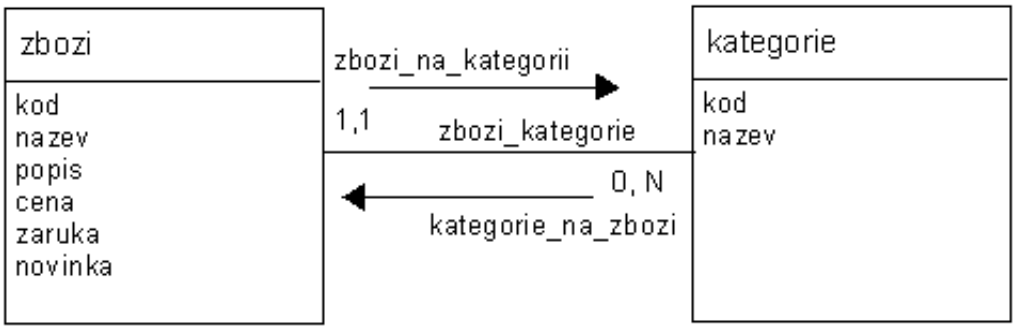
Vazby mezi entitami, kardinalita

Vazba mezi entitami představuje **logický vztah mezi entitami**. V datovém modelu jsou povoleny pouze vazby mezi dvěma entitami (binární vazby). Vychází se z možnosti realizovat jakoukoliv N-ární vazbu (mezi N entitami) pomocí N vazeb binárních a jedné centrální (vazební) entity.

Stejně jako entita má své instance, také binární vazba má své instance. Pokud je binární vazba definována jako logický vztah mezi dvěma entitami, potom je instance binární vazby vztah mezi dvěma instancemi dvou entit. Například vazba mezi entitou zboží a kategorie, může mít instanci "stůl od firmy xy patří do kategorie kuchyňské stoly".

Na vazbu můžeme pohlížet jako na dvě vazby v opačných směrech. V tomto smyslu se hovoří o takzvaných **rolích**, které představují pohled na danou vazbu ve směru od jedné entity ke druhé. Například na vazbu mezi kategorií zboží a zbožím můžeme pohlížet tak, že se ptáme, do jaké kategorie daná položka zboží patří, nebo tak, že se ptáme, jaké zboží patří do příslušné kategorie.

Ke každé z rolí přiřazujeme takzvanou **kardinalitu**. Kardinalita představuje omezení v počtu instancí druhé entity, které mají vztah s jakoukoliv instancí první entity. Definuje se vždy **maximální a minimální kardinalita**. Minimální kardinalita může nabývat hodnot 0 nebo 1 a maximální kardinalita 1 nebo N. K osvětlení problematiky vazeb a kardinality by mohl pomoci obrázek č. 2.



Obrázek č. 2 – příklad binární vazby ¨

Na obrázku č. 2 můžeme vidět vazbu pojmenovanou "zbozi_kategorie", která vyjadřuje vztah mezi zbožím a kategorií. Vazbu charakterizují dvě role podle toho, kterým směrem ji čteme. K rolím jsou přiřazeny kardinality. Pokud vazbu čteme směrem od zboží ke kategorii (role: zbozi na kategorii), zajímá nás, v kolika kategoriích může být zboží uloženo. Kardinalita "1,1" znamená, že zboží může být uloženo v jedné a právě jedné kategorii. Pokud vztah čteme z druhé strany (role: kategorie_na_zbozi), zajímá nás, kolik druhů zboží může být v jedné kategorii. Kardinalita "0,N" znamená, že kategorie může být prázdná nebo obsahuje N druhů zboží (teoreticky nekonečno).

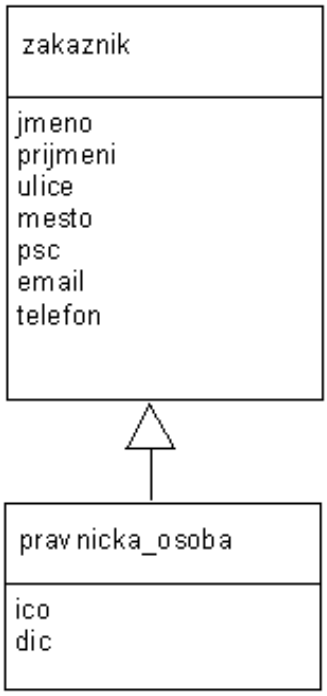
Při generování struktury databáze z modelu nějaké relační databáze, odpovídá každé entitě jedna relační tabulka. V datovém modelu se *nedefinují explicitně* atributy, které v relačních databázích označujeme jako **cizí klíče**.

