

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta elektrotechnická

Program Softwarové technologie a management

Státnice v kostce

Otázky oboru Softwarové inženýrství

Vytvořeno studenty na serveru <http://statnice.stm-wiki.cz>.

PDF vygeneroval David Vávra (vavrad1@fel.cvut.cz) z verze 16.6.2009

Otázka 01 - Y36SIN

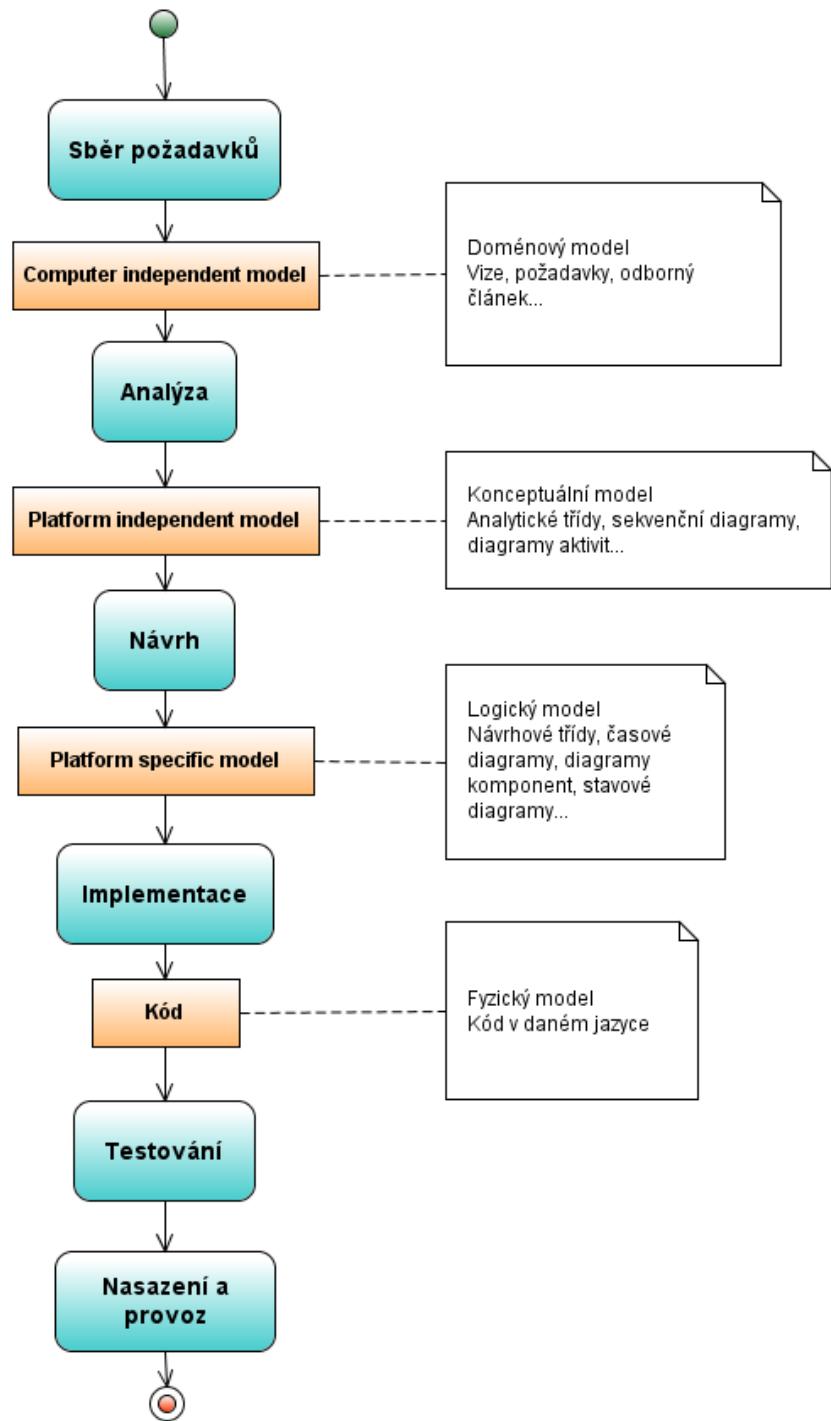
Zadání: Životní cyklus programového díla, analýza, návrh, implementace, provoz a údržba

Životní cyklus

Životním cyklem programového díla se rozumí proces, kterým software projde od zahájení svého vývoje, až po vyřazení z provozu. Tento proces samozřejmě silně závisí jak na druhu programu, tak na použité metodice vývoje, ale v každém případě projde těmito fázemi.

- Specifikace
 - Sběr požadavků
 - Analýza
- Vývoj
 - Návrh
 - Kódování
- Testování
 - Validace
 - Verifikace
- Nasazení a provoz
 - Vývoj
 - Údržba

Detailněji je tento proces zobrazen na následujícím activity diagramu.



Požadavky a analýza

Důležitým materiálem, který by se měl vytvořit je tzv. úvodní (předimplementační) studie. Jejím cílem je zhodnotit přínosy, které daný systém zákazníkovi přinese (vzhledem k současnemu stavu, případně již používanému systému), vize, rozpočet projektu, očekávané funkce, rozsah projektu, typické uživatele nebo známé aspekty problematiky. Obsah úvodní studie je v některých metodikách pevně dán, v jiných nikoliv. Případný neúspěch úvodní studie nám ušetří značné náklady na implementaci, která by byla odsouzena k nezdaru.

Sběr požadavků a analýza je klíčovou částí vývoje softwaru, protože přímo determinuje úspěch projektu. Mottem je „Do the right thing“, čili aby software skutečně dělal to, co dělat má. Pokud se v této fázi objeví chyba v zadání nebo jinde v projektu tak je jí velmi levné opravit, v případě nalezení při implementaci (nadej bože při testování) je její odstranění nesrovnatelně nákladnější.

Typickými výstupy sběru požadavků je model požadavků a Use Case diagram. Výstupy analýzy jsou například diagramy aktivit, analytických tříd nebo sekvenční diagramy.

Vývoj

Vývojem se myslí návrh a kódování. Mottem je „Do the thing right“, čili aby to fungovalo, jak to fungovat má.

Pomocí například GRASP vzorů (na to se Vás nikdo ptát nebude, dělá se to v Y36ASS) se z modelu analytických tříd přejde k modelu tříd návrhových. Zde je to správné místo na využití GOF (Gang of Four) návrhových vzorů, případně architektonických návrhových zdrojů. V momentě, kdy návrhář/architekt dokončí jak diagram návrhových tříd, tak další diagramy pokrývající tuto fázi (časový, stavový, komponent...), tak se projekt předá programátorům, kteří jej dle nich naprogramují.

Testování

O testování je zde vlastní otázka, proto nebudu zabíhat do podrobností, jde o to, že by se SW měl testovat průběžně během všech fází, protože jak jsem již zmínil, tak čím později se chyba nalezne, tím dražší je.

Nasazení, údržba a evoluce

V momentě nasazení programu jeho život zdaleka nekončí. I dnes některé banky stále používají systémy napsané v COBOLu, protože by jejich nahrazení byla velmi drahá. Dalším příkladem může být rok 2000, kdy byl strach o systémy, které byly programovány s tím, že rok 2000 je hodně daleko, a přesto se tohoto roku dožily a způsobily nemalé obavy ze svého chování. Po celé roky existence programu je zapotření zajistit uživatelskou podporu, opravu chyb (které tam nevyhnutně jsou, jejich množství a cena závisí od toho, jak jste si k srdci vzali ony poznámky o testování) a v neposlední řadě rozvoj systému. Platí, že změny v SW by se měly dít primárně pomocí konfiguračních souborů, protože je to významně levnější postup, než sahat přímo do kódu. Další pravdou je, že by se systému, kde může dojít snadno ke změně mělo s tímto počítat (nebudeme přepisovat celý účetní program, když se změní zákon).

Metodiky

Pro úplnost bych zde měl zmínit jednotlivé metody vývoje SW.

Rational Unified Process

Velmi známou agilní metodikou je Rational Unified Process (RUP), který spočívá v iterativním opakování sběru požadavků, analýzy, návrhu, implementace a testování v jednotlivých fázích (zahájení, rozpracování, konstrukce, nasazení). V každé fázi je kladen jiný důraz na jednotlivé části (při zahájení se nebude moc testovat, při nasazení je na požadavky již trochu pozdě).

Vodopád

Základní myšlenkou je, že se sběr požadavků, analýza, vývoj a testování dělí lineárně jako oddělené fáze a zpět se již nevrací. Tento metodiku i dnes pracuje asi 53 procent firem (k roku 2004). Model vodopádu má několik obrovských nevýhod. První z nich jsou nákladné opravy principiálních chyb, které se objeví často až při testování na konci. Druhou velkou nevýhodou je, že je SW hotov, až když je hotov, čili zákazník nevidí žádný postup a když mu dojdou peníze po cestě, tak má sice tunu diagramů, ale nic, co by se mu hodilo. Třetí nevýhodou je, že když zákazník nic nevidí, tak nemůže včas zarazit nějakou chybu z nepochopení, kterou on vidí ihned, ale programátor, který v problematice není sběhlý ji nevidí. Čtvrtou nevýhodou je, že zákazník mění své požadavky většinou poměrně rychle a na to je zapotřebí reagovat, což vodopád neumožňuje.

RAD model

Jedná se o velmi rychlý vodopád určený pro dobře vymezené projekty s krátkým vývojovým cyklem (méně než 3 měsíce). Systém je rozdělen na moduly, které vytvářejí jednotlivé teamy vesměs z již hotových SW komponent.

Evoluční modely

Evoluční modely provádějí nějakým způsobem průběžnou validaci softwaru tak, aby tento byl co nejrychleji ve shodě s požadavky zákazníka.

Přírůstkový model

SW je programován po jednotlivých přírůstcích. První přírůstek označujeme jako „ jádro“ a ten je postupně obalován dalšími přírůstky, dokud není aplikace hotova. Model je vhodný pro projekty, které se dají vhodně rozdělit na jednotlivé části.

Spirálový model

Jedná se o kombinaci mezi prototypováním (zahoditelný model systému na určité úrovni abstrakce) a opakováním sekvenčním vývojem. V každé otočce spirály se provede riziková analýza a následuje vývoj části s nejvyšší rizikovostí. Výsledkem každé otočky je prototyp na vyšší úrovni abstrakce, než byl v otočce přechozí.

Formální metodiky

Formální metody spočívají na formálních specifikacích a ve využití formální verifikace programů. Jsou drahé a náročné jak na zákazníka, tak vývojáře (příliš se nepoužívají).

Agilní metodiky

Principem agilních metodik je důraz na programátora jako jednotlivce, nikoliv jako součást firemního soukolí (důraz na osobní komunikaci - ne dokumenty, rovnostářská struktura teamu - ne hierarchická, odpovědnost jednotlivce za implementaci jednotlivé funkcionality - způsob vnitřního fungování programu není programátovi příkázán shora). Tyto vlastnosti kladou ale požadavky na team, jedním z nich je, že by team měl sedět v jedné kanceláři, aby docházelo k permanentnímu sdílení vědomostí o systému. Z toho plyne, že jsou tyto metodiky nevhodné pro velké projekty s velkým počtem účastníků. Další vlastností agilních metodik je možnost průběžného předávání použitelného softwaru (pro uživatele je lepší, když má program, který splňuje alespoň část funkcionality, než když nemá nic), s čímž souvisí komunikace se zákazníkem, z čehož vyplývá důraz na provádění změn (SW se dělá kvůli zákazníkovi, když to chce jinak, tak to chce jinak...to, že to neodpovídá původnímu plánu je mrzuté, ale dělá se to PRO zákazníka, ne KVŮLI zákazníkovi). Co se týče délky jednotlivých iterací, tak tyto jsou fixní, nikoliv konstantní (každá může mít jinou délku, ale pokud se nestihá, tak se iterace nepodložuje, ale neimplementovaná funkcionality se přesune do další iterace a ta se přepracuje).

Extrémní programování

Druhou velmi známou metodikou je extrémní programování (XP). Tato metodika klade obrovský důraz na spolupráci se zákazníkem, vyvíjí se vždy ta část SW, na kterou zákazník ukáže. Programovat se začne až tehdy, kdy jsou hotové testy implementované funkcionality (Test driven development), programuje se minimální množina příkazů, která ještě splňuje zadání (žádné kdyby přišlo toto, tak by se hodilo...). Programuje se ve dvojicích (které nejsou stálé), tím se zajišťuje vyšší čitelnost a nižší chybovost kódu. Zdrojové kódy se sdílí (každý může sahat do libovolné třídy programu bez ohledu na to, kdo ji originálně napsal). V neposlední řadě v XP neexistují přesčasy, protože nic není horší, než našvaný a demotivovaný programátor.

Literatura

- Přednášky z Y36ASS
- Přednášky z Y36SI3

- Ing. Božena Mannová M. Math, RNDr. Karel Vosátka CSc. - Řízení softwarových projektů
- Jim Arlow, Illa Neustadt - UML 2 a unifikovaný proces vývoje aplikací - Průvodce analýzou a návrhem objektově orientovaného softwaru
- http://cs.wikipedia.org/wiki/Extr%C3%A9mn%C3%AD_programov%C3%A1n%C3%AD#Testov.C3.A1n.C3.AD [http://cs.wikipedia.org/wiki/Extr%C3%A9mn%C3%AD_programov%C3%A1n%C3%AD#Testov.C3.A1n.C3.AD]
- http://en.wikipedia.org/wiki/Agile_software_development [http://en.wikipedia.org/wiki/Agile_software_development]
- http://www.fit.vutbr.cz/study/courses/PPS/public/pdf/2_Ziv_cykl.pdf [http://www.fit.vutbr.cz/study/courses/PPS/public/pdf/2_Ziv_cykl.pdf]
- http://www.sntcz.cz/Content.Node/solutions_services/professional_services/22757.cz.php [http://www.sntcz.cz/Content.Node/solutions_services/professional_services/22757.cz.php]
- <http://www.dbsvet.cz/view.php?cisloclanku=2003061102> [<http://www.dbsvet.cz/view.php?cisloclanku=2003061102>]

si/si1.txt · Poslední úprava: 2009/06/16 12:24 autor: Tasadar

Otázka 02 - Y36SIN

Zadání: Modelovací prostředky, UML, diagramy UML, jazyk OCL

Modelovací prostředky

- účel modelování spočívá v usnadnění, porozumění a pochopení systému, jeho funkce a smyslu
- vizuální
 - díky vizuálním prostředkům je snazší zorientovat se v celkové problematice než díky slovnímu popisu
 - UML: (class diagram, activity diagram, ... - viz. dále)
 - Craft.CASE: (nástroj na modelování obchodních procesů)

Jazyk UML (Unified Modeling Language)

UML je jednotný jazyk (grafický) pro specifikaci, vizualizaci, konstrukci a dokumentaci při OO analýze a návrhu (OOAaD) a pro modelování organizace (business modelling).

- vznik řady OO metod a metodologií (konec 80. let a první polovina 90.let) podobné notace vyjadřující totéž, komplikující rozdíl snaha o standardizaci týmem OMG
- 1995:
 - Booch + Rumbaugh (Rational Software): Unified Method v.0.8
 - Rational Software kupuje Objectory (Jacobson)
- 1997: UML v.1.1 standardem OMG (základem UML 1.0 od Rational)
- UML v.1.2
- 1999: UML v.1.3
- 2001: UML v.1.4
- snaha povzbudit vývojáře k modelování systémů před jejich vytvářením a umožnit univerzálnost nástrojů vizuálního modelování (interoperability)

Proč modelujeme? lepší pochopení vyvíjeného systému

Cíle modelování?

- vizualizace (jaký je nebo má být)
- specifikace struktury a chování systému
- „šablona“ pro konstrukci systému
- dokumentuje provedená rozhodnutí

Zásady modelování:

- volba modelů má vliv na zvládnutí problému a podobu řešení
- každý model lze vyjádřit na různé úrovni podrobností
- nejlepší modely jsou ty, které jsou spojeny s realitou
- většinou nestačí jedený model, ale je třeba vytvořit několik „nezávislých“ modelů

UML má čtyři mechanismy, které se prolínají celým jazykem.

Tyto mechanismy jsou :

specifikace

každý element může (či měl by?) být specifikován textem, který popisuje sémantiku tohoto elementu. Tato specifikace upřesňuje, blíže popisuje, udává smysl modelovaného elementu. Popisuje **business pravidla** elementů (tudíž má největší význam u

elementů popisujících problémovou doménu).

ozdoby (adornments)

další informace známé o elementu modelu. Každý element může být vyjádřen jednoduchým tvarem, ale je možno k němu přidávat i další informace - ozdoby (např. seznam metod, seznam atributů, název objektu, stereotyp,...).

Proč je těchto ozdob u elementu zobrazeno někdy více a někdy méně:

- model vytváříme postupně : zpočátku máme málo informací, které postupně doplňujeme
- tvorbou určitého diagramu sledujeme určité cíle - nechceme v něm zobrazit ty podrobnosti, které jsou v tomto nepodstatné z hlediska toho, co právě chceme zdůraznit (jedno z pravidel hledisek při tvorbě diagramu je totiž snadná čitelnost)

podskupiny (common division)

udávají, jak je možno rozdělovat (skupinovat) jednotlivé elementy; první způsob dělení :

- klasifikátor a instance : pro dva elementy UML jsme si toto rozdělení již probrali - objekt je instance, kdežto třída je klasifikátor. Podobný vztah klasifikátor-instance lze nalézt pro další elementy UML. Každý element je buď klasifikátor nebo instance, a toto rozlišení je velmi důležité. Osvojením tohoto dělení si usnadníte komunikace mezi členy týmu (stačí říct : "... klasifikátor elementu xxxx, instance elementu yyyy" a bez dalšího vysvětlování je jasné, o co jde), můžeme se s tímto setkat i v CASE nástrojích a v literatuře.
- rozhraní a implementace : zalistujme výše v tomto kurzu do kapitoly popisující objekt - mluvili jsme o protokolu zpráv, což je rozhraní objektu. Implementace pak jsou metody, které řeší, implementují, toto rozhraní. (Obecně však může být definované rozhraní pro každý klasifikátor).