Cizí klíč totiž není z hlediska konceptuálního modelu logickou součástí entity zboží, ale je definován pomocí vazby, jejíž implementace se může v různých databázových systémech lišit. Pokud se ale rozhodneme, že v naší aplikaci použijeme relační databázi, budou z modelu při generování datové struktury automaticky vygenerovány i atributy cizích klíčů.

Při vyskytnutí vztahu "1,N : 1,N" (jedna instance první entity má vztah s více instancemi druhé entity a naopak) se automaticky generuje vazební tabulka, která se bude skládat ze dvou cizích klíčů.

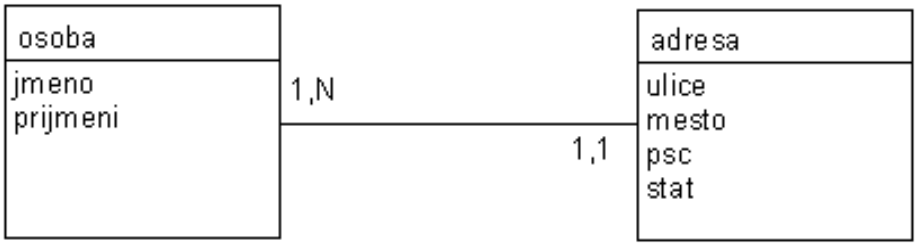
Generalizace mezi entitami

Datový model umožňuje sestavovat **hierarchie entit**. Znamená to, že nadřazená entita (super-entity) může mít své podřazené entity (sub-entities). Podřazená entita představuje speciální případ nadřazené entity, dědí od ní všechny atributy a může k nim přidat své specifické atributy. Hierarchie může mít více úrovní. Každá podřízená entita nesmí mít víc jak jednu nadřízenou entitu. Jednoduchý příklad generalizace viz obrázek č. 3:



Strukturované atributy

V praxi to znamená případ, kdy má entita atribut nebo kolekci atributů, které nejsou tvořeny jen atomickou hodnotou, ale mají také svou vnitřní strukturu.
Princip spočívá v tom, že se k hlavní entitě vytvoří entita, která nemůže existovat, aniž by existovala hlavní entita. Tato entita se propojí s hlavní entitou pomocí vazby s potřebnou kardinalitou.
Klasickým příkladem může být situace, kdy k jedné osobě sledujeme více kontaktních adres. V modelu vytvoříme následující strukturu:



Příklad návrhu datového modelu

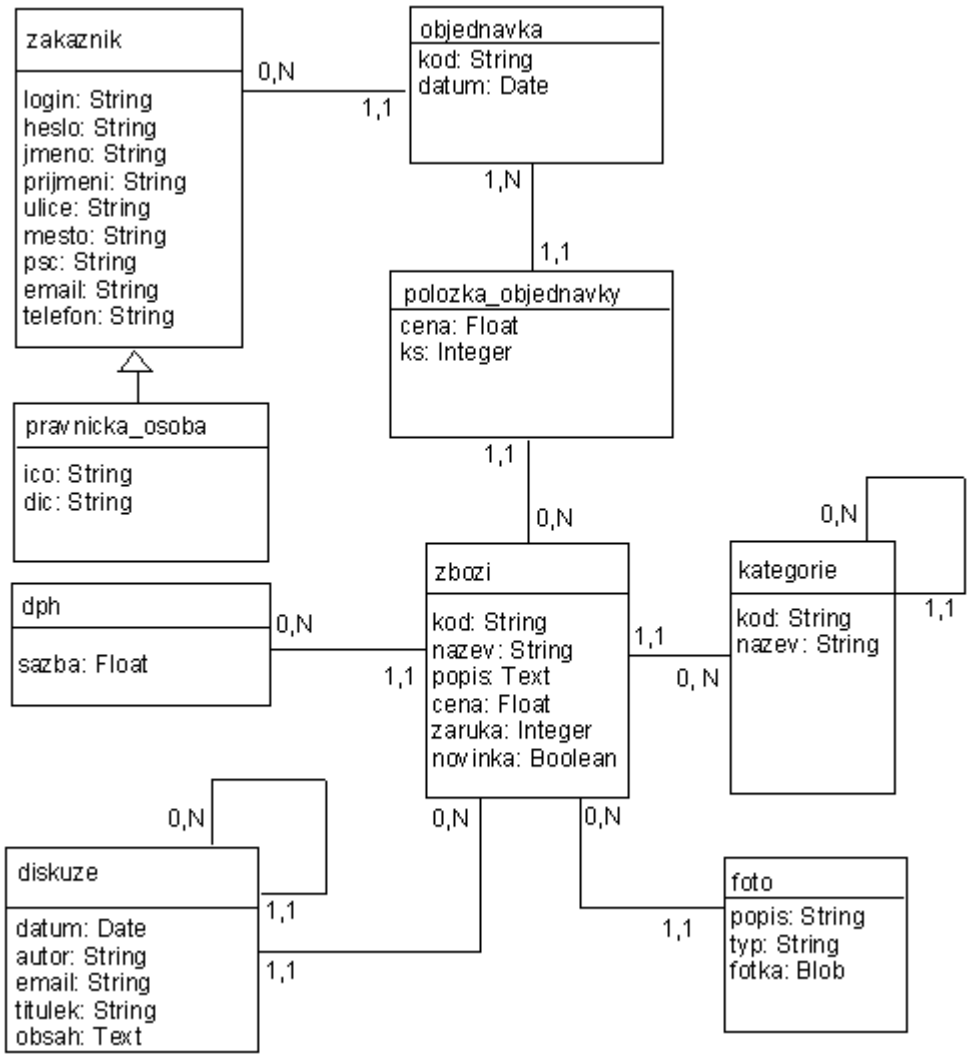
Jako shrnutí předešlého textu zde bude uveden příklad návrhu datové struktury elektronického obchodu. Tento příklad bude velice jednoduchý a neúplný a měl by sloužit především k objasnění toho, jak se vytváří datový model a ne jako obecný návod pro tvorbu datové struktury elektronického obchodu.