Toto důrazné oddělení má dvojí praktický význam :

- k tomu, abychom použili (již hotový) objekt, nemusíme znát jeho implementaci - stačí nám znát jeho rozhraní; programátoři (i neobjektoví) vlastně tohoto využívají při volání knihovních funkcí : programátor v jazyce C nemusí znát, jak je vnitřně vyřešena funkce printf, ale zná její rozhraní, tedy ji může používat
- a z druhé strany : vnitřek objektu může být (v budoucnu) libovolně změněn - ale jen tak, aby jeho klienty (tj. ty, které využívají jeho služby) nemuselo být nutno revidovat - jinak řečeno : i po úpravách musí objekt správně implementovat dohodnuté rozhraní; tedy ten, kdo vytváří/mění vnitřek objektu, nemusí nic vědět o tom, jak a když bude objekt použit - stačí, když bude správně implementovat rozhraní

mechanismy rozšiřitelnosti

jaký UML sám v sobě obsahuje připravené mechanismy umožňující rozšířit jazyk tak, aby vyhovoval momentálním potřebám. Máme k dispozici tři mechanismy rozšiřitelnosti :

- **omezení** (constraints) : jde o text ve složených závorkách {}. Podmínka či pravidlo v tomto textu musí být vždy splněna
- **stereotypy** (stereotypes) : s jejich pomocí lze z existujícího elementu vytvořit nový. Vytvoříme ho tak, že název stereotypu vložíme do dvojitých ostrých závorek : «novy_stereotyp». Stereotyp může mít rovněž přiřazen nový symbol - časté využití bývá např. v diagramu nasazení pro vytvoření symbolů tiskáren, serverů, notebooků apod. Některé stereotypy jsou již součástí jazyka UML a ještě se s nimi v tomto kurzu setkáme.
- **označené hodnoty** (tagged values) : umožňuje přidávat nové vlastnosti k elementům modelu. Zavedeme ji přidáním názvu s připojenou vlastní hodnotou ve složených závorkách, např. {autor=pavus, verze=0.1}
- **profily** : Profil UML definuje množinu stereotypů, značek a omezení, díky nimž lze jazyk UML přizpůsobit konkrétnímu účelu. (např. můžeme mít nějaký profil modelování aplikací pro platformu J2EE, jiný profil pro .NET, apod.)

Diagramy UML

model je souhrn všech předmětů a relací, jejichž znázornění v jednom obrázku by bylo nepřehledné, mnohdy zcela nemožné; naproti tomu diagram je jeden průhled, okénko, kterým se díváme na model;

diagramů je v UML několik typů, ale většinou máme v jednom projektu i více diagramů jenho typu (ať už z důvodu velikosti modelu, nebo chceme v různých diagramech zobrazit pohled na systém z různých úhlů, či se v různých diagramech chceme zaměřit na různé aspekty modelovaného systému).

je dobrým zvykem modelovat pouze ty diagramy, které přinášejí užitek

v diagramech bývají objekty znázorněny podobně jako třída - obdélníkem, název objektu je však vždy podržený

Máme tedy devět diagramů ve dvou skupinách:

*diagramy struktury(zaměřený na systémovou strukturu):

- diagram tříd
- diagram komponent
- diagram vnitřní struktury (composite structure diagram)
- diagram nasazení
- diagram balíčků
- diagram objektů, též se nazývá diagram instancí

*diagramy chování (zaměřený na chování systému):

- diagram aktivit
- diagram užití
- stavový diagram
- diagramy interakcí:
 - sekvenční diagram
 - diagram komunikace
 - přehled interakcí
 - diagram časování

Popis jednotlivých diagramů

diagram tříd

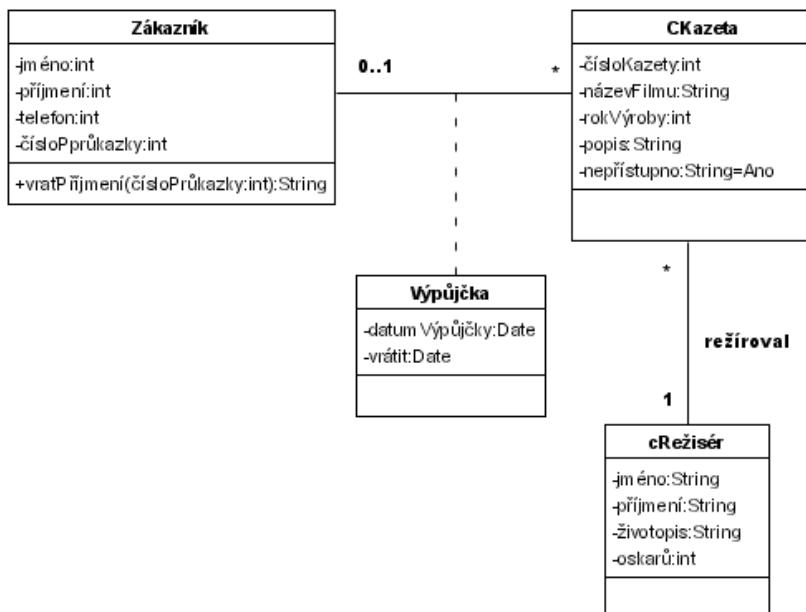
Zobrazuje statickou strukturu systému, popisuje typy objektů a statické vztahy mezi nimi.

Tři pohledy na systém při použití diagramu tříd

- Konceptuální – koncepty aplikační domény, bez vztahu k implementaci, jazykově nezávislá.
- Specifikační – pohled na program, specifikace rozhraní bez specifikace implementace.
- Implementační – je vidět implementace - hrance nejsou ostré, UML podporuje všechny tři pohledy

Tipy:

- nesnažit se použít veškerou dostupnou notaci
- přizpůsobit pohled etapě projektu:
 - analýza: konceptuální pohled
 - návrh: specifikační + implementační
- koncentrovat se na klíčové oblasti, raději méně aktuálních diagramů
- nezabírátnout brzy do implementačních detailů



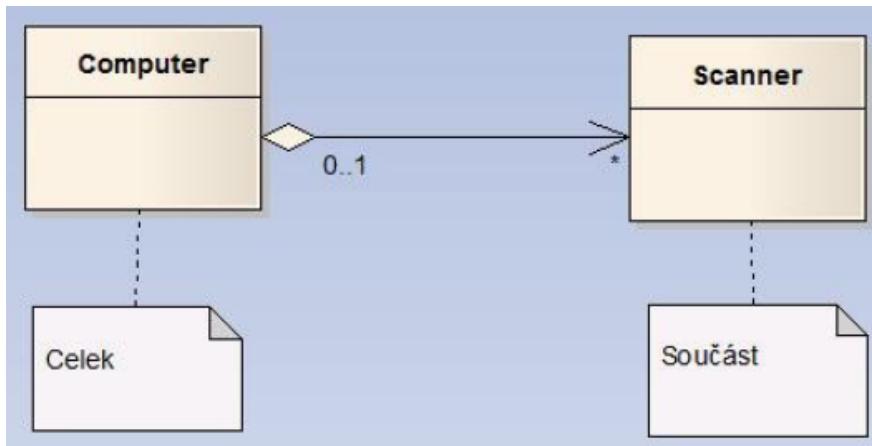
Created with Poseidon for UML Community Edition. Not for Commercial Use.

Na obrázku je příklad návrhového class diagramu z půjčovny videokazet:

- vidíme zde třídy **Zákazník**, **CKazeta**, **CRežisér** a asociativní třídu **Výpůjčka**
- třída má v horním oddílu (compartment) název, v prostřeném oddílu může být seznam atributů (např. u třídy Zákazník jsou to jméno, příjmení,) a ve spodním oddílu seznam **operací** (zde **vratPříjmení**)
- asociace mezi třídami je znázorněna (v naprosté většině případů) jako plná čára spojující dvě třídy; asociace může mít mnoho různých **ozdob** (adornments)
- konkrétní zákazník může mít vypůjčenu žádnou, jednu nebo více kazet (zobrazeno pomocí „*“ u asociace mezi třídami Zákazník a CKazeta, a to u konce připojeného ke třídě CKazeta)
- konkrétní kazetu může mít zapůjčenu žádný nebo jeden zákazník (zobrazeno pomocí „0..1“ na konci asociace připojeném ke třídě Zákazník)
- Kazetu (resp. film) režíroval právě jeden režisér
- protože asociace mezi zákazníkem a kazetou musí mít vlastní atributy (musí si pamatovat datum výpůjčky a datum, kdy by měla být kazeta vrácena), zavedli jsme **asociační třídu** **Výpůjčka**

Návrhové relace

- **Agregace**
 - Volná vazba mezi objekty
 - Typ celek/součást (celek řídí chod relace, součást poskytuje služby)
 - Agregované objekty
 - mohou být sdílené
 - nezanikají se zánikem celku
 - Celek někdy existuje nezávisle na součástech, někdy je na nich závislý
 - Součást může být sdílena více celky



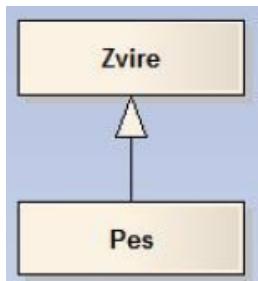
■ Kompozice

- Silnější forma agregace
- Součásti nemohou existovat mimo celek
- Součást patří právě jednomu celku
- Celek odpovídá za použití svých součástí
- Za předpokladu, že odpovědnost za součásti přejde na jiný objekt, může celek součásti uvolnit.
- Je-li zničen celek, musí zničit své součásti (nebo přenést odpovědnost na jiný objekt)



Generalizace

- Jako zobecnění (generalizace) se označuje relace mezi obecnějším a přesněji definovaným prvkem.
- Na následujícím obrázku je třída Zvíře obecnější než třída Pes.



Dědičnost tříd

- Potomci (podtřídy) přebírají charakteristiku svých předchůdců
- Potomci dědí
 - Atributy
 - Operace
 - Relace
 - Omezení
- Potomci mohou ke zděděnému fondu přidat novou charakteristiku

Realizace

- Fakt, že třída implementuje (realizuje) nějaké rozhraní znázorňujeme čárkovanou čárou, která směřuje od třídy k rozhraní.
- Na následujícím obrázku třída `TimeStamp` implementuje rozhraní `TimeComparable`.
- Rozhraní se používá k určení společných protokolů pro třídy, mezi nimiž by za normálních okolností neměly být dědičné vazby.

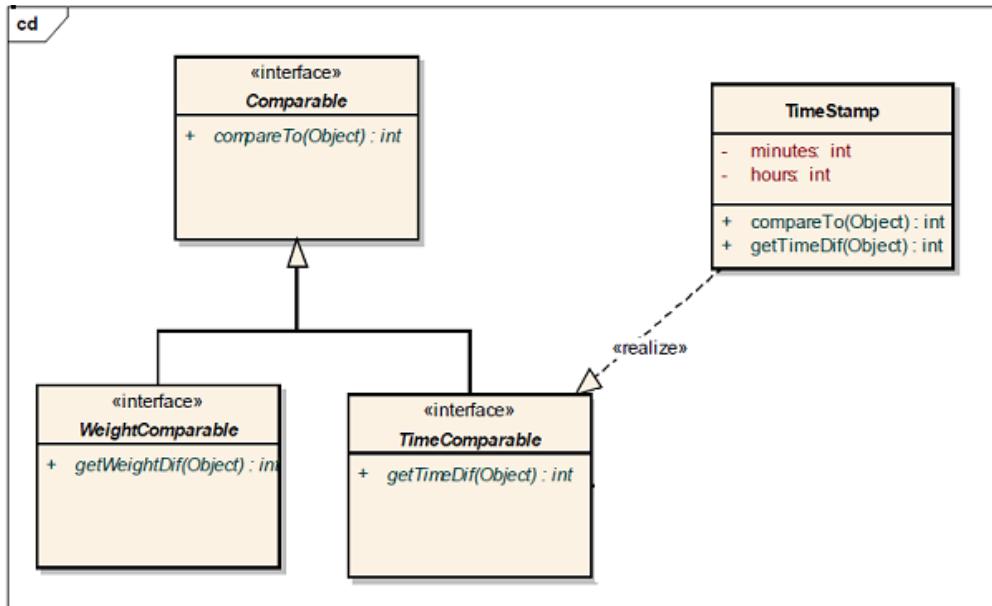


diagram komponent

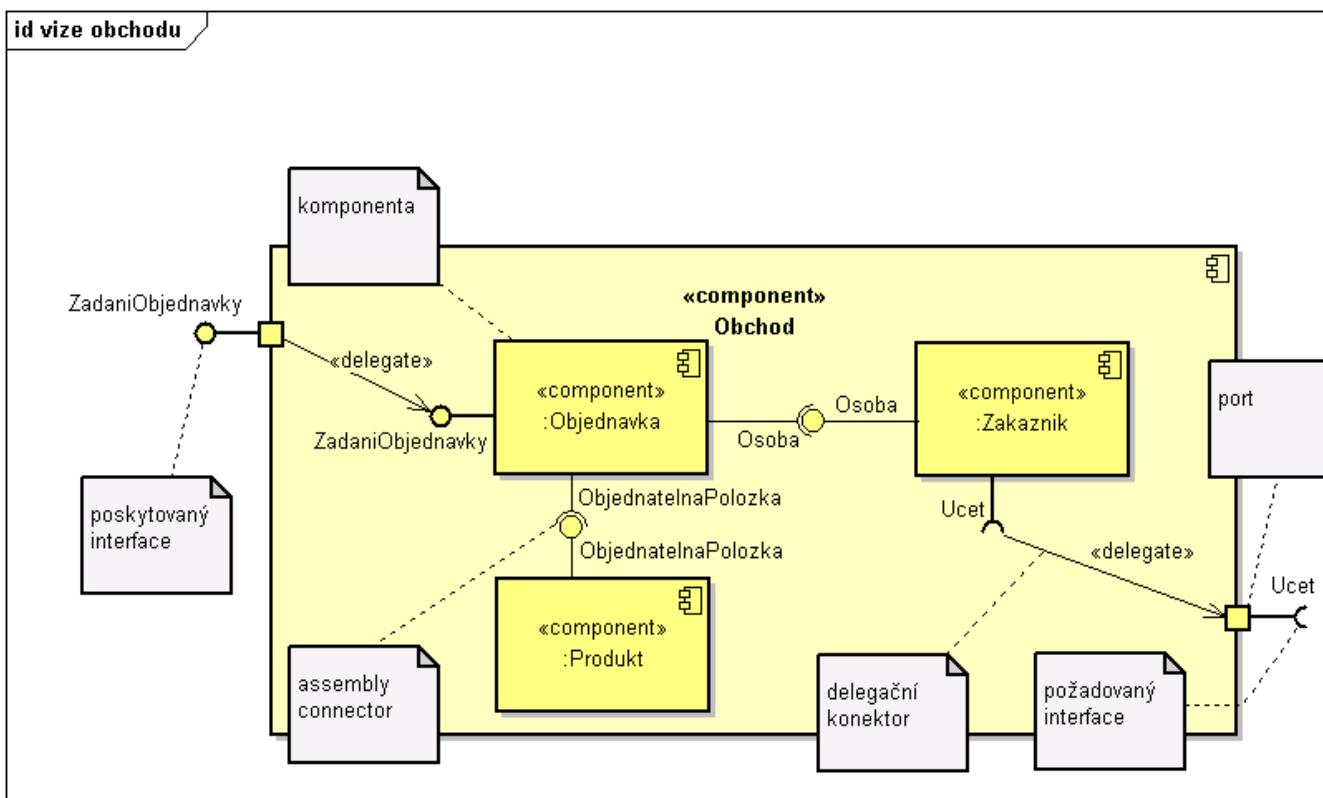
Popisuje jak jsou jednotlivé komponenty (části) systému propojeny. Komponenta je fyzická nahraditelná část systému, která obaluje implementaci a poskytuje realizaci množiny specifikovaných rozhraní.

Diagram komponent znázorňuje komponenty použité v systému, tj. logické komponenty (např. business k., procesní k.) či fyzické komponenty (např. EJB k., CORBA k., .NET k. atd.). D. komponent nám může pomoci zejména tam, kde používáme vývoj založený na komponentách a kde struktura systému je založena na komponentech.

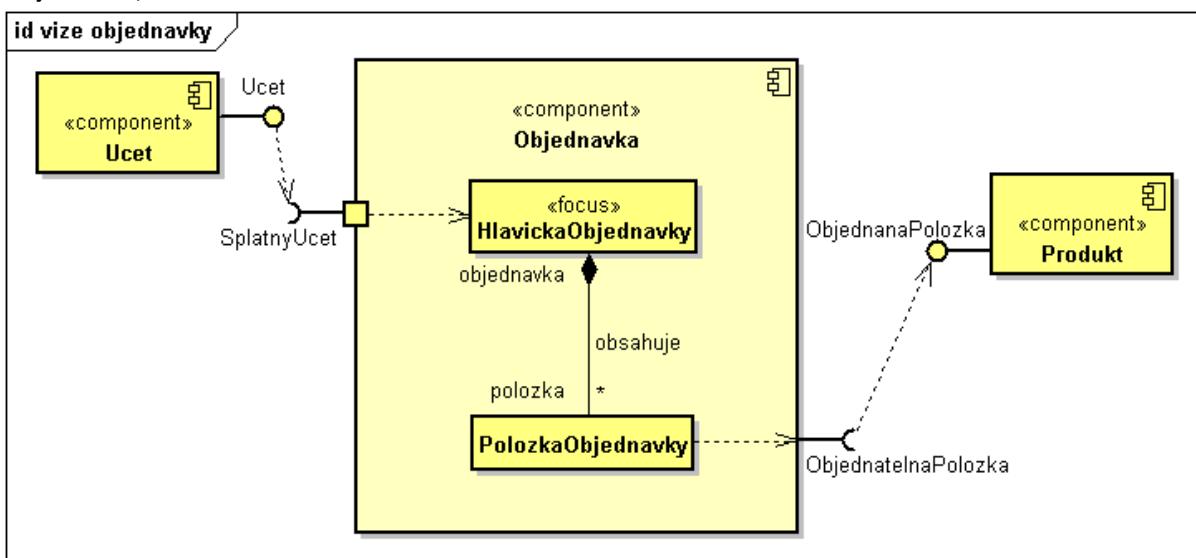
Komponenta je modulární část systému, která zapouzdřuje svůj obsah (tj. zapouzdřuje stav a chování i více klasifikátorů) a jejíž projev je nahraditelný (tj. komponenty, poskytující ekvivalentní funkcionality založenou na kompatibilitě jejich interfejsů, mohou být libovolně zaměňovány, a to buď už v čase tvorby designu, nebo až za běhu cílového systému). Komponenta je specializací strukturované třídy. Protože je komponenta specializací třídy, může mít své atributy a operace a může mít asociace a generalizace-specializace (více viz diagram tříd (class diagram)).

příklady komponent:

- zdrojový soubor
- ActiveX prvek
- Enterprise JavaBean objekt
- Java servlet
- stránka JSP



Na prvním obrázku je diagram komponent s jednou strukturovanou komponentou **Obchod** a jejími třemi vnitřními komponentami :**Objednavka**, :**Produkt** a :**Zakaznik**.



druhém obrázku je další diagram komponent, kde vidíme komponentu **Objednavka** a její vnitřní klasifikátory **HlavickaObjednavky** a **PolozkaObjednavky**, které realizují chování této komponenty.

V diagramech můžeme vidět :

- **komponenta** je znázorněna jako symbol klasifikátoru (pravoúhelník), s klíčovým slovem «component». Volitelně má v pravém horním rohu zobrazenou speciální malou ikonu komponenty. (PS.: ve verzi UML 1.x vypadal symbol komponenty právě tak, jako ona speciální ikona, přitom starý symbol komponenty je přípustný i ve verzi UML 2.0).

Existují dva druhy komponent :

- **basic component** : pro ni platí vše, co je v této kapitole o komponentě řečeno (pokud není výslovně uvedeno, že se informace týká jen komponenty typu packaging)
- **packaging component** : rozšiřuje komponentu typu basic o možnosti skupinování; takovéto komponenty se týkají aspekty spojené s problematikou **namespaces** (komponenta může dělit a importovat své členy, jako např. packages, use cases, komponenty, artefakty, třídy,)
- **poskytovaný** (provided) **interface** je implementován přímo danou komponentou (popř. klasifikátorem, který realizuje chování komponenty), nebo je to vlastně typ poskytovaného portu (zde poskytovaný interface **ZadaniObjednavky** je připojen na nepojmenovaný port).
- **požadovaný** (required) **interface** je typem požadovaného portu, nebo je určený závislostí usage přímo z dané komponenty (popř. klasifikátoru, který realizuje chování komponenty), (zde požadovný interface Ucet komponenty Obchod (1. diagram) je připojen na nepojmenovaný port, požadovaný interface **ObjednateInaPolozkaUcet** komponenty **Objednavka** (2. diagram) je připojen pomocí závislosti na interní třídu **PlozkaObjednavky**).
- **port** nám umožňuje organizování interfejsů do skupin : můžeme vytvářet pojmenované sady interfejsů
- **delegační konektor** (delegation connector) propojuje externí rozhraní komponenty (tak, jak je specifikováno portem) s vnitřní částí komponenty (tj. s vnitřní komponentou či třídou anebo s portem vnitřní komponenty).
- **montážní konektor** (assembly connector) spojuje požadovaný a poskytovaný interfejs; strana, která poskytuje interfejs, musí být schopna poskytnout minimálně všechny služby, které může požadovat strana s požadovaným interfejsem

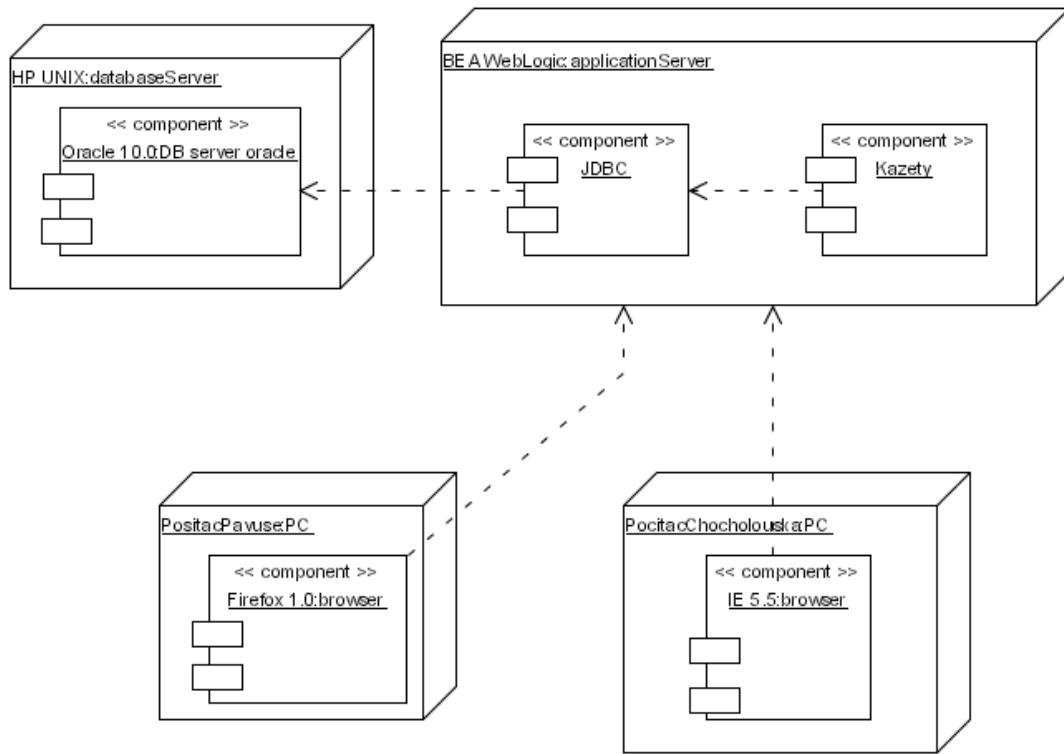
diagram nasazení

Diagram nasazení ukazuje rozmístění zdrojů (např. HW) a softwarové komponenty, procesy a objekty, které na nich žijí.

Zachycuje fyzickou architekturu počítačového systému. Pomocí něho je možné zobrazit počítače a zařízení, znázornit jejich vzájemná připojení a také software, který je na určitém zařízení nainstalován.

Hlavním hardwarovým prvkem je uzel, což může být jakýkoliv druh výpočetního prostředku. Uzel může být dvojího druhu, a to procesor, nebo zařízení (např. tiskárna nebo monitor). Procesor, narození od zařízení, umí spouštět komponentu. Uzel se zakresluje jako kvádr. Odlišení, zda se jedná o procesor nebo zařízení, je možné znázornit např. pomocí stereotypů «procesor» a «zařízení». V uzlu mohou být uvedeny i další informace, jako např. komponenty šířené na uzlu.

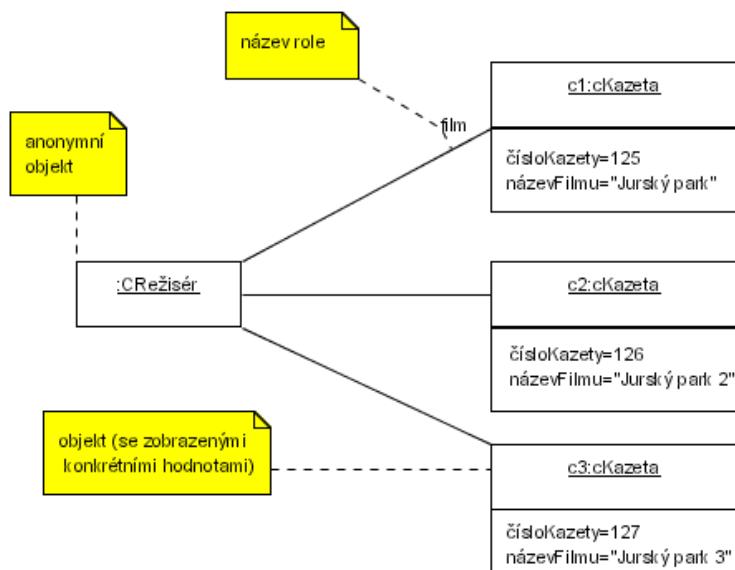
Mezi uzly mohou existovat různé typy vazeb, jako je spojení, agregace a závislost.



Zde vidíme instance uzlů, tedy např. **PocitacPavuse** který je instancí typu uzlu **PC**, HP Unix databázový server atd. Také zde vidíme instance komponent, a na kterých kusech HW tyto komponenty běží. V těchto diagramech se velmi často používají stereotypy, a to tak, že mají přiřazen vlastní vizuální symbol - tedy např. tiskárna nevypadá jako krychle, ale opravdu nám připomíná tiskárnu, nebo server opravdu může vypadat jako schématické znázornění serveru. Pak jsou tyto diagramy přístupné i pro čtenáře, kteří naprostě neznají UML.

diagram objektů

Objektový diagram zobrazuje objekty a jejich spoje (links) v jednom okamžiku, tj. je to snapshot běžícího systému (či jeho části). Objekty jsou instancemi tříd. Objekty jsou propojeny linky. Objektový diagram vypadá podobně, jako diagram tříd. Název konkrétního objektu se zapisuje před název třídy a odděluje se dvojtečkou.



Created with Poseidon for UML Community Edition. Not for Commercial Use.

Na obrázku je příklad objektového diagramu odvozeného z class diagramu naší půjčovny videokazet. :

- objekt je znázorněn podobně jako třída - obdélníkem, ale název objektu je vždy podtržený
- zápis **c1:CKazeta** značí objekt **c1**, který je instancí třídy **CKazeta** (obdobně je tomu s objekty **c2** a **c3**)
- zápis **:CRežisér** značí tzv. **anonýmní objekt** : nevíme, jak se objekt jmenuje, ale víme, že které třídy byl objekt odvozen
- třetí možnost zápisu by byla **c4**, tj. objekt bez vyznačení třídy : nevíme, ze které třídy je objekt odvozen (nebo to autor věděl, ale chtěl znázorit, že objekt **c4** mohl vzniknout z různých tříd)
- **spojení** (link) mezi objekty vzniklo jako instance asociace
- objekt **:CRežisér** je znázorněn v minimálním tvaru (jen jeho znázev), objekt **c1:CKazeta** je znázorněn s dalším * oddílem (kompartiment) ukazujícím aktuální hodnoty atributů
- u objektu **c1** je vyznačena role - tj. ve vztahu anonymního objektu **:CRežisér** versus **c1:CKazeta** hraje tato kazeta roli filmu (který byl režirován spojeným objektem - režisérem)

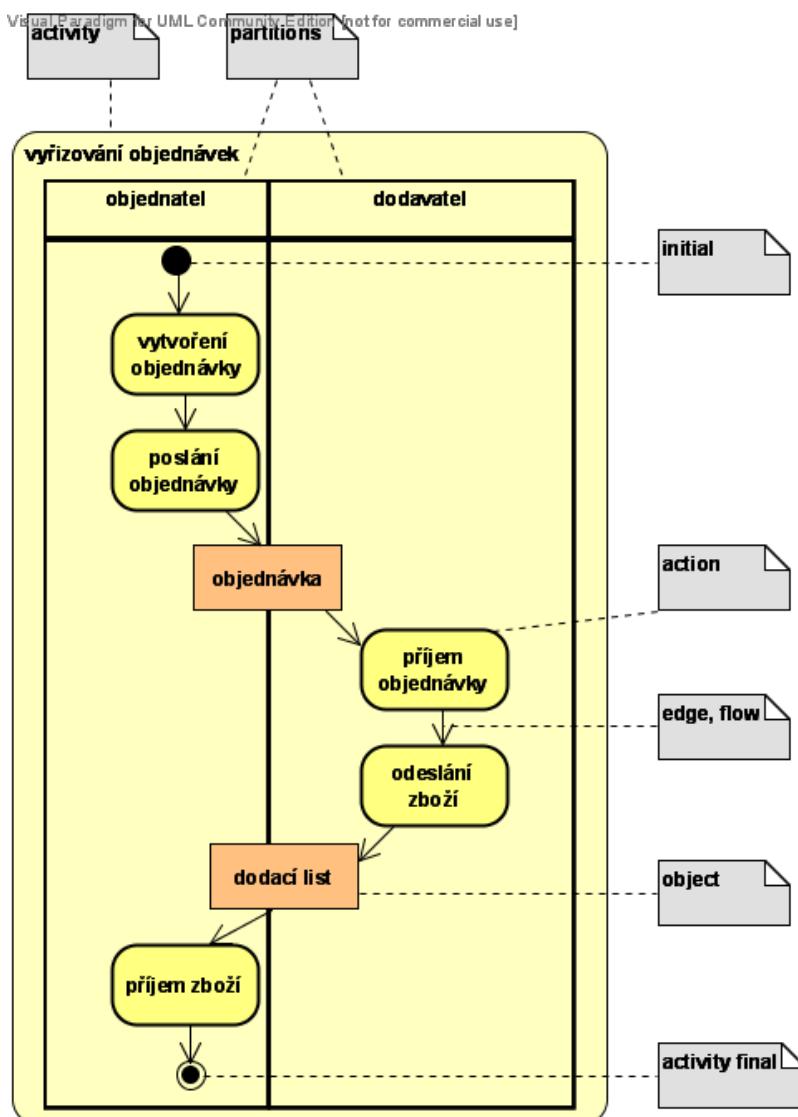
diagram aktivit

Diagram aktivit se používá pro popis dynamických aspektů systému.

Jde o jakýsi flowchart - znázorňuje tok řízení z aktivity do aktivity. Používá se také k modelování obchodních (business) procesů a workflow.

Diagram aktivit se soustředí spíše na proces výpočtu než na objekty účastnící se výpočtu (i když i objekty mohou být znázorněny jako prvek aktivity).

Diagramy stavů a diagramy aktivit jsou si podobné (oba ukazují sekvenci stavů, které nastávají v čase, a ukazují podmínky způsobující přechody mezi stavů). Rozdíl mezi těmito diagramy je však v tom, že d. stavů se soustředí na stavu objektu (tj. objektu provádějícího výpočet či objektu, se kterým je výpočet prováděn), kdežto d. aktivit se zaměřuje na **stav samotného výpočtu** (stav procesu, algoritmu,), kde může být účastno i více objektů, a kde jsou znázorněny řídící a informační toky mezi prvky diagramu.



Na obrázku je příklad jednoduchého diagramu aktivit:

- **aktivita** (activity) **vyřizování objednávek** je ta, která je zde modelována (pro znázornění, jak vlastně toto vyřizování objednávek probíhá)
- **oddíly** (partitions, swimlanes): plavební dráhy - rozdělení diagramů na více částí pro znázornění odpovědností za různé části aktivity; zde jde o rozdělení aktivity na dvě různé firmy
- **akce** (action) : nejprimitivnější, nejnižší prvek výpočtu. Nemůže být dále dekomponován.
- **hrana, tok** (edge, flow) zobrazuje přechod z jedné aktivity do další (v příkladu je přechod vždy automatický, dojde k němu po ukončení předcházející akce)
- **objekt** (object) může vstupovat do aktivit, může být jejich výsledkem
- **počáteční, finální uzel aktivity** (initial, activity final) : zvláštní uzly pro označení počátku a konce aktivity

Aktivity mohou být spouštěny akcemi a nebo jako součásti jiných chování, např. přechody v diagramu stavového stroje, metodou, případem použití, jinou aktivitou... Aktivita je zobrazena v diagramu aktivitu.

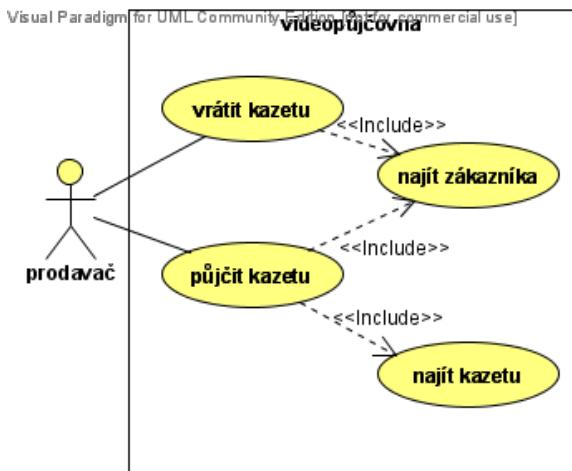
Aktivita je specifikace chování - popisuje sekvenční a souběžné kroky výpočetní procedury. Aktivita je modelována jako graf uzlů **aktivity** (activity nodes) propojených **řídícími a datovými toky** (control flow, data flow). PS.: Uzel aktivity je něco naprostě jiného než uzel v deployment modelu! Paradoxně na tu zmatečnost je poukazováno přímo v UML reference manuálu.

Uzly aktivity jsou :

- vložené aktivity
- akce: modelují účinky na systém
- lokace dat
- řídící konstrukce (control construct)

diagram užití

Use case diagram (UC diagram) zobrazuje chování systému (nebo jeho části) z hlediska uživatele.



Na obrázku je use case diagram jednoduchého systému půjčovny videokazet:

- každý **případ užití**, typová činnost (use case) charakterizuje určité použití systému účastníkem; případ užití má jméno, může mít textovou specifikaci
- **účastník** (actor) reprezentuje kohokoliv (či cokoliv) mimo systém, kdo se systémem komunikuje a interaguje (člověk, HW, čidlo, jiný systém, ...); jediné, co actor může, je přijímat nebo předávat do systému informace

PS.: spíše než konkrétního uživatele reprezentuje actor roli : jeden konkrétní účastník (člověk, technické zařízení, jiný informační systém,) může být vyjádřen i více actory (vystupují ve stejné roli), a více různých konkrétních účastníků může být vyjádřeno jedním actorem (vystupují ve společné roli)

- **ohraničení** (boundary) udává hranice systému/susbsystému (zde videopůjčovna); obecně je to vlastně klasifikátor (systém/subsystém/třída), jehož funkcionality pomocí use case popisujeme
- **vztahy, relace** (relationships) :
 - asociace mezi **prodavač** a **půjčit kazetu**: jde o **komunikační asociaci**, vyjadřuje tok informace mezi vnějším prvkem a případem užití
 - závislost mezi půjčit kazetu a najít zákazníka má stereotyp «include» : UC půjčit kazetu zahrnuje do sebe UC najít zákazníka (zahrnovaný use case); toto nám umožňuje re-use : najít zákazníka v systému musíme jak tehdyn, když si chce zákazník půjčit videokazetu, tak tehdyn, když videokazetu vrací - «include» nám umožní nadefinovat hledání zákazníka jen jednou a opakováně ho zahrnout do více případů užití (vrátit kazetu, půjčit kazetu)

V diagramu je naznačeno, jak actor používá systém, ale jsou uvedeny jen názvy činností - my však potřebujeme specifikovat další podrobnosti: zde není jiná možnost, než naplnit do jednotlivých UC textové specifikace (jiná možnost ovšem je - podrobnosti lze namodelovat dalším navázaným diagramem, např. sekvenčním).

V těchto specifikacích by mělo být zejména popsáno :

- co systém dělá (ale ne jak to dělá)
- měly by být používány jen pojmy problémové domény
- jak a kdy činnost začíná a končí
- kdy má systém interakce s actorem
- které údaje jsou měněny

- jaké kontroly vstupních údajů jsou prováděny
- základní, alternativní a chybové průběhy
- způsoby popisu nejsou v UML formalizovány: může to být strukturovaný/formátovaný text (pre- a post-conditions), pseudokód, ...

Příklad popisu případu užití vrátit kazetu : vysvětlující komentáře budou psány takto

případ užití : vrátit kazetu název případu užití ID: UC_02 jednoznačný identifikátor **Účastníci : prodavač seznam účastníků, kteří se podílejí na komunikaci Vstupní podmínky**

1. zákazník chce vrátit (dříve vypůjčenou) videokazetu
2. prodavač je zalogován do systému

zde jsou obsaženy všechny podmínky, které musí být splněny, aby mohl být use case spuštěn; neboli : všechny uvedené podmínky musí být vyhodnoceny jako pravda; může jít jak o podmínky, jejichž pravdivost je „ověřitelná“ přímo systémem (podmínka 2), tak o stav okolního světa (podmínka 1) PS.: tak jednoduché a obecné podmínky jako zalogování do systému se zde však obvykle neuvádějí...