Jaké entity se budou vyskytovat v našem systému, musíme zjistit ve **fázi specifikace a sběru**

funkčních požadavků na aplikaci. Tato fáze je nedílnou a velice důležitou součástí všech metodologií analýzy a návrhu SW.

Na základě rozhovorů, konzultací se zákazníkem a vlastních zkušeností sestavíme seznam všech možných entit, které v aplikaci předpokládáme, a postupem času tento seznam upravujeme (některé entity přidáváme, některé rušíme). K jednotlivým entitám postupně přidáváme identifikované atributy a vazby, až vznikne výsledný datový model.

Budeme předpokládat, že prodáváme zboží, které je organizováno do kategorií. Kategorie jsou uspořádány v hierarchii. Zboží si objednává jen ten zákazník, který se zaregistroval. Ke každému typu zboží se může diskutovat v jednoduchém diskusním fóru. U zboží sledujeme kód, název, cenu, sazbu DPH, popis a libovolné množství obrázků. Konceptuální datový model by mohl vypadat následovně:



Obrázek č. 5 – výsek datové struktury elektronického obchodu

Zákazník může vytvořit 0 až N objednávek. Každá objednávka se skládá z jedné až N položek,

které jsou provázány s určitým druhem zboží a udávají, kolik kusů daného zboží se objednává a za jakou cenu (ceny se v čase mění). Speciální typ zákazníka představuje právnická osoba, u které navíc sledujeme IČO a DIČ. Zboží patří vždy do jedné kategorie. Kategorie může a nemusí mít podkategorie. Byla zde použita vazba entity na sebe samu. (Tímto způsobem se obvykle modeluje hierarchie.) Ke každému zboží dále ukládáme diskusní příspěvky, které jsou opět organizovány hierarchicky. V databázi budeme mít k dispozici také číselník DPH. Ke každému zboží můžeme ukládat libovolné množství fotografií. Datová struktura není úplná a použity byly pouze základní atributy. V reálu bychom datovou strukturu upřesnili podle požadavků zákazníka. Z modelu můžeme pomocí CASE nástroje vygenerovat datovou strukturu podle potřeby pro konkrétní databázový systém. Pro libovolnou relační databázi by byla ke každé entitě automaticky vygenerována relační tabulka s příslušnými atributy. Tabulky by byly dále doplněny o primární a cizí klíče podle stanovené jmenné konvence.

Indexy

Databázové indexy slouží ke zrychlení přístupu k datům a měly by se používat u všech sloupců, podle kterých se vyhledává, třídí nebo podle kterých se spojují tabulky.

Při ukládání dat do tabulek nejsou záznamy obvykle nijak tříděny a ukládají se většinou za sebe tak, jak byly postupně vloženy. V momentě, kdy chceme data z tabulky později vybrat podle nějakého kritéria, je nutné projít všechny záznamy a vybrat z nich ty, které kritériu vyhovují. K tomu, aby při výběru několika záznamů nebylo potřeba procházet všechny ostatní, slouží právě indexy, ve kterých jsou data organizována tak, aby bylo možné rychle vybrat pouze relevantní záznamy.

Indexy se vytvářejí nad jedním nebo několika sloupci tabulky, každá tabulka může mít indexů několik. Index definovaný nad sloupcem tabulky umožňuje rychlý přístup k záznamům podle hodnot tohoto sloupce.