Tok událostí

1. Systém spustí případ užití (UC_05 najít zákazníka)

zde je spuštěn zahnutý případ užití pro nalezení konkrétního zákazníka (UC05) - ten vyzve uživatele k zadání čísla průkazky zákazníka, podle čísla průkazky ho vyhledá a vrátí jeho identifikaci zpět do klientského případu užití (tj. nic nezobrazí) : vlastní zobrazení zákazníka si zajišťuje každý klientský případ užití „po svém“ (každý by mohl chtít zobrazit jinou sadu údajů zákazníka a jiným způsobem)

1. Systém zobrazí údaje vyhledaného zákazníka jako jméno, příjmení, adresu, telefon a vypůjčené kazety jako seznam záznamů : číslo kazety + název filmu + datum výpůjčky + vrátit (jako předpokládané datum vrácení)
2. Uživatel vybere kazety, které zákazník vrací, a výběr potvrdí
3. Systém od zákazníka odstraní všechny kazety, které předtím uživatel označil, a případ užití skončí

následné podmínky

seznam vypůjčených kazet byl u zákazníka aktualizován *zde jsou uvedeny podmínky, které po úspěšném dokončení případu užití musí být pravdivé*

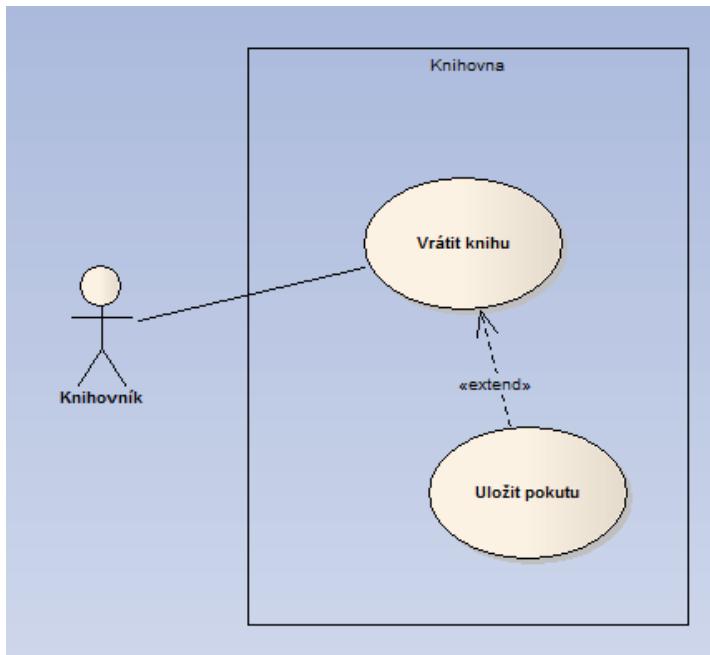
alternativní tok

1. k bodu 2. hlavního toku : pokud v seznamu vypůjčených kazet není žádná kazeta, systém tuto skutečnost oznámí a případ užití skončí *zde jsou uvedeny alternativy k hlavnímu toku, tj. průběhy nastávající jen zcela výjimečně a nestandardně*

chybový tok

1. k bodu 1. hlavního toku : pokud zákazník není nalezen, systém tuto chybu vypíše, upozorní obsluhu na to, že průkazka by měla být zničena a případ užití skončí *zde jsou uvedeny chybové větvě k hlavnímu toku, tj. průběhy, které by neměly vůbec nastat - pokud nastanou, je to považováno za chybu (např. nekonzistence dat v systému, nesoulad mezi reálným světem a evidovanými informacemi apod.) a tato situace musí být nějak vyřešena*

Příklad 2 - relace extend

**Případ užití: Vrátit knihu**

1. Knihovník zadá ID čtenáře.
2. Systém zobrazí seznam aktuálně vypůjčených knih čtenáře.
3. Knihovník vyhledá v seznamu knihu, kterou zákazník vraci.
- místo rozšíření: Uložit pokutu
4. Knihovník vraci knihu.

Případ užití: Uložit pokutu**Vstupní podmínky**

1. Vrácení knihy je opožděno

Hlavní scénář

1. Knihovník zadá podrobnosti o pokutě do systému.

2. Systém vytiskne pokutu.

Relace include vs. relace extend

* include

- vyčleňuje společné kroky několika případů užití do samostatného případu užití
- případu užití „Vrátit kazetu“ na prvním obrázku říkáme klientský
- případu užití „Najít zákazníka“ na prvním obrázku říkáme dodavatelský
- klientský případ užití je bez dodavatelského neúplný

* extend

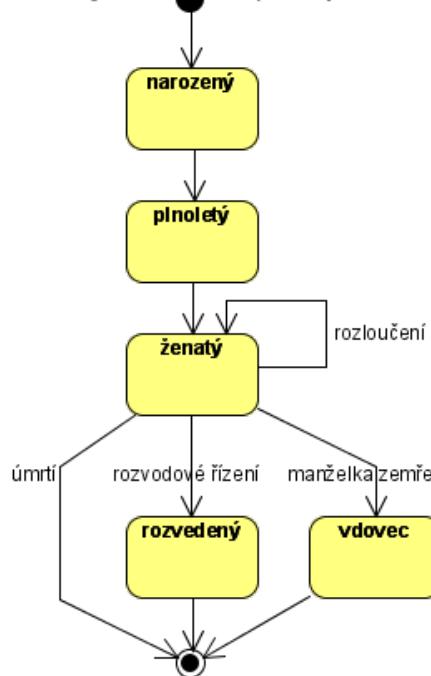
- umožňuje rozšířit případ užití o nové chování
- případu užití „Vrátit knihu“ na druhém obrázku říkáme bázový
- případu užití „Uložit pokutu“ na druhém obrázku říkáme rozšiřující
- bázový případ užití neví o rozšiřujícím; má pro něj jen tzv. místa rozšíření; je úplný i bez rozšiřujícího

- rozšiřující bývá zpravidla bez neúplný (tzn. že nemůže být přímo inicializován aktérem, tzn. nemůže k němu vést asociace přímo od aktéra)

stavový diagram

Diagram obsahuje stavový stroj (state machine). Stavový stroj vyjadřuje stavy určitého objektu a přechody mezi těmito stavy.

Visual Paradigm for UML Community Edition [not for commercial]



Na obrázku je příklad jednoduchého diagramu stavového stroje :

- **stav** (state) : situace, kdy modelovaný objekt splňuje nějakou podmínu, provádí nějakou operaci nebo čeká na událost - např. ve stavu narozený splňuje objekt podmínu, že mu není dosud 18 let a čeká, až dosáhne plnoletosti
- **přechod** (transition)
- **spojení** mezi dvěma stavy; objekt přejde z prvního stavu do druhého stavu (za splnění určitých podmínek)
- **rozloučení** ukazuje přechod do téhož stavu (něco význačného se stane, ale stav se nezmění)
- **počáteční, finální stav**: zvláštní pseudostavy pro počátek a konec automatu

Stavový stroj je graf **stavů a přechodů** (mezi těmito stavami), který popisuje reakce objektu (přesněji : reakce instance klasifikátoru) na obdržení události. Stavový stroj může být připojený ke klasifikátoru (např. use case, class), nebo ke collaboration či k metodě.

Element, ke kterému je připojen stavový stroj, se nazývá **vlastník (owner) stavového stroje**.

Celý stavový stroj je vlastně **složený stav** (composite state), který je rekurenci dekomponován na **substavy**. Nejspodnější (listové) stav je již nemají substavy.

Stavový stroj může mít reference na jiný stavový stroj použitím stavového substroje (submachine state). Ve stavovém stroji může být v jeden okamžik aktivní více stavů.

Pro vysvětlování podrobností o stavech musíme vědět, **co je to přechod**:

přechod (transition) je relace ve stavovém stroji mezi dvěma stavy, kde objekt v prvním stavu přejde do druhého stavu tehdy, když nastane specifikovaná událost (event) a jsou splněny specifikované podmínky (guards), přičemž se provede specifikovaný efekt (effect - action nebo activity). Přechod je nepřerušitelný.

Říká se, že přechod je odpálen (transition is fire). Přechod může mít jeden nebo více zdrojových stavů a jeden nebo více

cílových stavů.

Co je to stav

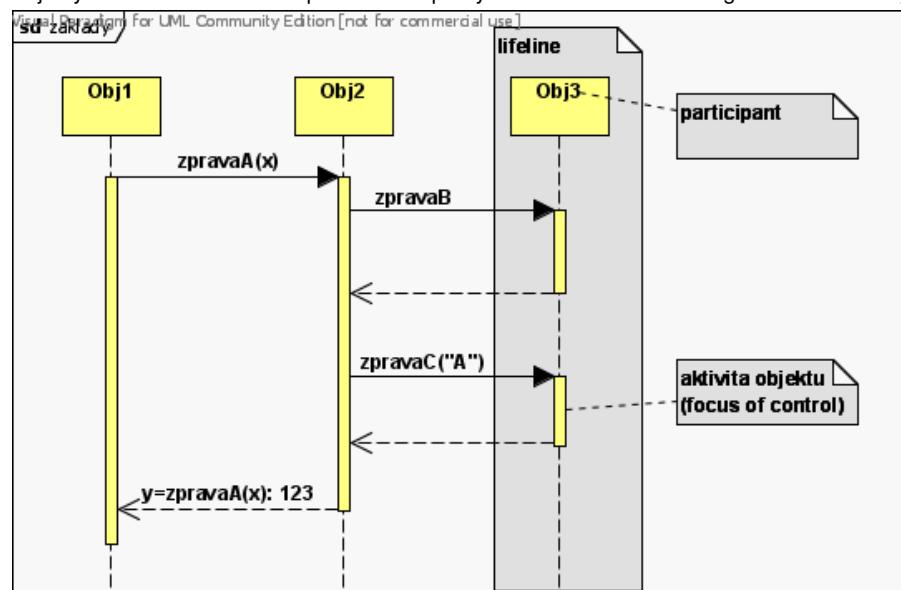
stav (state) : situace, kdy modelovaný objekt splňuje nějakou podmínu, provádí nějakou operaci nebo čeká na událost. - např. ve stavu narozený splňuje objekt podmínu, že mu není dosud 18 roků a čeká, až dosáhne plnoletosti. Stavy jsou obsaženy ve stavovém stroji, který popisuje jak se vyvíjí objekt v čase dle svých reakcí na události.

sekvenční diagram

Přitom sekvenční d. a d. komunikací jsou téměř izomorfní - tj. dají se převádět z jednoho tvaru na druhý (často i automatizovaně) (to však platí jen pro jednoduché sekvenční diagramy, které nepoužívají strukturované mechanismy jako interaction use a combined fragment - vysvětleny níže v sekci další podrobnosti).

Použití sekvenčního diagramu bývá vhodnější v těch případech, kde jsou důležité časové souvislosti interakcí, ovšem nevidíme v něm zobrazené vztahy mezi objekty.

Objekty si mohou posílat zprávy. Sekvenční diagram zobrazuje jejich časovou posloupnost.



Jedná se o jednoduchý sekvenční diagram. Časová osa je svislá (čas běží zhora dolů), na vodorovné ose jsou rozmištěny objekty:

- lifeline (čára života) reprezentuje participantu v interakci:
 - **Participant** je většinou konkrétní objekt.
 - Lifeline také ukazuje, kdy participant žije - v tomto diagramu všechny objekty existovaly už před posláním první zprávy, a dále existují po dokončení sekvenčního diagramu (resp. nevíme, kdy který objekt zanikne nebo nás to v tuto chvíli nezajímá)
 - **execution specification, focus of control** (aktivita objektu) vyjádřená zdvojeným úsekem (obdélníkem) na lifeline : ukazuje periodu, kdy je který objekt aktivní, neboli kdy provádí nějakou činnost, nějak se chová, včetně „podřízeného“ chování dalších objektů (tj. u aktivního objektu vyznačuje celý jeho život, u pasivního objektu časový interval, kdy je prováděna operace objektu, včetně čekání na návrat ze zavolené operace; v programátorském světě je to analogické době, po kterou je určitá hodnota ve „stack“ - v zásobníku))
- zpráva:
 - v našem příkladě máme jen jednoduché obyčejné zprávy : u nich se zpráva zobrazuje plnou čárou a plnou šipkou a v našem příkladu je znázorněno např. ohledně zprávy **zpravaA(x)** : objekt **obj1** posílá (tedy je to sender) zprávu **zpravaA** s jedním argumentem, který je naplněn hodnotou atributu **x**, objektu **obj2** (to je tedy receiver)
 - zobrazení návratové zprávy není povinné (vždy si ji lze „domyslet“), ale může přidat na přehlednosti: zobrazuje se

šipkou s čárkovnou čárou

- návratová zpráva může, ale nemusí vracet hodnotu : po poslání zpravaA(x) a po té, co obj2 dokončí svou činnost, tak vrátí zpět návratovou hodnotu 123, kterou si obj1 převezme do svého atributu y

V diagramu je znázorněna tato sekvence :

1. objekt obj1 je hned na počátku aktivován (má focus) a poslal zprávu **zpravaA** s parametrem **x** objektu **obj2** : obj1 přeruší zpracování (ztratí focus a předá řízení obj2) a čeká, až obj2 odpoví na zpravaA(x)
2. **obj2** získává focus a spustí vlastní metodu
3. v jistém bodě zpracování posílá obj2 zprávu **zpravaB** do **obj3** - obj2 započne čekat na odpověď od obj3
4. **obj3** získává focus (a spouští vlastní metodu)
5. **obj3** dokončí své zpracování, pak vrátí peška (předá řízení zpět) do obj2
6. **obj2** pokračuje v práci, a posílá další zrávu do obj3
7. až obj3 dokončí zpracování, tak vrátí řízení zpět do obj2
8. **obj2** pokračuje v práci a až skončí, tak vrátí řízení do **obj1**, přitom ale vrátí návratový návratovou hodnotu 123
9. **obj1** si převezme návratovou hodnotu do atributu **y**

Popis šípek:

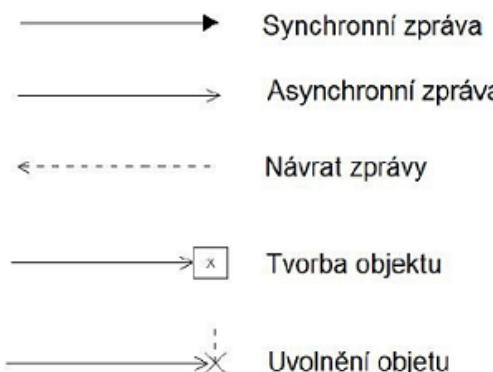


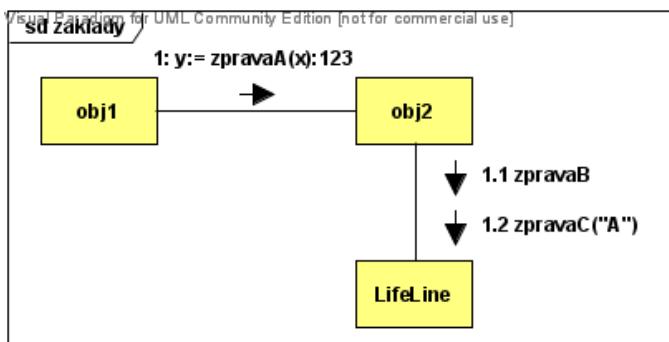
diagram komunikace

Diagram komunikací je i není nový v UML 2.0 : totiž v UML 1.x se jmenoval **collaboration diagram**, což bylo matoucí (collaboration byly statické struktury, kdežto diagram zahrnoval zprávy).

D. komunikací a sekvenční d. jsou témař **izomorfni** - tj. dají se převádět z jednoho tvaru na druhý (často i automatizovaně) (to však platí jen pro jednoduché sekvenční diagramy, které nepoužívají strukturované mechanismy jako interaction use a combined fragment). D. komunikaci i sekvenční d. ukazují interakce, ale každý svým vlastním způsobem.

Použití diagramu komunikací bývá vhodnější v těch případech, kde chceme zdůraznit strukturální aspekty spolupráce, tj. ukázat hlavně **kdo s kým komunikuje** - jsou méně vhodné pro zdůraznění časových souvislostí interakcí.

Objekty si mohou posílat zprávy. Diagram komunikací ukazuje objekty (přesněji : části kompozitní struktury nebo role ve spolupráci(collaboration)) a spojení a zprávy, které si objekty posílají. Čas zde nevystupuje jako zvláštní dimenze, proto musí být sekvence zpráv a spouběžnost threadů určena pomocí čísel sekvenční :



Jedná se o diagram komunikací odvozený z našeho jednoduchého sekvenčního diagramu

- uzly v tomto diagramu reprezentují části strukturované třídy nebo role collaborations (dále pro zjednodušení je budeme nazývat jako objekty) a korespondují s lifeline v sekvenčním diagramu.
- komunikační cesty (paths) jsou vyjádřeny čárami (spojkami, connectors) mezi uzly. Čáry můžou být nazvány svým jménem a/nebo jménem výchozí asociace (existuje-li). Můžou být vyjádřeny multiplicity.
- zpráva:
 - zpráva je zobrazena jako malá pojmenovaná šipka umístěná blízko spojnic; v našem příkladě máme jen jednoduché obyčejné zprávy : zde je šipka plná a v příkladu je znázorněno např. ohledně zprávy **zpravaA(x)** : **objekt obj1 posílá** (tedy je to sender) zprávu zpravaA s jedním argumentem, který je je naplněn hodnotou atributu x, objektu **obj2** (to je tedy **receiver**), ten po svém zpracování vrací hodnotu 123, kterou si obj1 převezme do atributu y
 - **sekvenční výraz** (umístěný před názvem zprávy) určuje pořadí, v jakém jsou zprávy posílány : pozor, není to jen prosté pořadové číslo, ale je v něm vyjádřeno i zanoření
 - návratováHodnota není povinná

V diagramu je znázorněna tato sekvence :

1. objekt **obj1** je hned na počátku aktivován (má focus) a posal zprávu **zpravaA** s parametrem x objektu **obj2 : obj1** přeruší zpracování (ztratí focus a předá řízení **obj2**) a čeká, až obj2 odpoví na zpravaA
2. **obj2** získává focus a spustí vlastní metodu
3. v jistém bodě zpracování posílá obj2 zprávu **zpravaB** do **obj3** - obj2 započne čekat na odpověď od obj3
4. **obj3** získává focus (a spouští vlastní metodu)
5. **obj3** dokončí své zpracování, pak vrátí pešku (předá řízení zpět) do obj2
6. **obj2** pokračuje v práci, a posílá další zrávu do obj3
7. až **obj3** dokončí zpracování, tak vrátí řízení zpět do obj2
8. **obj2** pokračuje v práci a až skončí, tak vrátí řízení do obj1, přitom ale vrátí návratovou hodnotu 123
9. **obj1** si převezme návratovou hodnotu do atributu y

D. komunikací vlastně obsahuje jen čtyři typy elementů : základní frame, lifeline, komunikační cesty (paths) a zprávy

Stručný diagram interakce

Diagram přehledu interakcí definuje interakce pomocí varianty diagramu aktivit : tato varianta zahrnuje fragmenty sekvenčních diagramů spolu s konstrukty pro řízení toku.

Visual Paradigm for UML Community Edition [not for commercial use]

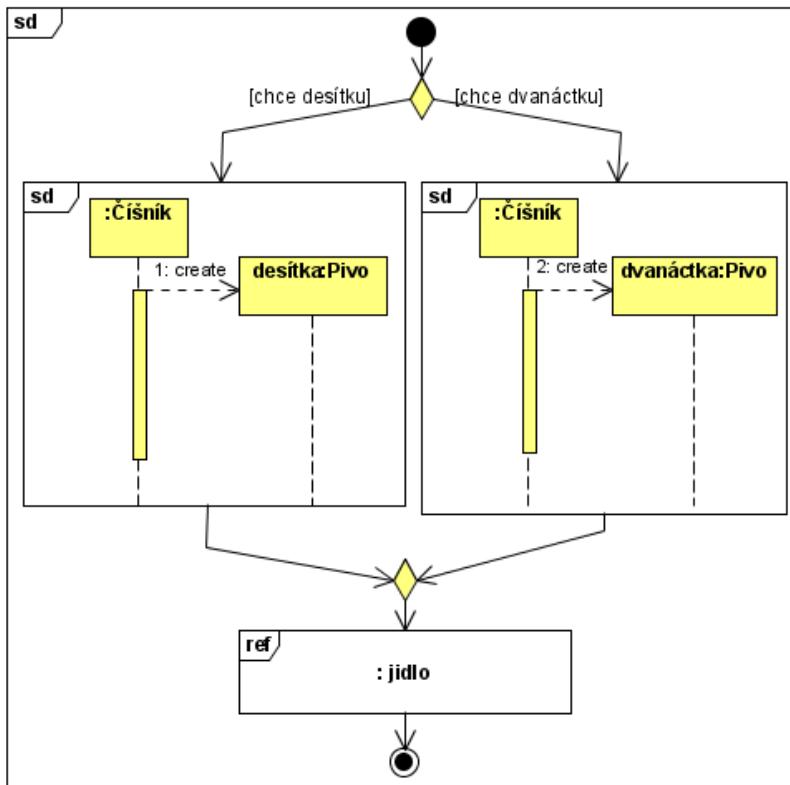
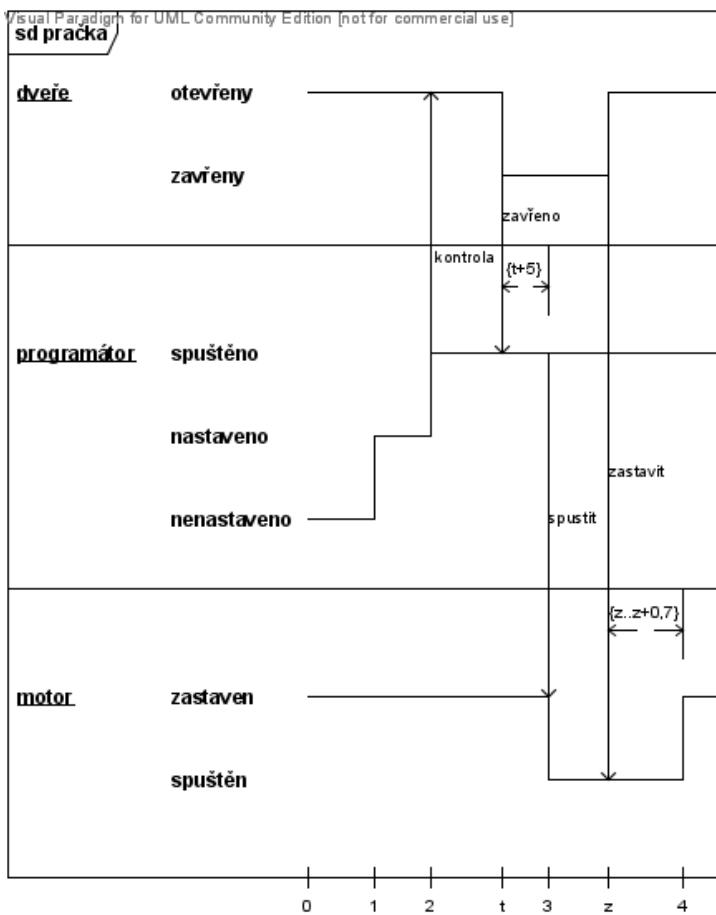


Diagram přehledu interakcí obsahuje jako své uzly interakce (online vložené interakční diagramy) a interaction use (ad interaction use viz sekvenční diagram)

- frame (rám): význam a jmenovka jsou stejné, jako v sekvenčním diagramu. Ovšem v refrenčním manuálu k UML se jeho autoři (trio amigos) odchylují od specifikace UML (sami to přiznávají), a používají jmenovku *interover* namísto sd
- alternative combined fragment zde má obdobu v uzlu decision (rozhodnutí) a odpovídajícím uzlu merge (splaynění)
- parallel combined fragment zde má obdobu v uzlu fork (rozdvojení) a odpovídajícím uzlu join (spojení)
- větvení a spojování zde musí být (na rozdíl od d. aktivit) explicitně uvedeno

diagram časování

Diagram časování se používá k zobrazení iterakcí, kde hlavním záměrem je zobrazit podrobnosti o času. Je to alternativní forma sekvenčního diagramu (sequence diagram), která **explicitně zobrazuje změny stavu lifeline v čase** (v určitých časových jednotkách). Může být užitečný pro modelování real-time aplikací. Tento diagram bude patrně velmi povědomý všem, kdo se zabývají časováním integrovaných obvodů.



Jedná se o jednoduchý diagram časování pro řízení motoru pračky. Časová osa je vodorovná (čas běží zleva doprava), na svislé ose jsou rozmístěny jednotlivé **lifelines**, tj. **dvere** (pračky), **programátor** (pro výběr programů pračky a samotné spuštění programu) a motor (ad lifeline : viz sekvenční diagram (sequence diagram)) U každé lifeline jsou uvedeny jednotlivé stavy (např. dveře jsou **otevřeny** nebo **zavřeny**) či jednotlivé testovatelné podmínky; pozice názvů těchto stavů pak určují pozice, na které přeskakuje čára lifeline pro zobrazení přechodů mezi stavů. V diagramu je znázorněna tato sekvence :

1. na začátku sekvence jsou **dvere** ve stavu **otevřeny**, **programátor** ve stavu **nenastaveno** a **motor** ve stavu **zastaven**
2. v čase označeném **osovou značkou 1** uživatel nastavil programátor pračky (tj. nastala změna stavu programátoru na **nastaveno**), což nemá vliv ani na dveře, ani na motor
3. v čase označeném **osovou značkou 2** uživatel spustil vykonávání nastaveného programu (např. zmáčknul tlačítko start), a programátor poslal **zprávu (message)** **kontrola** dveřím (pro provedení kontroly, zda jsou dveře zavřeny a lze tedy spustit prací cyklus dle spuštěného programu) a začal čekat na odpověď od dveří; motor pračky je stále zastaven
4. v čase označeném značkou **t** uživatel zavřel dveře, tj. programátor se dozvěděl (pomocí zprávy **zavřeno**), že může spustit prací cyklus; motor pračky je však stále zastaven, neboť:
5. mezi značkami **t** a **3** je uveden time constraint $\{t+5\}$ značící, že mezi příjemem zprávy **zavřeno** (tj. mezi okamžikem označeným značkou **t**) a posláním zprávy **spustit** k motoru (tj. okamžikem označeným značkou **3**) se musí počkat 5 sekund (pro jistotu : uživatel mohl chtít po zavření dveří ještě přaprogramovat, otevřít dvířka pro doplnění prádla apod. - tyto zvláštní varianty průběhu sekvence zde však nemáme zakresleny)
6. v čase označeném **3** tedy **programátor** poslal motoru zprávu **spustit** a motor se začne točit : jeho stav se změní na **spuštěn**
7. v nějakém (blíže nespecifikovaném) okamžiku může uživatel otevřít dveře : v tomto čase označeném značkou z tedy dveře poslaly motoru zprávu **zastavit**
8. v čase označeném **4** se motor zastaví, tedy přejde do stavu **zastaven**; přitom mezi značkami **z** a **4** je uveden **time constraint** $\{z..z+0,7\}$ značící, že mezi příjemem zprávy **zastavit** (tj. mezi okamžikem označeným značkou **z**) a přechodem

stavu motoru do zastaven (tj. okamžikem označeným značkou 4) musí uběhnout nejvýše 0,7 s dlouhá prodleva (pro ochranu uživatele : aby nesáhl do bubnu práčky, když se ještě točí - předpokládá se ovšem, že tam nestihne střít ruce do 0,7 s 😊)

Diagram časování také obsahuje základní frame (rám) : význam a jmenovka jsou stejné, jako v sekvenčním diagramu.

Odlišnosti od klasické formy sekvenčního diagramu :

- osy jsou obyčejně naopak : čas plyne zleva doprava
- lifelines jsou ve zvláštních kompartmentech, řazené svisle
- lifeline posakuje nahoru a dolů, čímž se zobrazují změny stavu; pořadí stavů může nebo nemusí být důležité; je možná i alternativní forma, kdy lifeline osciluje kolem vodorovné přímky a změny stavů jsou vyznačeny přímo na ní (viz diagram níže)
- může být měřící časová osa; **osové značky** pak indikují časové intervaly, někdy přímo diskrétní okamžiky, ve kterých nastávají změny
- časy jsou synchronizovány pro všechny lifelines dohromady
- může být zobrazena hodnota objektu

OCL

* Object Constraint Language

* jazyk, který umožňuje doplňovat UML o dodatečné informace

* **standardní rozšíření jazyka UML, které poskytuje operace:**

- upřesňování omezujících pravidel
- upřesňování provozních pravidel během modelování
- upřesňování těl vyhledávacích dotazů

* **důvody použití**

- umožnění modelovacím nástrojům vhodné generování kódu
- umožňuje přesnější modelování (výsledkem bude méně nesprávných výkladů modelů)
- umožňuje modelovacím nástrojům „přemýšlet o modelech UML“ (což např zahrnuje ověřování konzistence modelů)

* **nevýhody**

- obtížná čitelnost, nepravidelná syntaxe
- není příliš rozšířený, zná ho jen málo lidí
- někdy přesnost, kterou OCL nabízí nemusí být potřeba

* **výrazy jazyka OCL**

- připojují se k modelovaným prvkům UML
- skládají se z
 - kontextu balíčku (nepovinný): definuje jmenný prostor pro výrazy v jazyce OCL
 - kontext výrazu (povinný): definuje kontextovou instanci výrazu
 - jednoho nebo více výrazů
- kontext výrazů lze definovat např. vložením výrazu OCL do poznámky, která se připojí k modelovanému prvku

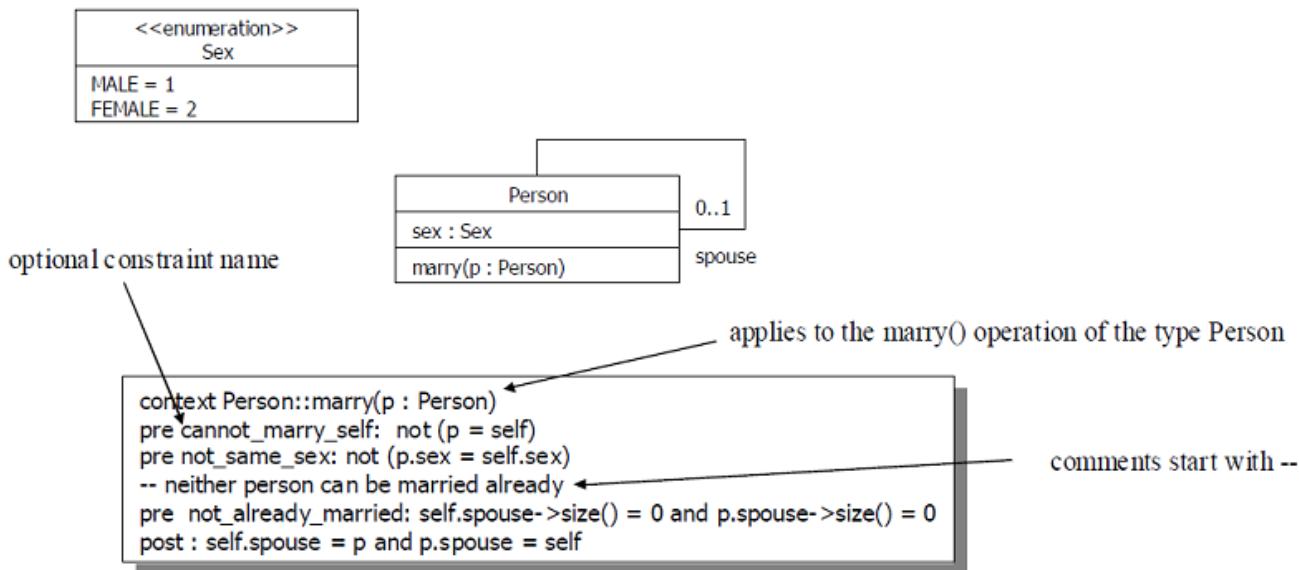
* **příklady použití**

- v diagramech iterace se používá k určování
 - kontrolní podmínky
 - selektorů pro čáry života
 - argumentů zpráv
- v diagramech aktivit se používá k určování

- akčních uzlů volání
- kontrolních podmínek pro přechody
- objektových uzlů
- stavů objektů
- ve stavových diagramech se používá k určování
 - kontrolních podmínek
 - podmínek stavů
 - cílů akcí
 - hodnot atributů

* příklad

- osoba si nesmí vzít sama sebe
- osoba si nesmí vzít osobu, která má stejné pohlaví
- osoba si nesmí vzít osobu, která je již ženatá/vdaná
- po sňatku musí mít self za manžela/manželku osobu p && osoba p musí po sňatku za manželku/manžela osobu self



si/si2.txt · Poslední úprava: 2009/06/09 12:45 autor: D M

Otázka 03 - Y36DBS

Zadání: Návrh relačního schématu. Normalizace schématu formou dekompozice. Kritéria kvality dekompozice. Návrh schématu relační databáze přímou transformací z konceptuálního schématu

Slovniček pojmu

- **relace** - množina prvků, též množina n-tic
- **atribut** - jeden prvek z dané relace
- **doména** - množina hodnot, kterých může atribut nabývat
- **stupeň relace** - je počet atributů relace
- **relační schéma** - výraz tvaru $R(A, f)$, kde R je jméno schématu, $A = \{A_1, A_2, \dots, A_n\}$ je konečná množina jmen atributů, f je zobrazení přiřazující každému jménu atributu A_i neprázdnou množinu (obor hodnot atributu), kterou nazýváme doménou atributu D_i , tedy $f(A_i) = D_i$. Často se tímto pojmem rozumí název relace a v závorce uvedené její atributy, tedy například $\text{Kino}(Název, Adresa, Rok_založení)$.
- **relační model** - sdružení dat do tzv. relací (tabulek), které obsahují n-tice (řádky). Tabulky (relace) tvoří základ relační databáze. Tabulka je struktura záznamů s pevně stanovenými položkami (sloupci - atributy). Každý sloupec má definován jednoznačný název, typ a rozsah, neboli doménu. Záznam se stává n-ticí (řádkem) tabulky. Pokud jsou v různých tabulkách sloupce stejného typu, pak tyto sloupce mohou vytvářet vazby mezi jednotlivými tabulkami. Tabulky se poté naplňují vlastním obsahem - konkrétními daty.
- **relační databáze** - databáze založená na relačním modelu
- **tabulka** - reprezentace instance relačního schématu
- **primární klíč** - sloupec (nebo sloupce), který jednoznačně určuje řádky v tabulce
- **cizí klíč** - slouží pro vyjádření vztahů, relací, mezi databázovými tabulkami. Jedná se o pole či skupinu polí, která nám umožní identifikovat, které záznamy z různých tabulek spolu navzájem souvisí.
- **normalizace** - proces rozkladu údajů do množin dat, která jsou spojeny pomocí společného prvku (jinak: Normalizace je proces rozhodování jaký sloupec umístíme v které tabulce.)
- **funkční závislost** - sloupec A je funkčně závislý na B právě tehdy když, pro každou hodnotu ve sloupci A existuje nejvýše jedna hodnota ve sloupci B .
- **konceptuální model databáze**
 - databázový model umožňující zobrazit a popsat objekty v databázi a vztahy mezi nimi z hlediska jejich významu a chování
 - výsledkem konceptuálního modelování je implementačně nezávislé databázové schéma, tj. schéma obecně aplikovatelné v jakémkoli technicko-programovém prostředí.

Návrh relačního schématu

Při návrhu relačního schématu se obvykle používá transformace z konceptuálního schématu.

Konceptuální (někdy také sémantické) modely jsou pokusem umožnit vytvoření popisu dat v databázi nezávisle na jejich uložení. Konceptuální model představuje formální popis modelované reality. Hlavními úkoly je nalezení entit, vztahů a atributů. Slouží obvykle k vytvoření schémat s následnou transformací na databázové schéma. Spojíme-li sémantickou a databázovou úroveň, dostáváme se k tzv. objektově-orientovaným SŘBD. Konceptuální modely používají pojmy:

entita (objekt) - student, předmět

vztah (relationship) - studuje

atributy (vlastnost) - věk, rč

Činnosti při tvorbě E-R modelu:

E-R model je množina pojmu, které nám pomáhají, na konceptuální úrovni abstrakce popsat uživatelskou aplikaci za účelem specifikovat následně strukturu databáze. Každá entita musí být jednoznačně identifikovatelná. Atribut (skupina atributů), jehož hodnota slouží k identifikaci konkrétní entity se nazývá identifikačním klíčem.

1. **identifikace typů entit** jako množiny objektů stejného typu. Např. KNIHA, ABONENT_KNIHOVNY, ZAMESTNANEC označují typy entit.
2. **identifikace typů vztahů**, do kterých entity identifikovaných typů mohou vstupovat. Např. ABONENT(entita) MA_PUJCEN (vztah) daný EXEMPLAR (entita).
3. na základě přiměřené úrovně abstrakce **přiřazení jednotlivým typům entit a vztahů popisné atributy**. Např. PRIJMENI (popisný atribut) daného ZAMESTNANCE (entita), DATUM (popisný atribut), do kdy si daný ABONENT (entita) VYPUJCIL (údaj typu vztah) daný EXEMPLAR (entita).
4. **formulace integritních omezení** (IO) vyjadřujících s větší či menší přesností soulad schématu s modelovanou realitou.

Entita je objekt reálného světa, který je schopen nezávislé existence a je jednoznačně odlišitelný od ostatních objektů. Vztah je vazba mezi dvěma entitami (obecně i více entitami). Hodnota popisného typu popisným typem budeme rozumět jednoduchý datový typ. Atributem budeme rozumět funkci přiřazující entitám či vztahům hodnotu popisného typu, určující některou podstatnou vlastnost entity nebo vztahu.

Příklad:

lineární zápis:

E: Zaměstnanec, Oddělení

R: Je zaměstnán na(Zaměstnanec, Oddělení)



Do E-R modelu dále značíme:

Kardinalita vztahů = násobnost účasti ve vztahu (1:1, 1:N, M:N)

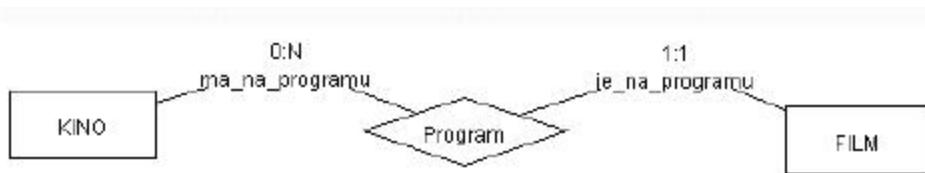
Parcialita = povinnost účasti ve vztahu:

povinná účast = všechny výskyty účastníka musí být zapojeny do příslušného vztahu

nepovinná účast = jednotlivé výskyty členské entity mohou být zapojeny do vztahu daného typu

Příklad:

Kino může mnohokrát. Film musí právě jednou.