Organizace dat v indexu umožňuje nejen přímé vybrání záznamů s určitou hodnotou, ale samozřejmě také záznamů v intervalu hodnot. Kromě toho jsou prvky v indexu provázané podle svého pořadí při řazení (ať už se jedná o číselné, nebo řetězcové sloupce), takže indexy umožňují také rychlé seřazení tabulky podle sloupců, nad kterými je index definován. Tím pádem umožňují i rychlé vybrání minima a maxima. Informaci o počtu hodnot a počtu různých hodnot (SQL funkce COUNT a COUNT DISTINCT) databáze obvykle uchovávají nezávisle na indexu ve statistikách tabulky, které používají například také při hledání strategie pro vyhodnocování složitějších dotazů.

Indexy nad řetězcovými sloupci umožňují také rychlejší vyhledávání pomocí operátoru LIKE, avšak pouze v případě, kdy je znám začátek hledaného výrazu – tedy např. X LIKE 'text%' využití indexu dovoluje, X LIKE '%text%' ne.

Použití indexů se často zanedbává a faktem je, že u malých tabulek obsahujících řádově desítky záznamů je jejich význam zanedbatelný. U větších tabulek indexy naopak výkon ovlivňují zásadně. Vzhledem k tomu, že správa indexu stojí určitou režii při každém vkládání záznamu nebo jeho mazání, měli bychom se vytváření indexů vyhnout u tabulek, do kterých se převážně vkládá a jen výjimečně se z nich čte, což jsou např. logy.

Kromě běžných indexů lze definovat také unikátní indexy, které do tabulky nedovolí vložit více záznamů se stejnou hodnotou sloupců, nad kterými je index definován (výjimku tvoří hodnoty NULL). Tato informace může databázovému serveru sloužit také k efektivnějšímu uspořádání dat. Speciálním typem indexu je primární klíč označující sloupce, které jednoznačně identifikují libovolný záznam v tabulce. Definování primárního klíče by mělo být samozřejmostí.

Po vytvoření indexů se o ně již dále nemusíme starat, databázový server sám zajišťuje jejich automatickou aktualizaci a sám rozhoduje o tom, jaké indexy využije při získávání dat.

Uložené procedury
Dalším prvkem jsou uložené procedury. Jedná se o posloupnosti (mnohdy předkompilovaných) příkazů SQL a příkazů pro řízení běhu. Jde o významné vylepšení standardních možností jazyka SQL dovolující nám používat parametry, deklarovat proměnné, větvit běh příkazů a taktéž možnost navracení hodnoty.

Jednoduché přiblížení činnosti uložené procedury spočívá v představě sestaveného podprogramu nebo funkce, která bude následně uložena jako nový objekt databáze. Uložené procedury běží na SQL serveru, a nikoli na klientu, který zadal dotaz. Uložené procedury zpravidla obsahují posloupnost příkazů jazyka SQL a příkazy pro řízení běhu, které zpracovávají tabulky z databáze.

Speciálním druhem uložené procedury je tzv. *trigger* (spouštěč), který je automaticky vyvoláván v okamžiku, když dojde k určité nastavené změně dat v tabulce. Může být navázán třeba na přidání nových dat, jejich aktualizaci či výmaz. Triggery se využívají pro kaskádové zřetězení změn v souvisejících tabulkách. Jinými slovy se jedná o systém automatického vyvolávání činností, které vedou k zachování integrity databáze. Za nevýhodu *aktivačních procedur* lze považovat fakt, že při každé operaci musí server zjišťovat, zda se na ni neváže nějaká procedura. Tím ztrácí čas a tolik žádaný výkon. Jako téměř vždy je tedy na pořadu otázka vhodného kompromisu.

Za klady uložených procedur lze považovat:

- Získáváme na výkonu v porovnání s posloupností standardních příkazů SQL, neboť příkazy jsou v uložených procedurách obvykle předkompilovány. Při prvním provádění uložené procedury je vytvořen prováděcí plán činnosti. Plán je po vytvoření uložen do vyrovnávací paměti a následné opakování téže procedury je pak mnohem rychlejší než provádění obdobných příkazů SQL.
- Pomocí uložených procedur lze vykonat složitější operace, než bychom vykonali přímo pomocí SQL.
- Významným kladem je nepřerušitelnost provádění procedury. Po jejím spuštění je procedura provedena sekvenčně jako celek, i když má třeba vnitřní strukturu složitou. Správná konstrukce procedur potom zajišťuje minimum kolizí s integritními omezeními.
- Možnost nastavování a předávání parametrů umožňuje propojení více uložených procedur, a tím vlastně vytvářet jakési dávkové programy.