Slabé entitní typy

Součástí klíče některých entitních typů nemusí pouze být jejich vlastní atributy. V takovém případě nemusíme být schopni rozlišit mezi dvěma instancemi jednoho entitního typu na základě hodnot jeho vlastních atributů. V takovém případě mluvíme o slabém entitním typu. Jeho různé instance jsou identifikovatelné tím, že jsou v povinném vztahu k instanci entity jiného typu. Tento druhý entitní typ nazýváme identifikační vlastník a vztahu, který je spojuje identifikační vztah. Slabý entitní typ má vždy povinné členství ve vztahu k tzv. identifikačnímu vlastníku (jedná se tedy o existenční závislost). Opačné tvrzení neplatí. Nelze o každé exisťní závislosti mluvit jako o závislosti identifikační. Tedy existenčně závislá entita ještě není slabá entita.

ISA hierarchie, podtypy entit

Speciální atributy představují v abstraktním modelování takové atributy, které danému typu entity přřazují jeho nadtyp. Atribut je pak podtypem svého nadtypu. Jde o tzv. ISA-hierarchii.

Transformace ER modelu na relační schéma

Z ER modelu tedy můžeme podle několika pravidel vytvořit relační schéma. Tento proces bude podrobněji popsán v poslední části otázky. Jde především o tyto transformace:

1. entita se transformuje na tabulku
2. Každý atribut entity se změní na sloupec tabulky
3. Primární klíč entity bude primárním klíčem tabulky
4. provedou se potřebné úpravy, aby bylo relační schéma validní.

Relační model

Relační databázový model důsledně **odděluje data**, která jsou chápána jako relace, **od jejich implementace**. Přístup k datům je symetrický, tj. při manipulaci s daty se nezajímáme o přístupové mechanizmy k datům. Pro manipulaci s daty jsou k dispozici dva silné prostředky - **relační kalkul** a **relační algebra**. Pro omezení redundance dat v relační databázi jsou navrženy pojmy umožňující normalizovat relace. Od matematické relace se relační model liší v několika aspektech:

- relace je vybavena pomocnou strukturou, které se říká schéma relace. Schéma relace se skládá ze jména relace, jmen atributů a domén
- prvky domén, ze kterých se berou jednotlivé komponenty prvků relace, jsou atomické (dále nedělitelné) hodnoty.

Relační model dat se tedy skládá z těchto částí, které definují relaci:

- jména atributů
- domény atributů
- n-tice atributů
- relace
- schémata relací
- jména schémat relací

Inuitivně (pracovně), ale nepřesně: relace = tabulka, schéma = záhlaví tabulky.

Příklad tabulka kino:

název	adresa
Blaník	Václavské nám.
Mír	Strašnická
Domovina	V Dvorcích

Schéma relací: Kino (název, adresa)

Relační algebra

Relační algebra je nejzákladnějším prostředkem pro práci s relacemi. Jedná se o dotazovací jazyk mezi jehož základní operace patří projekce, selekce a spojení.

- **Projekce** relace R s položkami A na množinu položek B vytvoří relaci s položkami B a záznamy, které vzniknou z původní tabulky R odstraněním položek A-B. Odstraněny jsou i eventuelně opakující se záznamy. Značí se: **R[B]** (laicky: jde o výcuc určitých sloupců/položek B z A)
- **Selekce** relace R s položkami A podle logické podmínky F vytvoří tabulku s týmiž položkami a ponechá ty záznamy z původní tabulky, které splňují logickou podmínku F. Značí se: **R(Q)** (laicky: jde o výcuc určitých řádek z A, které splňují podmínku F)
- **Spojení** relací R a S s položkami A a B vytvoří minimalizovanou tabulku se záznamy, jejichž projekce na A je z tabulky R a projekce na B je z tabulky S. Značí se: **R * S** (laicky: jde o spojení tabulek R a S podle určitého sloupce)

Relační kalkul

Relační kalkul je dotazovací jazyk, který vychází z predikátové logiky 1.řádu a v relačních databázích se vyskytuje ve dvou formách. Jedná se o n-ticový (řádkový) a doménový relační kalkul. Doménový relační kalkul oproti řádkovému pracuje s proměnnými, které nemají za hodnoty n-tice, ale jednotlivé prvky z domén, tj. jednoduché hodnoty atributů.

Integritní omezení

Je nutné zajistit, aby se do relací dostala pouze „správná“ data - přípustné n-tice. Úplná definice relačního schématu je:

(R,I) ... schéma relační databáze

R = { R1 ,R2 ,..., Rk}

I ... množina integritních omezení

Přípustná relační databáze se schématem (R, I) je množina relací $R_1^*, R_2^*, \dots, R_k^*$ takových, že jejich n -tice vyhovují tvrzením v I . Integrálním omezením jsou např. klíče.

Příklad:

$KINO(NÁZEV_K, ADRESA)$,
 $FILM(JMÉNO_F, HEREC, ROK)$
 $MÁ_NA_PROGRAMU(NÁZEV_K, JMÉNO_F, DATUM)$

Integrální omezení:

IO1: *primární klíč*

IO2: *Cizí klíč*

IO3: V kinech se nehráje více, než dvakrát týdně

IO4: Jeden film se nedává více, než ve třech kinech

Podmínky, které musí splňovat relační tabulka:

- všechny hodnoty v tabulce musí být elementární - tzv. Dále nedělitelné - podmínka 1.NF
- sloupce mohou být v libovolném pořadí
- řádky mohou být v libovolném pořadí
- sloupce musí být homogenní = ve sloupci musí být údaje stejného typu
- každému sloupci musí být přiřazeno jednoznačné jméno (tzv. atribut)
- v relační tabulce nesmí být dva zcela stejné řádky. Tzn., že každý řádek je jednoznačně rozlišitelný.

Normalizace schématu formou dekompozice

- předpokládejme, že máme relaci, která není v požadované normální formě
- je potřeba tuto relaci restrukturalizovat na množinu normalizovaných relací
- tato restrukturalizace obvykle zahrnuje rozdelení původní relace do několika menších normalizovaných relací
- tento proces je nazýván **dekompozice**
- dekompozici je potřeba provádět pečlivě:
 - dekompozice by neměla způsobit ztrátu informace
 - mělo by být možné zkontrolovat integrální omezení v dekomponované verzi stejně snadno jako v původní verzi

Příklad: Dostaneme relaci $REZERVACE = (ID_NAMORNIKA, JMENO_NAMORNIKA, ID_LODE, DEN)$

Víme, že existuje funkční závislost $ID_NAMORNIKA \rightarrow JMENO_NAMORNIKA$. Je tato relace ve 3 normální formě?

Není, protože existuje tranzitivní závislost způsobená závislostí $ID_NAMORNIKA \rightarrow JMENO_NAMORNIKA$.

Předpokládejme, že bychom původní relaci $REZERVACE$ dekomponovali do relací $R1 =$

$(ID_{NAMORNIKA}, JMENO_{NAMORNIKA})$ a $R2 = (ID_{LODE}, DEN)$.

Jsou tyto relace ve 3. normální formě?

Ano (všechny neklíčové atributy jsou závislé jen a jen na klíci, nejsou závislé vzájemně; předtím to neplatilo, protože ID_{LODE} a DEN jsou tranzitivně závislé na $ID_{NAMORNIKA}$ - přes $JMENO_{NAMORNIKA}$), ale nyní nezjistíme, kdo si zarezervoval kterou lodě... došlo ke **ztrátě informace**.

Pozn.: v příkladu se patrně předpokládá, že jméno námořníka je jedinečné a tedy také jedninečně určuje id lodě.

Vlastnost bezztrátového spojení

Definice: Pokud je relace R dekomponovaná do dvou částí X a Y takových, že $X \subseteq R$ a $Y \subseteq R$, dekompozice má vlastnost bezztrátového spojení, pokud pro každou platnou instanci r relace R platí: $\pi_X(r) \bowtie \pi_Y(r) = r$.

- jinými slovy, dekompozice má vlastnost bezztrátového spojení pouze v případě, že jsme schopni zpěně rekonstruovat původní relaci prostřednictvím operace join
- všimněte si, že instance r musí splňovat všechny funkční závislosti, které platí pro relaci R

Poznámka: Ve výše uvedené definici se vyskytuje vzorec, který si jistě zaslouží drobný komentář. Symbolem π s dolním indexem X označujeme instanci, která vznikla z původní instance projekcí na množinu atributů relace X (při této operaci může dojít k odstranění duplicitních řádků). Analogicky toto platí pro druhé π s indexem Y . Symbol mezi těmito znaky je operátor označující přirozené spojení. Celá definice vlastně řeší, co se stane, když znova spojíme nově vzniklé instance přirozeným spojením (přes společné atributy). Pokud je spojení bezztrátové, získáme původní instanci r . Pokud byla dekompozice provedena špatně, můžeme získat více nebo méně záznamů (v obou případech se jedná o jev, způsobený ztrátou informace - i když se zdá, že máme řádky navíc, jsou to řádky, o kterých nemáme dostatečnou informaci).

Příklad: Mějme nějakou instanci r z R (reprezentovanou tabulkou):

ID_NAMORNIKA	JMENO_NAMORNIKA	ID_LODI	DEN
101	Lubby	103	03/05/96
101	Lubby	102	06/05/96
102	Dennis	102	17/05/96
103	Rusty	104	13/05/96

Jak můžeme relaci R dekomponovat do dvou relací, které budou ve 3. normální formě a neztratit přitom žádnou informaci?

Provedeme dekompozici relace R na relace:

- $R_1 = (ID_{NAMORNIKA}, JMENO_{NAMORNIKA})$
- $R_2 = (ID_{NAMORNIKA}, ID_{LODI}, DEN)$

Pro instanci r získáme pro každou z nově vzniklých relací R_1, R_2 po jedné nové instanci:

Instance $\pi_{R_1}(r)$:

ID_NAMORNIKA	JMENO_NAMORNIKA
101	Lubby
102	Dennis
103	Rusty

Instance $\pi_{R_2}(r)$:

ID_NAMORNIKA	ID_LODE	DEN
101	103	03/05/96
102	102	06/05/96
102	102	17/05/96
103	104	13/05/96

Spojením $\pi_{R_1}(r) \bowtie \pi_{R_2}(r)$ získáme zpět instanci r .

Je zřejmé, že takto to bude fungovat pro libovolnou instanci r relace R .

Věta: Nechť F je množina funkčních závislostí platných pro relaci R . Dekompozice relace R na relace s množinami atributů $R_1 \subseteq R$ a $R_2 \subseteq R$ má vlastnost bezztrátového spojení právě tehdy když uzávěr funkčních závislostí F^+ zahrnuje alespoň jednu z následujících funkčních závislostí:

- $R_1 \cap R_2 \rightarrow R_1$,
- $R_1 \cap R_2 \rightarrow R_2$.

Jinými slovy, atributy společné v R_1 a R_2 musí být klíčem buď v R_1 nebo R_2 .

Příklad: V předchozím příkladě je $R_1 \cap R_2 = \{\text{ID_NAMORNIKA}\}$.

Existuje funkční závislost $\text{ID_NAMORNIKA} \rightarrow \{\text{ID_NAMORNIKA}, \text{JMENO_NAMORNIKA}\}$ a současně $R_2 = (\text{ID_NAMORNIKA}, \text{JMENO_NAMORNIKA})$.

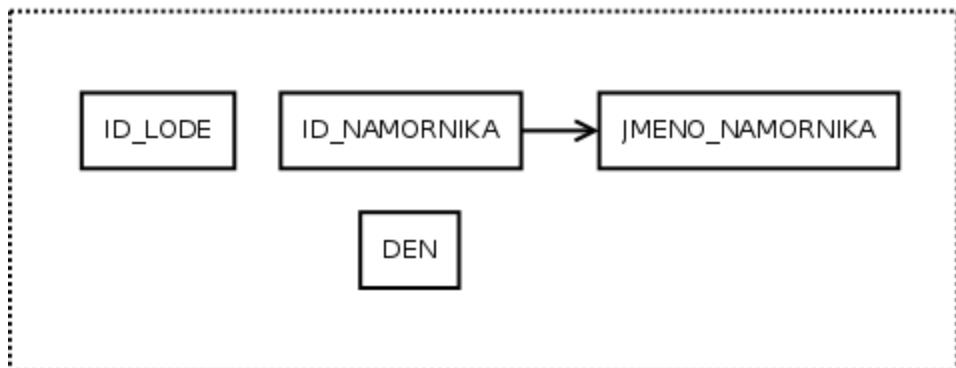
Platí tedy výše uvedná věta (všimněme si, že $R_1 \cap R_2 \rightarrow R_2$) a dekompozice má tedy vlastnost bezztrátového spojení.

Poznámka: Proč platí předchozí věta? Jednoduše řečeno, podmínky věty zajišťují, že atributy účastnící se v přirozeném spojení ($R_1 \cap R_2$) jsou kandidátním klíčem pro alespoň jednu z uvedených dvou relací. Tím je zajištěno, že se nikdy nemůžeme dostat do situace, kdy by se nám vygenerovaly „falešné“ n-tice, protože pro každou hodnotu atributů přes které se provádí spojení, zde bude jedinečná n-tice v alespoň jedné z relací.

Grafické znázornění dekompozice

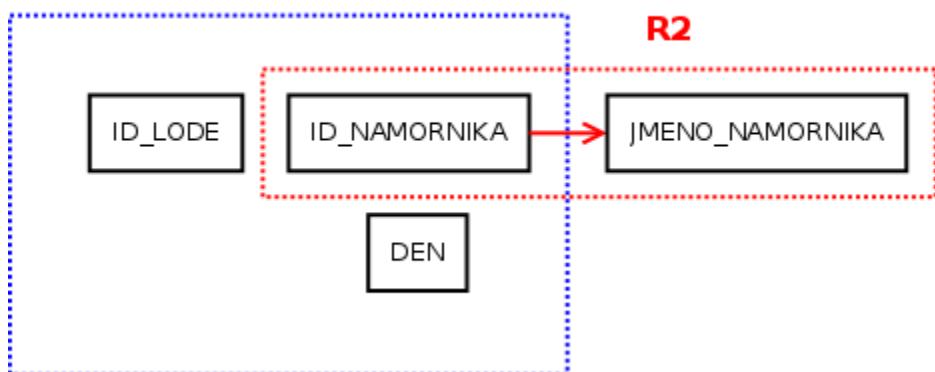
Výše zmíněný příklad můžeme graficky znázornit. Situace před dekompozicí vypadala takto:

R



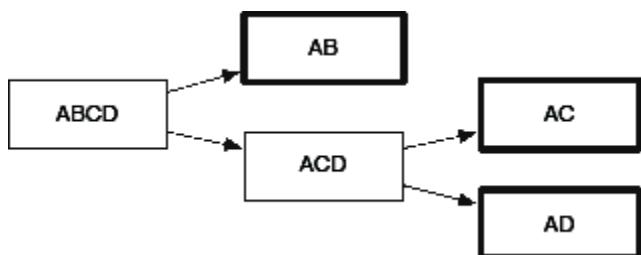
Situace po dekompozici:

R1



Jak dlouho lze provádět dekompozici

Příklad: Dostaneme relaci $R = (A, B, C, D)$ a množinu funkčních závislostí $F = \{A \rightarrow BCD\}$. Jak dále můžeme provádět dekompozici relace R , tak aby tato dekompozice měla vlastnost bezztrátového spojení?



Poznámky:

- všech normálních forem včetně BNCF lze dosáhnout postřednictvím dekompozice mající vlastnost bezzávislosti spojení
- pokud tedy již nemůže být relace dále dekomponována tak, aby nedošlo ke ztrátě informace (nelze provést dekompozici s vlastností bezzávislosti spojení), je zaručeně v BCNF
- **ale pozor:** to, že lze dosáhnout BNCF úplnou dekompozicí neznamená, že je to vždy potřeba, BCNF může být dosaženo již v některé dřívejší fázi dekompozice

Algoritmus pro provádění dekompozice do 3NF/BCNF

def decompose(R, F^+):

- nechť A je některý atribut z R
- nechť $X \subset R$
- **if** v množině F^+ existuje funkční závislost ($X \rightarrow A$), která porušuje 3NF/BCNF:
 - $R_1 = R - A$
 - $R_2 = XA$
 - decompose(R_1, F^+)
 - decompose(R_2, F^+)
- **else:**
 - done() # hotovo

Poznámka: R označuje relaci, kterou dekomponujeme. F^+ značí uzávěr funkčních závislostí (viz následující příklad).

Příklad: Dostali jsme relaci $R = (A, B, C, D, E)$ a $F = \{A \rightarrow B, B \rightarrow AE, AC \rightarrow D\}$. Klíče jsou AC, BC.

První průchod (R):

$A \rightarrow B$ porušuje BCNF v R .

Dekomponujeme R na $R_1 = (A, C, D, E)$ a $R_2 = (A, B)$

R_2 je nyní v BCNF, takže zde už nemáme co na práci. Zato R_1 vyžaduje další dekompozici.

Druhý průchod (R_1):

$A \rightarrow E$ (tranzitivně přes B) porušuje BCNF v R_1 .

Dekomponujeme na $R_{11} = (A, C, D)$ a $R_{12} = (A, E)$

Jak R_{11} , tak R_{12} jsou nyní v BCNF. Jsme tedy hotovi.

Finální dekompozice je tedy: $R_{11} = (A, C, D)$, $R_{12} = (A, E)$, $R_2 = (A, B)$

Kritéria kvality dekompozice

Kritéria kvality

Naše požadavky na kvalitu jsou:

- výsledná schémata by měla mít stejnou sémantiku (význam) (**podprobněji viz a)**)
- nová relace by měla obsahovat stejná data jako obsahovala původní relace (**podprobněji viz b)**)

a) pokrytí závislostí

Tomuto požadavku se také někdy říká „pokrytí původní množiny vlastností“. Cílem je aby původní schéma a schémata získaná dekompozicí nějak odrážela stejné vlastnosti. Důsledkem porušení je chudší sémantika. Vlastnosti to jsou vlastně funkční závislosti (označeny F). Musí tedy platit:

$$F^+ = \cup F_i^+$$

F jsou funkční závislosti v R

⁺ značí uzávěr (Matematicky: Uzávěr je nejmenší uzavřená množina, která danou množinu obsahuje. Uzavřená množina je taková množina, jejíž doplňek je otevřená množina. Množina je otevřená, pokud s každým bodem X, který do ní patří, patří do této množiny i jeho okolí.).

Pokud toto platí, říkáme, že R „**má vlastnost pokrytí závislostí**“. To znamená, že vezmemeli funkční závislosti v jednotlivých R_i (schématech vzniklých dekompozicí) a uděláme jejich uzávěr, měli bychom dostat totéž jako když uděláme uzávěr z F (tedy funkční závislosti v původním schématu).

Příklad: Máme-li například tabulku ADRESAR(MESTO, ULICE, DUM, PSC), jsou zde dvě netriviální závislosti: {město, ulice} → psc a psc → město.

Schéma chceme normalizovat, proto uděláme dekompozici: tabulka_ulic(ulice,dům,psc) a tabulka_měst(město,psc).

V dekomponovaném schématu můžeme opět najít závislost psc → město, ale už nemůžeme ověřit {město, ulice} → psc, což je špatně, protože jsme tak ztratili část vlastností v původním schématu.

b) bezzávislosti spojení

Nové spojení by měly obsahovat stejná data jako obsahovala původní relace. Dekompozici lze považovat za několik projekcí původní relace na množiny atributů nových schémat. Pro každou přípustnou relaci S^{*} by tedy mělo platit:

$$S^* = * S_i^* [A_i]$$

* na pravé straně výrazu značí spojení

S^{*} je relace podle schématu S

To znamená, že (**Poučka:**) S^* se dá rekonstruovat pomocí přirozeného spojení projekcí na atributy jednotlivých relací dekompozice. Lidsky řečeno: původní relace (tj. S^*) můžeme získat tak, že spojíme (tj. *) projekce na atributy (tj. to $S_i^* [A_i]$, což si můžeme představit jako výsledek JOIN), které vznikly v jednotlivých „pod-schématech“ vzniklých dekompozicí („jednotlivé relace dekompozice“). Pokud toto platí říkáme, že dekompozice „**má vlastnost bezztrátového spojení**“

Příklad: Máme tabulku ZNAMKY(predmet,student,znamka), tedy to je to S^* a tu dekomponujeme na: ZAPIS(předmět,student) a ZNAMKOVANI(předmět,znamka)

Na první pohled je vidět, že ztratíme informaci o tom, jakou který student dostal známku. Přestože možných kombinací máme více, nevíme která platí, takže informací máme méně. Důsledkem, že není daná dekompozice bezztrátová je, že ztratíme závislost mezi data – musíme tedy provést dekompozici dostatečně smysluplně.

Poučky:

Máme schéma R(A,B,C) a A,B,C jsou disjunktní (tj. různé, výlučné..) množiny atributů a funkční závislost $B \rightarrow C$. Rozložíme-li R na schémata R1(B,C) a R2(A,B) je tato dekompozice bezztrátová.

Z toho plyne:

Je-li dekompozice R1(B,C) a R2(A,B) bezztrátová, musí platit buď $B \rightarrow C$ nebo $B \rightarrow A$.

Normální formy

Proces normalizace se při návrhu databáze dělá proto, abychom dosáhli co nejvyšší výkonnosti při použití co nejúspornějších zdrojů. Normalizace databáze organzuje data podle jejich významu, je méně náchyná k problémům, snadněji upravitelná, redukuje přebytečná a opakovaně zadávaná data.

Máme pět normálních forem (a BCNF, což je varianta 3.). Obvykle normalizujeme do 3. normální normy (případně BCNF), normalizace probíhá postupně od 1. normální formy k vyšší tzv. chceme-li 3. musíme mít 1. a 2. splněnu!!! Ještě existuje 4. a 5. normální forma, ale v praxi se nepoužívají a často se ani neuvádí.

1. normální forma (1NF)

Každý atribut obsahuje pouze atomické hodnoty.

Atomické = dále nedělitelné (z pohledu databáze). Typickým příkladem je adresa: namísto sloupce adresa „Nerudova 123, Praha 4“ musíme mít sloupce ulice, čp, město, psč, atd.. Má to hned několik praktických důvodu – nelze například vyhledávat nebo seřadit výsledek dotazu podle jednotlivých částí adresy. Důležitá je i poznámka, že hodnoty mají být atomické z pohledu databáze viz. například datum nemusíme rozdělit na den, měsíc, rok atd, jelikož datum je v databázi atomická hodnota (existuje datový typ datum) – lidsky řečeno, tedy jeden atribut (sloupec) by měl obsahovat právě jednu hodnotu databázového typu. To je zároveň nejjednodušší i nejobtížnější myšlenka datového modelování.

Tato tabulka není 1NF

Jména	Příjmení	Adresa
Jan	Novák	Ostravská 16, Praha 16000
Petr	Nový	Svitavská 8, Brno 61400

Pepa	Vojtak	Znojemská 1, Beroun
------	--------	---------------------

Tato tabulka už je v 1NF

Jména	Příjmení	Ulice	č.p.	Město	PSC
Jan	Novák	Ostravská	16	Praha	16000
Petr	Nový	Svitavská	8	Brno	61400
Pepa	Vojtak	Znojemská	1	Beroun	26601

2. normální forma (2NF)

Každý neklíčový atribut je plně závislý na primárním klíci

Toto znamená, že se nesmí v řádku tabulky objevit položka, která by byla závislá jen na části primárního klíče. Z definice vyplývá, že problém 2NF se týká jenom tabulek, kde volíme za primární klíč více položek než jednu. Jinými slovy, pokud má tabulka jako primární klíč jenom jeden sloupec, pak 2NF je splněna triviálně.

Tato tabulka není v 2NF

číslo_zamestance	Jméno	Příjmení	číslo_pracovny	název pracoviště
1	Jan	Novák	10	studovna
2	Petr	Nový	15	posluchárna
3	Pepa	Vojtak	10	studovna

Jako primární klíč v této tabulce nemůže zvolit pouze číslo_zamestance, protože název pracoviště není závislý na ID zaměstnance, nemůžeme zvolit ani dvojci (číslo_zamestance,číslo_pracovny), protože jméno, příjmení a název pracovny nejsou plně závislé na dvojci zvoleného klíče. Lidsky řečeno: jméno, příjmení jsou závislé na čísle_zamestanace, zatímco název pracovny je závislý pouze na čísle pracovny a nezávisí vůbec na jménu zaměstnance. Není tedy možné docílit 2NF v jedné tabulce – musíme je rozdělit – odborně se tomu říká „dekompozice relačního schématu“.

Toto schéma už je v 2NF

číslo_zamestance	Jméno	Příjmení	číslo_pracovny
1	Jan	Novák	10
2	Petr	Nový	15
3	Pepa	Vojtak	10

číslo_pracovny	název pracoviště
10	studovna
15	posluchárna

10	studovna
----	----------

3. normální forma (3NF)

Žádný atribut není tranzitivně závislý na klíči.

Jiná definice (stejný význam): Všechny neklíčové atributy musí být vzájemně nezávislé. Myslim, že na škole se spíš preferuje první definice, takže vysvětlení: Tranzitivní závislost je taková závislost, mezi minimálně dvěma atributy a klíčem, kde jeden atribut je funkčně závislý na klíči a druhý atribut je funkčně závislý na prvním.

Tato tabulka není v 3NF

id	Jména	Příjmení	funkce	plat
1	Jan	Novák	programátor	35 000
2	Petr	Nový	technik	25 000
3	Pepa	Vojtak	vedoucí	75 000

Předpokládejme, že podle funkce je daný plat. V této tabulce je tedy na id závislá funkce, a plat je závislý na funkci, takže plat je transitivně závislý na id. Řešením je stejně jako v 2NF dekompozice.

Toto schéma je v 3NF

id	Jména	Příjmení	funkce
1	Jan	Novák	programátor
2	Petr	Nový	technik
3	Pepa	Vojtak	vedoucí

funkce	plat
programátor	35 000
technik	25 000
vedoucí	75 000

Rozdíl mezi 2NF a 3NF je v tom, že 2NF řeší situaci, kdy je primární klíč složený z více atributů zatímco 3NF řeší i to je-li primární klíč jeden atribut. Radši ještě jeden příklad: následující tabulka vyhovuje 2NF, ale ne 3NF.

Tato tabulka splňuje 2NF, ale není v 3NF

Turnaj	Rok	Vítěz	Datum narození
Indiana Invitational	1998	Al Fredrickson	21. června 1975
Cleveland Open	1999	Bob Albertson	28. září 1968
Des Moines Masters	1999	Al Fredrickson	21. července 1975
Indiana Invitational	1999	Chip Masterson	14. března 1977

Máme složený klíč (název_turnaje,rok_turnaje), který unikátně identifikuje řádku. 3NF je narušena tím, že neklíčový atribut datum_narozeni vítěze je přes neklíčový atribut vítěz závislý na klíči (turnaji). Náprava:

Toto schéma je v 3NF

Turnaj	Rok	Vítěz
Indiana Invitational	1998	Al Fredrickson
Cleveland Open	1999	Bob Albertson
Des Moines Masters	1999	Al Fredrickson
Indiana Invitational	1999	Chip Masterson
Vítěz	Datum narození	
Al Fredrickson	21. června 1975	
Bob Albertson	28. září 1968	
Al Fredrickson	21. července 1975	
Chip Masterson	14. března 1977	

Ještě se hodí poznamenat, že pokud bychom měli závislost klíč → klíč → neklíč nebo byly všechny atributy součástí nějakého klíče je schéma také v 3NF. Jinak 3NF je v praxi často dostačující tj. nejsou zde (většinou) aktualizační anomálie ani redundance (nemusí platit vždy viz. BCNF).

Boyce-Coddova normální forma (BCNF)

Školní definice: **Schéma R je v BCNF, jestliže pro každou netriviální závislost A→B platí, že A obsahuje klíč schématu R.**

Trochu upravená: Tabulka splňuje BCNF, právě když pro dvě množiny atributů A a B; A→B a současně B není podmnožinou A platí, že množina A obsahuje primární klíč tabulky.

Boyce/Coddova normální forma se pokládá za variaci třetí normální formy. V podstatě je vymezena stejnými pravidly jako 3NF forma, říká, že musí platit i mezi hodnotami uvnitř složeného primárního klíče. Tj. aby byla porušena musí platit tyto podmínky:

- Relace musí mít více kandidátních klíčů
- Minimálně 2 kandidátní klíče musí být složené z více atributů
- Některé složené kandidátní klíče musí mít společný atribut.

Pro vysvětlení se podívejme na příklad:

Toto schéma není v BCNF

Přednáška	Učitel	Místnost	Čas
Systémy	Vomáčka	M1	Ut3
Programování	Kryl	M3	St2
Programování	Kryl	M2	Pa4

Možné klíče jsou tady hodina-učitel, hodina-místnost a hodina přednáška - všechny atributy jsou součástí nějakého klíče tudíž je to schéma v 3NF, ale není v BCNF, protože učitel je závislý na přednášce (závislost mezi podklíči). Také je vidět, že přesto, že je schéma v 3NF, je zde redundance – informace, kdo učí kterou přednášku (předpokládejme, že to jeden předmět – jeden přednášející). Řešením je opět dekompozice:

Toto schéma už je v BCNF

Přednáška	Místnost	Čas
Systémy	M1	Ut3
Programování	M3	St2
Programování	M2	Pa4
Přednáška	Učitel	
Systémy	Vomáčka	
Programování	Kryl	

Tuto úpravou jsme odstranili redundanci přednáška – učitel a zároveň jsme neztratili informaci kdo co učí a kdy.

Každé schéma, které je v BCNF je také v 3NF, obráceně to ale neplatí. Má-li ale schéma jediný klíč nebo jednoduché klíče, potom je-li v 3NF je i v BCNF.

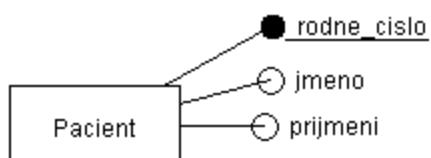
Návrh schématu relační databáze přímou transformací z konceptuálního schématu

Zadání mluví o návrhu relačního schématu databáze a ve skriptech (a tedy pravděpodobně i u státnic) si potřpí na důsledné odlišování toho co je možná v relačním schématu a co je možné v reálně používaných relačních databázích. Rozdíl je především v tom, že relační model jako takový nepovoluje NULL sloupce, zatímco ve všech běžně používaných databázích toto není problém. Na rozdíly z toho vyplývající upozorním v dalším textu.

Reprezentace silného entitního typu

- reprezentace silného entitního typu není problém
- výsledné schéma bude mít všechny atributy převáděně entity
- identifikační klíč budou tvořit všechny atributy, které se podílejí na klíči

Příklad:



Výsledné schéma relace:

```
Pacient(__rodne_cislo__, jmeno, prijmeni)
```

V SQL:

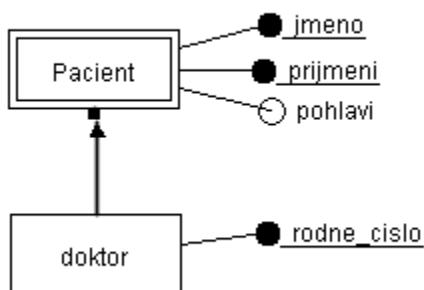
```
CREATE TABLE Pacient (
    rodne_cislo Integer NOT NULL,
    jmeno VarChar2(255) NULL,
    prijmeni VarChar2(255) NULL,
    Constraint PK_Pacient PRIMARY KEY (rodne_cislo)
)
```

Reprezentace slabého entitního typu

- nejprve do schématu relace dáme všechny atributy slabého typu, v tento moment schéma obsahuje jen část identifikačního klíče
- následně přidáme všechny identifikační atributy identifikačního vlastníka (tj. všechny atributy které se podílejí na klíci u entity na kterém je daná slabá entita závislá)

Příklad:

- předpokládáme, že nějaká instituce nesmí uchovávat rodná čísla pacientů, jen doktorů a spolehlá se, že jeden doktor nemá více pacientů se stejným jménem a příjmením



Výsledná schémata relací:

```
Doktor(__rodne_cislo__)
Pacient(__rodne_cislo__, __jmeno__, __prijmeni__, pohlavi)
```

V SQL:

```
CREATE TABLE doktor (
    rodne_cislo Integer NOT NULL,
    Constraint PK_doktor PRIMARY KEY (rodne_cislo)
)

CREATE TABLE Pacient (
    jmeno VarChar2(255) NOT NULL,
    prijmeni VarChar2(255) NOT NULL,
```

```

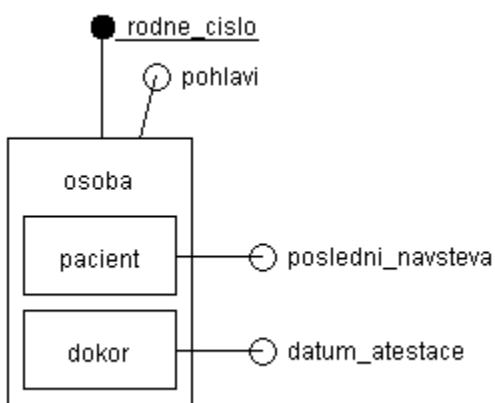
    pohlavi Char(1) NULL,
    d_rodne_cislo Integer NOT NULL,
    Constraint PK_Pacient PRIMARY KEY (jmeno, prijmeni, d_rodne_cislo)
)

```

Reprezentace ISA hierarchie

- při převodu předpokládám postup „shora dolů“, tj. od zdroje ISA hierarchie (u jednoduché závislosti, kdy je do zdroje ISA hierarchie vnořena jenom jedna úroveň entit je to jedno, ale při komplikovanějším vnoření to jedno není)
- v každém kroku vezmeme jednu entitu, vytvoříme k ní relaci jakoby se jednalo o klasický silný entitní typ a nakonec k ní přilepíme identifikační atributy zdroje ISA hierarchie (tj. pokud jedeme „shora dolů“ tak přilepíme identifikační atributy z nejbližšího nadtypu)

Příklad:



Výsledná schémata relací:

```

Osoba(__rodne_cislo__, pohlavi)
Patient(__rodne_cislo__, posledni_navsteva)
Doktor(__rodne_cislo__, datum_atestace)

```

V SQL:

```

CREATE TABLE osoba (
    rodne_cislo Integer NOT NULL,
    pohlavi Char(1) NULL,
    Constraint PK_osoba PRIMARY KEY (rodne_cislo)
)

CREATE TABLE patient (
    posledni_navsteva Date NULL,
    o_rodne_cislo Integer NOT NULL,
    Constraint PK_patient PRIMARY KEY (o_rodne_cislo)
)

CREATE TABLE doktor (
    datum_atestace Date NULL,

```

```

    o_rodne_cislo Integer NOT NULL,
    Constraint PK_dokor PRIMARY KEY (o_rodne_cislo)
)

```

Reprezentace vztahů

Vztah 1:1

Můžou nastat tři případy.

Oba entitní typy mají nepovinné členství ve vztahu

- pokud nemůžeme povolit pro sloupec NULL hodnoty, nemáme jinou možnost než vytvořit schémata tří relací, pro každý entitní typ jedno, třetí bude vztahové a bude obsahovat klíče obou předchozích jako cizí klíče
- jako primární klíč schématu vztahové relace může sloužit klíč kterékoliv ze dvou ostatních schémat relací
- případné atributy vztahu bude také obsahovat vztahová relace
- pokud bychom mohli použít NULL hodnoty, mohli bychom použít jen dvě relace a na konec jedné z nich přilepit klíčové atributy schématu druhé relace

Příklad:



Výsledná schémata relací:

```

Zamestnanec(__osobni_cislo__, jmeno, ...)
Auto(__spz__, vyrobce, ...)
Pouziva(__osobni_cislo__, __spz__)

```

Integritní omezení:

```

Pouziva[osobni_cislo] ⊆ Zamestnanec[osobni_cislo]
Pouziva[spz] ⊆ Auto[spz]

```

Jeden entitní typ má nepovinné členství ve vztahu, druhý povinné

- entitní typ, který má povinné členství ve vztahu je závislý na druhém entitním typu (který je nezávislý, protože členství ve vztahu pro něj není povinné)
- definujeme schémata dvou relací
- ke schématu relace pro závislý entitní typ přidáme atributy odpovídající identifikačnímu klíči nezávislého entitního typu, stejně tak k tému schématu přidáme případné atributy relace, v tomto schématu můžou být primárním klíčem opět jak klíčové atributy závislého typu, tak nezávislého

- pokud bychom mohli použít NULL hodnoty, mohli bychom použít jen jedno „slepené“ schéma relaci, přičemž všechny sloupce závislého entitního typu by mohly nabývat NULL hodnoty

Příklad:



Výsledná schémata relací:

```

classDiagram
    class Zamestnanec {
        __osobni_cislo, jmeno, ...
    }
    class Auto {
        __spz, vyrabce, ..., __osobni_cislo
    }
    Zamestnanec "0..1" -- "1..1" Auto : Pouziva
  
```

Integritní omezení:

```

classDiagram
    class Auto {
        __osobni_cislo
    }
    class Zamestnanec {
        __osobni_cislo
    }
    Auto "0..1" -- "1..1" Zamestnanec : Pouziva
  
```

Oba entitní typy mají povinné členství ve vztahu

- definujeme jedno schéma relaci: vznikne slepením schémat relací pro původní entitní typy
- oba původní klíče můžou být primárními klíči
- případně přidáme atributy odpovídající atributům vztahu

Příklad:



Výsledné schéma relace:

```

classDiagram
    class Zamestnanec {
        __osobni_cislo, jmeno, ..., __spz, vyrabce, ...
    }
    Zamestnanec "1..1" -- "1..1" Auto : Pouziva
  
```

Vztah 1:N

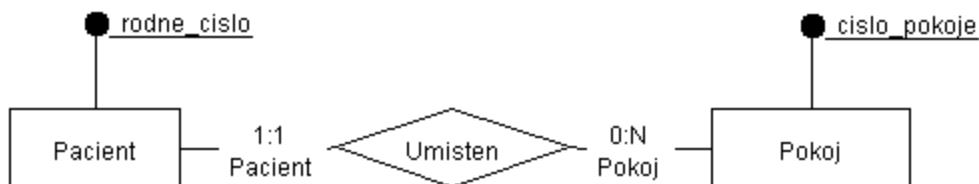
- Entita jednoho typu je **determinantem** entity druhého typu, pokud entitu druhého typu jednoznačně určuje. Tedy pokud pro **N** záznamů entity druhého typu existuje jenom **1** entita prvního typu, je entita prvního typu determinantem.
- přidat příklad ze skript

Povinné členství determinantu ve vztahu

- v takovém případě definujeme dvě relační schémata, pro každý entitní typ jedno
- k relačnímu schématu determinantu přilepíme cizí klíč odkazující k identifikačnímu klíči druhého

- entitního typu
- primárním klíčem bude klíč determinantu

Příklad:



Výsledná schémata relací:

```

  Pacient(__rodne_cislo__, ... , cislo_pokoje, ...)
  Pokoj(__cislo_pokoje__, pocet_luzek, ...)
  