- Umožňují vytvořit další vrstvu zabezpečení databáze. Existuje totiž možnost zrušit uživatelům veškerý přístup k tabulkám a pohledům (zamezíme uživatelům přímý přístup) a vytvoříme uložené procedury, které mohou pracovat s těmito objekty databáze. Uživatelům pak stačí přidělovat práva pro spuštění (execute) procedur.

- Velká část aplikační logiky může být přesunuta na databázový server – díky uloženým procedurám s vlastními proměnnými, podmínkami, cykly, popř. vyvoláváním vestavěných funkcí databázového serveru.

Za nevýhodu uložených procedur lze považovat zejména nutnost správy procedur – parametry, vlastnosti, vhodnost použití…

Vytvoření procedury
Stejně jako u běžných programovacích jazyků pod pojmem procedura rozumíme blok kódu se vstupními parametry. Vstupní hodnoty se pak v těle procedury zpracují a výsledkem činnosti může být nějaká úprava údajů z databázové tabulky, popřípadě lze odevzdat výsledek výpočtu či vytvořený řetězec. Uloženou proceduru vytváříme pomocí příkazu *create procedure*.

```
create procedure nazev_procedury (seznam_parametrů) as
begin
  tělo procedury
end
```

Vytvoříme nyní proceduru bez parametrů:

```
create procedure cisteni as
begin
  delete from tabulka_1;
  delete from tabulka_2;
  delete from tabulka_3;
end
```

Pokud se například tabulka_1 odkazuje do tabulky_2 a tabulky_3, tak můžeme považovat za výhodu už to, že bude při mazání obsahu tabulek automaticky pamatováno na správné pořadí a s referenční integritou nemusí být problémy. Také si nemusíme pamatovat názvy původních tabulek a navíc lze jednoduše odstupňovat práva k tomuto úkonu "čištění". Přístupové právo k proceduře cisteni by měl nejspíše jen správce databáze či administrátor aplikace.

Vytvořenou proceduru cisteni lze poté spouštět pomocí příkazu execute:

```
  execute cisteni;
```

Vstupní parametry každé procedury jsou dány svými názvy a příslušnými datovými typy. Seznam parametrů je při vytváření procedury uveden v kulatých závorkách za názvem procedury. Pro odlišení názvů sloupců tabulek a názvů lokálních proměnných se používá dvojtečková konvence. Je-li tedy uvedena před názvem dvojtečka, jde o název lokální proměnné. Pokud bychom tedy chtěli například smazat určitou osobu podle předeného rodného čísla, vypadalo by to například takto:

```
create procedure rc_delete(rc_prom varchar(10)) as
begin
  delete
    from Zamestnanec
      where rc = :rc_prom;
end
```

Proceduru s uvedeným parametrem – rodné číslo mazané osoby – bychom pak spustili pomocí příkazu:

```
execute procedure rc_delete "7258304460";
```

Pokud by měla procedura více parametrů, oddělovali bychom je v definici pomocí čárky a podobně při vyvolávání procedury se pak musí uvést potřebné parametry oddělené čárkou, samozřejmě také ve správném pořadí. Další možností vylepšení je pro některé případy procedur uvedení návratové hodnoty s výsledkem operace (někdy hovoříme o uložených funkcích). Potřebujeme-li tedy, aby procedura navracela hodnoty, uvedeme seznam vracených proměnných v závorce za klíčovým slovem returns. Například bychom chtěli zavést proceduru pro výpočet následujícího členu Fibonacciovy posloupnosti s uvedením dvou potřebných předcházejících členů (parametrů):

```
create procedure Fib_clen
  (prom_1 integer, prom_2 integer)
 returns (clen integer) as
begin
  :clen = :prom_1 + :prom_2;
end
```

Následuje vyvolání procedury pro vlastní výpočet. Musí být však dopředu deklarována výstupní proměnná – třeba prom. Proceduru vyvoláme příkazem s klíčovým slovem returning values před výstupním parametrem:

```
execute procedure Fib_clen 3, 5 returning values :prom;
```

Nástroje CASE

Na automatizaci vývoje softwarových aplikací pomocí výpočetní techniky měl vždy vliv především vznik metodik vývoje software a také vznik diagramů pro znázornění systému z různých úhlů pohledu. A jakou roli hrály či hrají CASE nástroje?

Nástroje, které podporují vývoj softwarových aplikací se nazývají CASE – *Computer Aided Software Engineering*. CASE nástrojů existuje celá řada, a to jak pro strukturované, tak i pro objektově orientované metody vývoje.

Některé CASE nástroje jsou integrovány do moderních prostředí pro vývoj software (Borland, Oracle). Přestože vypadá, že tvorba diagramů v těchto nástrojích je jednoduchá, vyžaduje vysokou znalost a profesionálnost tvůrce modelů a těch, kdo nástroj používají. Druhým předpokladem úspěchu je vhodnost použitých metod, na kterých je CASE založen.

Stejně jako například řešení CRM (Customer Relationship Management) nepředstavuje jen program na evidenci kontaktů či platební morálky zákazníků, ale vyžaduje definování a zavedení strategie vztahu k zákazníkům, tak i CASE nejsou jen nástrojem na kreslení diagramů, ale musí podporovat metodiky využívané ve firmě.

Jednou z nejdůležitějších vlastností CASE nástrojů je zajištění souvislostí, které člověk neumí mentálně pojmout. Samotné použití CASE nám ale nezaručí bezchybný a rychlý vývoj aplikace nebo systému. Stále je nutné mít na paměti, že se jedná pouze o jednu částečku zapadající do mozaiky vývoje, kde nezbytné je také řízení projektu, zavedení a plnění metodiky, využití

odborníků na jednotlivé oblasti (analytik, návrhář, programátor, tester), použití a vyhodnocení metrik apod.

Druhy CASE systémů
Jak již bylo uvedeno, existuje dnes mnoho CASE nástrojů. Je to dáno nejen podporovanou metodikou, ale také tím, v jaké fázi vývoje je nástroj používán. CASE nástroje se využívají ve fázích specifikace požadavků, analýzy, návrhu, kódování a údržby.

Nástroje použité v různých etapách se liší a je obvyklé, že pokrývají jen určité činnosti. Stírají se také hranice mezi CASE a integrovanými vývojovými nástroji (viz například CASE a současně také integrovaný vývojový nástroj QI Builder, který je součástí informačního systému QI od společnosti DC Concept a celý informační systém je v něm vytvořen).

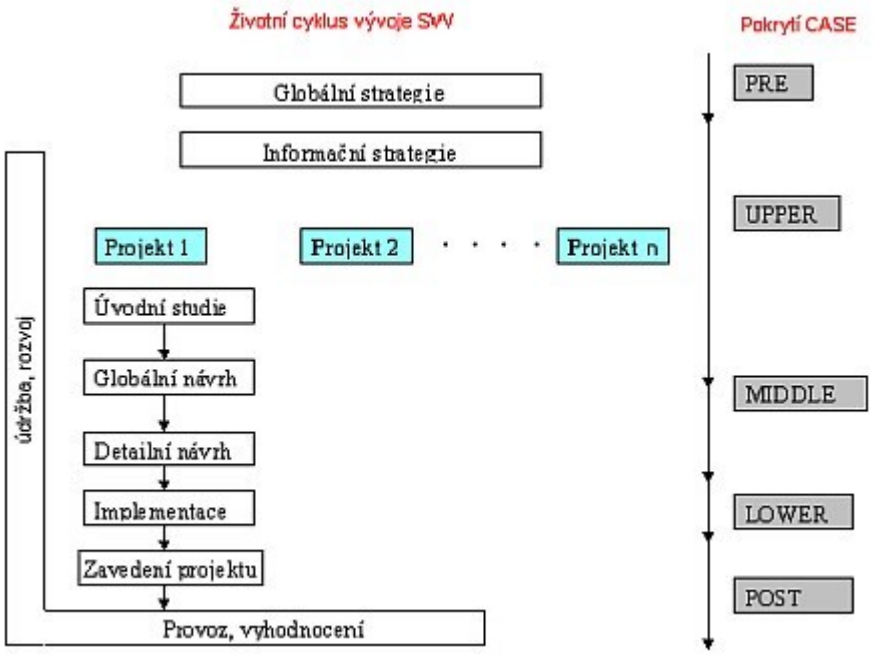
Podle životního cyklu vývoje software lze CASE nástroje rozdělit do následujících skupin:

- *Pre CASE* – podporují tvorbu globální strategie.
- *Upper CASE* – podporují plánování, specifikaci požadavků, modelování organizace podniku a globální analýzu IS. Hlavním úkolem nástroje je analýza organizace, zachycení procesů v organizaci, definice klíčových informačních toků a dokumentace zjištěných požadavků.

  Z těchto údajů je jasné použití při specifikaci cílů, počáteční specifikaci požadavků a řízení projektů. Použité nástroje mohou být DFD (Data Flow Diagram) a ERD (Entity Relationship Diagram) bez podrobných atributů, prostředky pro řízení projektů a sledování ekonomických ukazatelů, popis základních vlastností systému prostředky OO modelování.
- *Middle CASE* – podporují podrobnou specifikaci požadavků a vlastní návrh systému. Tato třída CASE nástrojů je nejúspěšnější. Používají se pro podrobnou specifikaci požadavků, návrh systému, dokumentaci a vizualizaci systému. Použité metody a nástroje jsou DFD včetně podrobného popisu procesů, datových úložišť, podrobné ERD, pro OOAN – diagramy tříd, instancí, přechodové diagramy apod.

  Dále CASE nástroje této kategorie obsahují systém správy dokumentů a konfigurace, systém pro vyhodnocování metrik, vývoj prototypů, návrh rozhraní. Mohou obsahovat také generátory obrazovkových formulářů a tiskových sestav a také generátory (kostry) definic dat. Tento druh CASE je jádrem komerčně dodávaných CASE systémů.
- *Lower CASE* – obsahují nástroje pro podporu kódování, testování, údržby a reverzního inženýrství. Integrovány jsou různé nástroje, jako jsou generátory kódu (mohou generovat jen kostru nebo až 75 procent výsledného kódu, kde programátor doplňuje většinou jen detaily). Dále pak jde o prostředky pro reverse engineering (rekonstrukce dokumentace a modelů z existujícího SW), prostředky pro sledování a vyhodnocení metrik, prostředky plánování a zjištění kvality SW (sběr informací o průběhu testování, vyhodnocení výsledků testů, řízení testování), pro správu konfigurace, prostředky sledování a vyhodnocování práce systému. Funkce CASE nástrojů této kategorie se často překrývají s funkcemi obecných vývojových prostředí.

- *Post CASE* – podporuje organizační činnosti (zavedení, údržbu a rozvoj IS).



Pokrytí fází životního cyklu druhy CASE

Komponenty CASE systémů
Z toho, jaké jsou obecné funkce a vlastnosti CASE nástrojů vyplývá také to, z jakých komponent se tyto systémy skládají. Mezi důležité funkce a vlastnosti CASE patří:

- *Konzistentní grafické ovládací prostředí* (podle zásad tvorby GUI) – jednotný vzhled obrazovek, popisků, tlačítek, jednotné ovládání, použití symbolických ikon apod.
- *Centrální databáze* pro uchování informací o všech objektech IS (tímto způsobem se zaručí, že informace je použitelná v libovolném dalším kroku projektování)
- *Prostředky verifikace konzistentnosti dat a podpora normalizace dat*
- *Textový editor* pro popis jednotlivých objektů – pro účely technické a uživatelské dokumentace systému, možnost jejího přímého generování ze systému
- *Možnost rychlého návrhu uživatelských obrazovek* včetně simulace vstupů a výstupů (je vyžadováno pro prototyping)
- *generátor zdrojových programů* (pro případy častého znovupoužití daného kódu)
- *export / import dat* – pro práci s modely a dokumentací, které byly vytvořeny v jiných programech nebo jsou v jiných programech dále využívány a zpracovávány

První komponentou je grafické rozhraní. Skládá se z obrazových primitiv, která jsou předdefinována (kružnice, čtverce, přímky, křivky, šipky) s možností jejich uchování. Grafické rozhraní je v podstatě elektronická tabule, na kterou analytik konstruuje grafy a diagramy.

Toto rozhraní umožňuje vytváření grafů z primitiv, přiřazování názvů grafickým objektům, sdružování jmen s objekty, kontrolu pozic jmen v diagramu. Umožňuje také editaci

vytvořených grafů (schopnost mazat, přepisovat, modifikovat grafické objekty). Úsilí jaké je třeba k vytvoření diagramu může být měřeno počtem stisků klávesy nebo kliknutí myši.

Jednou z vlastností grafického rozhraní je také to, že systém automaticky vymaže odkazy na mazaný objekt – aby byla zachována konzistence modelu. Grafické rozhraní by mělo také umožňovat přemístění jednoho nebo více grafických objektů. Z dalších vlastností uveďme obnovu objektů do jejich předchozího stavu (po výmazu, i více kroků zpět) – funkce "UNDO" nebo změnu měřítka objektů.

Pokud jsme zmínili grafické rozhraní, je jasné, že musí existovat nějaké vstupy, kterými bude možno s danými grafickými prvky či s grafy pracovat, manipulovat a také pomocí nich celý systém ovládat, tedy vstupní rozhraní. Mezi vstupní zařízení řadíme jak standardní vstupní periferie počítače (klávesnice, myš), tak také další technická zařízení jako scanner, světelné pero a speciální hardware.

Vstupní rozhraní zajišťuje přenos vstupní informace do systému a grafickou interpretaci vstupních operací. Jakýkoliv pohyb myši, stisk klávesy, pohyb světelným perem musí být promítnut do systému a následně na obrazovku počítače (ve spolupráci s výstupním rozhraním).

Výstupní rozhraní se stará o provedení výstupů ze systému. Mezi výstupní technická zařízení může patřit monitor, tiskárna nebo plotter. Výstupní rozhraní zahrnuje také definici výstupního formátu tisků – formát papíru, kvalita tisku, fonty a velikost písma, okraje, tituly, apod.

Slovník
Důležitou komponentou CASE systémů je slovník, jehož přítomnost v podstatě klasifikuje systém do rodiny CASE nástrojů. V některých nástrojích je slovník automatický (jakmile vytvoří uživatel objekt diagramu, je ve slovníku automaticky o tomto objektu vytvořen záznam). Velikost slovníku definuje maximální počet procesů, toků, datových objektů. Existují různé metody navigace (přístup k položkám databáze CASE) ve slovníku, přístup pomocí diagramu, přímý přístup do slovníku apod.

Slovník funguje většinou také jako textový procesor pro účely dokumentace. Ve slovníku je obsažena definice datových struktur (použitých datových entit), definice vztahů v hierarchii procesů (rodič-potomek) a v neposlední řadě také relace mezi daty a procesy. Modelování systémů může být doprovázeno možností vytvářet vstupně/výstupní obrazovky. Vývojová prostředí čtvrté generace tyto prostředky obsahují, u CASE nástrojů to však obvyklé není. Každý CASE, který generování obrazovek podporuje, může mít jinou úroveň použití grafiky při definici obrazovek, jiný stupeň sjednocení mezi použitou grafikou a textem na obrazovce, jinou úroveň integrace slovníku dat s výstupem apod.

Důležitou vlastností CASE nástrojů je možnost kontroly kvality modelu. Systém kontroluje tvořené modely a diagramy podle pravidel tvorby a podle definovaných logických souvislostí. Dále kontroluje izolované a nedefinované jednotky dat, procesy a moduly bez specifikace, uložení dat jako externí zdroje nebo self vazby entit (vazby sama na sebe).

K těmto kontrolám je použit slovník, kde jsou uloženy vazby a významy jednotlivých entit. Na základě těchto kontrol bývají generovány zprávy, které uživateli oznamují možnost nebo nemožnost provedení daného kroku. Samozřejmostí by měla být podpora generování seznamů datových položek a jejich atributů.

Na složitých projektech, při kterých jsou využívány CASE, pracuje obvykle mnoho odborníků. Proto je třeba, aby nástroje CASE podporovaly práci v síťovém prostředí, výměnu dokumentů, správu verzí modelů, různé formy komunikace a také použití centrálního

slovníku. Je také vhodné mít k dispozici podporu připojení na externí databáze (např. pro generování schémat, skriptů nebo struktur databází nebo pro uložení slovníku).

Omyly jsou věčné...
Neměli bychom zapomínat, že CASE není samo o sobě metodologií, ale používá již zavedené metodologie. Nástroj CASE není ani náhradou programovacích jazyků (může generovat části kódu, ale programovací jazyky nenahrazuje). Užívání CASE automaticky nezlepší práci vedoucích pracovníků podniku nebo specialistů pro IT, CASE je nástroj, který může zlepšit produktivitu práce.

CASE odstraňuje potřebu disciplíny a přísného vývoje aplikací IT, systémy CASE často selhávají právě díky nedisciplinovanosti uživatelů. Od CASE se často očekává jako výstup tvorba aplikačního programového vybavení. Hlavní přínos přitom je v úplném poznání fungování podniku a vytváření úplných podkladů právě pro programování aplikací.

Produktivita dosažená pomocí CASE není okamžitě zřejmá – na počátku je nutné vykonat velmi mnoho práce, která není dlouho vidět. Užívání CASE nezaručí konzistenci výstupů. Když se dá stejný CASE dvěma systémovým analytikům, dospějí k dvěma naprosto odlišným řešením.

Dnes již nemusí být definice CASE nástroje přesná, protože rozdíly mezi CASE, vývojovými nástroji a generátory kódů se stírají. Přes všechny zmíněné "nedostatky" CASE, které jsou způsobeny převážně špatným přístupem jejich uživatelů, mají tyto nástroje své pevné místo v projektových týmech, zvláště dnes při rostoucí složitosti navrhovaných programů a softwarových řešení.

Příklady CASE nástrojů

Mezi nejznámější CASE nástroje patří:

- Power Designer od společnosti Sybase (umožňuje definování DB struktur, tvorbu kostry DB aplikací včetně menu …)
- CASE\4\0 německé společnosti microTOOL (typický middle CASE)
- Rational Rose od společnosti IBM, dříve Rational
- ORACLE Designer od společnosti Oracle (výborný a robustní v kombinaci s databází Oracle),
- System Architect od společnosti Popkin Software and Systems
- Select Enterprise od společnosti Select (OO nástroj),
- a mnoho dalších...