```

Integritní omezení:

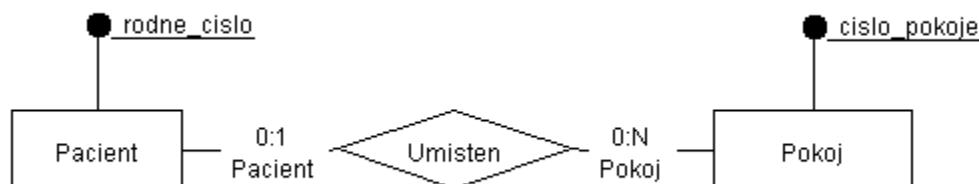
```

  Pacient[cislo_pokoje] ⊆ Pokoj[cislo_pokoje]
  
```

Nepovinné členství determinantu ve vztahu

- definujeme tři schémata relace, pro každý entitní typ jedno, třetí vztahové
- vztahové schéma bude obsahovat cizí klíče odkazující na primární klíče obou entitních typů, případně atributy odpovídající atributům relace, primární klíč vztahového relačního schématu bude ten z cizích klíčů, který odkazuje na primární klíč determinantu
- pokud bychom měli povolené NULL hodnoty, můžeme použít stejné řešení jako v předchozím bodě, s tím že cizí klíč by mohl být i NULL

Příklad:



Výsledná schémata relací:

```

  Pacient(__rodne_cislo__, ...)
  Pokoj(__cislo_pokoje__, pocet_luzek, ...)
  Umisten(__rodne_cislo__, cislo_pokoje)
  
```

Integritní omezení:

```

Umisten[cislo_pokoje] ⊑ Pokoj[cislo_pokoje]
Umisten[rodne_cislo] ⊑ Pacient[rodne_cislo]

```

Vztah M:N

- Vždy definujeme tři schémata relace, jedno pro každý entitní typ, a třetí vztahové, které bude obsahovat všechny primární klíče účastníků vztahu a případně další vztahové atributy.

Zdroje

- <http://www.cs.bilkent.edu.tr/~aacar/courses/CS352/SP09/Decomp.pdf> [http://www.cs.bilkent.edu.tr/~aacar/courses/CS352/SP09/Decomp.pdf] - dekompozice (hlavní zdroj, zájemcům doporučuji přečíst) [en]
- http://www.cs.toronto.edu/db/courses/343/Summer2003_L5201/tutorial8_s.pdf [http://www.cs.toronto.edu/db/courses/343/Summer2003_L5201/tutorial8_s.pdf] - dekompozice (řešené příklady)
- <http://www.cs.sfu.ca/CC/354/zaiane/material/notes/Chapter7/node7.html> [http://www.cs.sfu.ca/CC/354/zaiane/material/notes/Chapter7/node7.html] - dekompozice
- <http://service.felk.cvut.cz/courses/Y36DBS/> [http://service.felk.cvut.cz/courses/Y36DBS/] - stránka předmětu Y36DBS
- http://www.miroslavkrupa.cz/download/TZD_dist_2.pdf [http://www.miroslavkrupa.cz/download/TZD_dist_2.pdf] - relační datový model
- <http://www.owebu.cz/databaze/vypis.php?clanek=334> [http://www.owebu.cz/databaze/vypis.php?clanek=334] - relační datový model
- <http://www.owebu.cz/databaze/vypis.php?clanek=298> [http://www.owebu.cz/databaze/vypis.php?clanek=298] - konceptuální datový model
- http://cs.wikipedia.org/wiki/Relační_model [http://cs.wikipedia.org/wiki/Relační_model] - relační datový model
- http://cs.wikipedia.org/wiki/Relační_databáze [http://cs.wikipedia.org/wiki/Relační_databáze] - relační databáze
- <http://www.manualy.net/article.php?articleID=13> [http://www.manualy.net/article.php?articleID=13] - normální formy
- <http://interval.cz/clanky/databaze-a-jazyk-sql/> [http://interval.cz/clanky/databaze-a-jazyk-sql/] - normální formy
- http://en.wikipedia.org/wiki/Third_normal_form [http://en.wikipedia.org/wiki/Third_normal_form] - normální formy
- http://homen.vsb.cz/~s1i95/ISVDAS/IS/IS_relace.htm [http://homen.vsb.cz/~s1i95/ISVDAS/IS/IS_relace.htm] - relační algebra

Otázka 04 - Y36DBS

Zadání: Transakce, zotavení z chyb, koordinace paralelního přístupu, ochrana dat. Základní techniky ukládání záznamů do souborů a přístupu k datům v souborech. Použití souborů při konstrukci databáze. Indexy.

Slovník pojmu

- **Žurnál** - soubor, do kterého jsou ukládány změnové vektory při provádění změn v databázi. Je používán při rekonstrukci databáze po výpadku a k ROLLBACK operaci.
- **Instance** - Databázový server, který příjmá požadavky od klientů a pracuje se soubory databáze. Je to např. mysqld u MySQL.

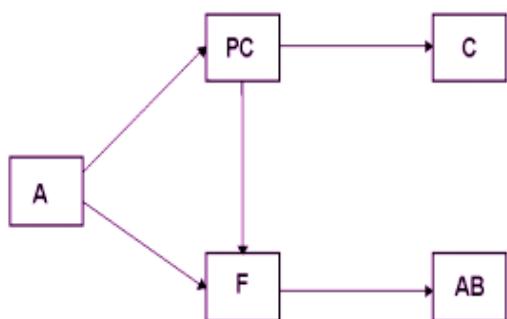
Transakce

Transakce je skupina příkazů, která neuvede databázi do nekonzistentního stavu. Transakce splňuje následující vlastnosti (**ACID**):

- **Atomicity (Atomicita)** - transakce se provede celá nebo vůbec
- **Consistency (Konzistence)** - transakce transformuje databázi z konzist. stavu do jiného konzist. stavu
- **Isolation (Izolovanost)** - dílčí výsledky transakce nejsou viditelné jiným transakcím a vrácením transakce (ROLLBACK) nesmí být zasažena žádná jiná transakce
- **Durability (Trvalost)** - změny uložené úspěšnou transakcí jsou uloženy do databáze a nemohou být ztraceny

Transakce se může nacházet v těchto stavech:

- Aktivní (A - **active**) - od počátku provádění transakce
- Částečně potvrzený (PC - **partially committed**) - stav po provedení poslední operace transakce
- Chybný (F - **failed**) - nelze pokračovat v normálním průběhu transakce
- Zrušený (AB - **aborted**) - nastane po skončení operace ROLLBACK
- Potvrzený (C - **committed**) - po úspěšném vykonání COMMIT



Transakce je zahájena příkazem **BEGIN** (nebo **START TRANSACTION**) a potvrzena příkazem **COMMIT**. Pro vrácení změn na počátek transakce slouží příkaz **ROLLBACK**. Mohou být vyvolány implicitně (samotným

systémem) nebo explicitně.

Zotavení z chyb (Recovery)

Výpadek nastane po:

- chybě v transakci
- zhroucení systému
- výpadku média

Možnosti obnovení:

- Obnova ze žurnálu
- Obnova ze záložní kopie

Obnova ze žurnálu probíhá nalezením transakcí, které probíhaly po posledním kontrolním bodu a aplikací procedur REDO (znovu provedení transakce) a UNDO (zrušení, ROLLBACK). Kontrolní bod vzniká periodickým ukládáním žurnálu na disk. Postup obnovy instance Oracle databáze:

1. Při (re)startu instance přečte synchronizační známky (SCN) v hlavičkách DB souborů a řídících souborů. Dále rozhodne, jestli pro obnovu budou stačit online redo log soubory nebo je potřeba použít archivované žurnály.
2. Postupně se od nejstaršího SCN přehraje celá transakční aktivita (REDO).
3. Datové i undo bloky jsou obnovené do stavu těsně před pádem instance.
4. Na transakce, které nebyly v době pádu ukončené příkazem COMMIT aplikuje ROLLBACK operaci.
5. Databáze je synchronizovaná a konzistentní.

Paralelní zpracování transakcí a uzamykání dat

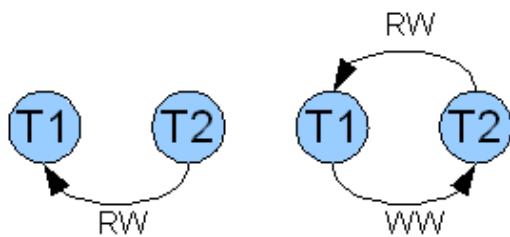
Hlavní důvody pro paralelní zpracování:

- **zvýšení výkonu a propustnosti** - zatímco jedna transakce čeká na data, druhá zatím může provádět agregační funkce na získaných datech
- **upřednostnění rychlých transakcí** - „pomalé“ transakce tak nebrzdí ty rychlejší

Rozvrhy:

- Transakce jsou při paralelním zpracování organizovány do rozvrhu operací.
- K zajištění izolovanosti dat se používá **zámků**, neboli zamykání dat na úrovni databáze/tabulky/řádku, jiným transakcím se tak odebere přístup k datům. Zamykání vede k řazení transakcí do fronty a zpomalení paralelního zpracování.
- **Rozvrh** je stanovené provádění akcí více transakcí ve stanoveném čase.
- **Seriové rozvrhy** zachovávají operace každé transakce pohromadě tím, že řadí celé transakce za sebe. Pro n transakcí proto existuje $n!$ seriových rozvrhů. **Nedochází ke vzniku konfliktů** mezi transakcemi a výsledkem je vždy konzistentní stav databáze.
- **Uspořadatelný rozvrh** Transakce už mohou být zpracovávány paralelně, ale stále existuje ekvivalentní sériový rozvrh.
- **Konflikty** zabraňují uspořadatelnosti rozvrhu, existují tyto typy:

- **WRITE-WRITE** – přepsání nepotvrzených dat
 - R1(x), R2(x), W1(x=3), W2(x=x+5) T2 přepíše neuloženou hodnotu T1
 - pozn. R1(x) - transakce T1 čte data x, W2(x) - transakce T2 mění hodnotu x, commit obou transakcí nejsou uvedeny
- **READ-WRITE** – neopakovatelné čtení
 - R1(x), R2(x), W2(x=x+4), R1(x) - T1 prečte podruhé stejnou proměnnou, ale ta už má jinou hodnotu
- **WRITE-READ** – čtení nepotvrzených dat
 - R1(x), W1(x=x+4), R2(x), rollback(T1) - T2 teď obsahuje hodnotu založenou na odvolaných datech
- **Fantom** - T1 přečte množinu řádků, T2 přidá řádek, při další operaci T1 přibyl řádek, který nebyl v předchozích operacích.
- Uspořádání bez konfliktů lze provést jako test na acyklickost grafu, kde konfliktní situace představují hrany a transakce vrcholy, viz. následující příklad:
 - S1: uspořadatelný rozvrh - R1(x), R2(x), W1(x)
 - S2: neuspořadatelný rozvrh - R1(x), R2(x), W2(x), W1(x)



- **Legální rozvrh** souvisí se zamykáním a splňuje tyto podmínky:
 1. Objekt je nutné mít uzamknutý, pokud k nemu chce transakce přistupovat.
 2. Transakce se nebude pokoušet uzamknout objekt již uzamknutý jinou transakcí (nebo musí počkat, než bude objekt odemknut).
- **Uzamykací protokol** je soustava pravidel, která zaručuje vznik sériového rozvrhu a splňuje tyto vlastnosti:
 1. Transakce zamyká objekt, chce-li k němu přistupovat
 2. Transakce se nesnaží uzamknout uzamčený objekt
 3. Transakce neodemyká objekt uzamčený jinou transakcí
 4. Transakce po svém skončení odemkne všechny objekty, které zamknula
- **Dvoufázové uzamykání:**
 1. fáze (uzamykací) - uzamyká se, nic neodemyká
 2. fáze (odemycací) - od prvního odemknutí, do konce se už nic nezamyká
 - zaručuje uspořadatelnost, ale už ne zotavitelnost ani bezpečnost proti kaskádovému rušení transakcí (ROLLBACK jedné transakce spustí ROLLBACK dalších závislých transakcí) nebo uváznutí (deadlock).
 - http://en.wikipedia.org/wiki/2_phase_locking [http://en.wikipedia.org/wiki/2_phase_locking]

Způsoby zamykání dat

Optimistický způsob zamykání

Předpokládá se, že nebude docházet ke konfliktům při přístupu k datům jiné transakce a data jsou zamykána pouze na nezbytně nutnou dobu. Zámky na čtení se nepoužívají. Může dojít ke vzniku deadlocku, kdy si dvě transakce navzájem drží zamknutá data. Transakce, která se dostane k datům první je také uloží, ostatní kolidující transakce provedou ROLLBACK.

Pesimistický způsob zamykání

Předpokládá se, že ke kolizím bude docházet často, data se zamykají i při přístupu pro čtení a po celou dobu transakce.

Ochrana dat

Ochrana dat před neautorizovaným přístupem, je docílena pomocí:

- uživatelských práv - udílení práv na objekty v databázi
- autentizace - ověření uživatele při připojení k databázi, je možné provádět ji přes heslo uložené v DB, přes OS, LDAP nebo servery 3. stran

Zabezpečení lze dále posílit šifrováním přenosu a auditem (sledováním přístupů do databáze).

Zabezpečení přístupu

Skupina SQL příkazů DCL (Data Control Language) přiděluje práva pro uživatele a uživatelské skupiny na objekty v databázi, obsahuje příkazy GRANT a REVOKE.

Syntax: GRANT privileges ON table T0 user

Příklad: GRANT insert,select,delete ON jidelnicek T0 cisnici

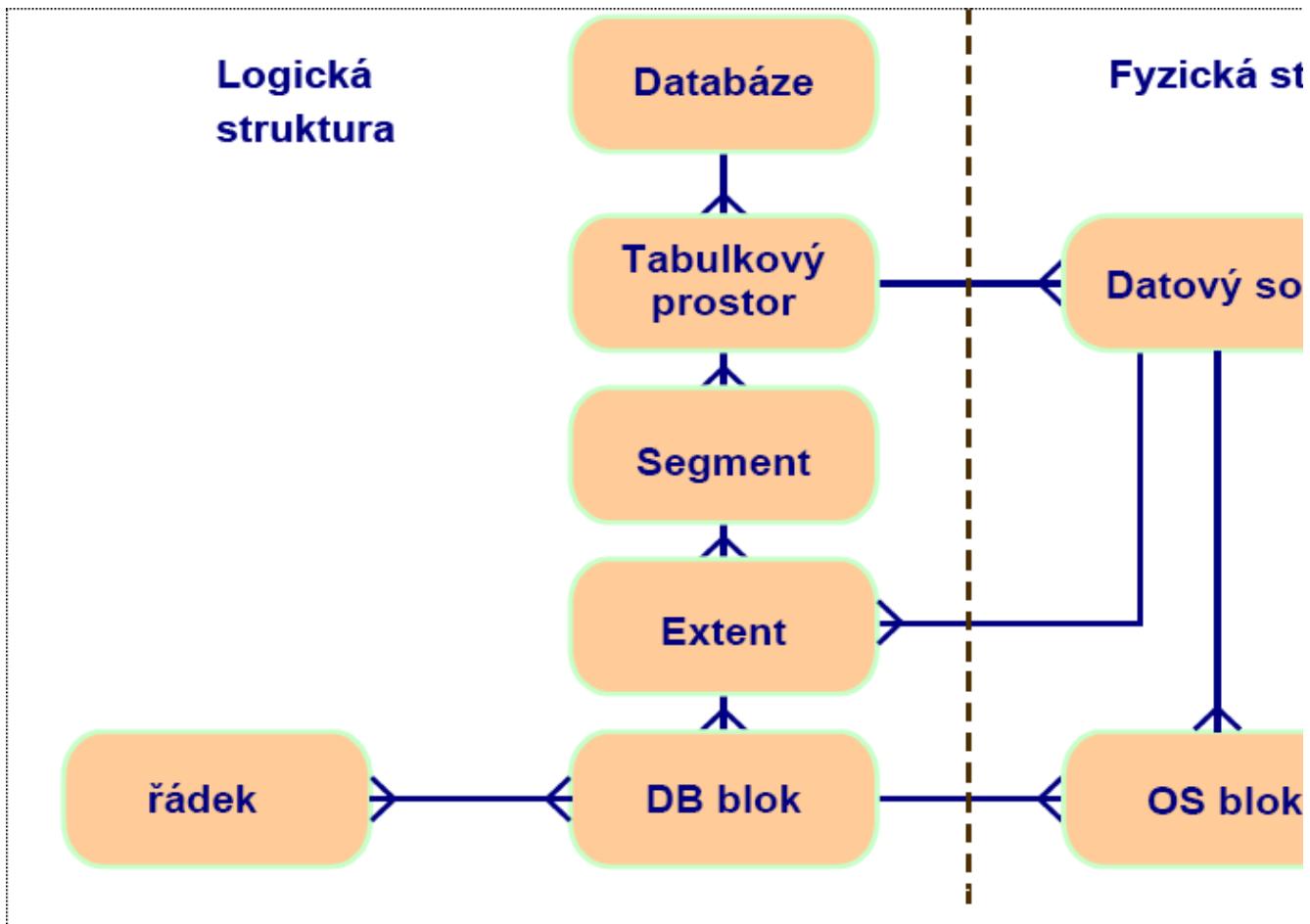
Privilegia se dále dělí na:

- **systémová** - právo na provedení akce (např. GRANT create session T0 brumla)
- **objektová** - vztahuje se ke konkrétnímu objektu (např. GRANT update(jmeno,prijmeni) ON osoby T0 kejchal)

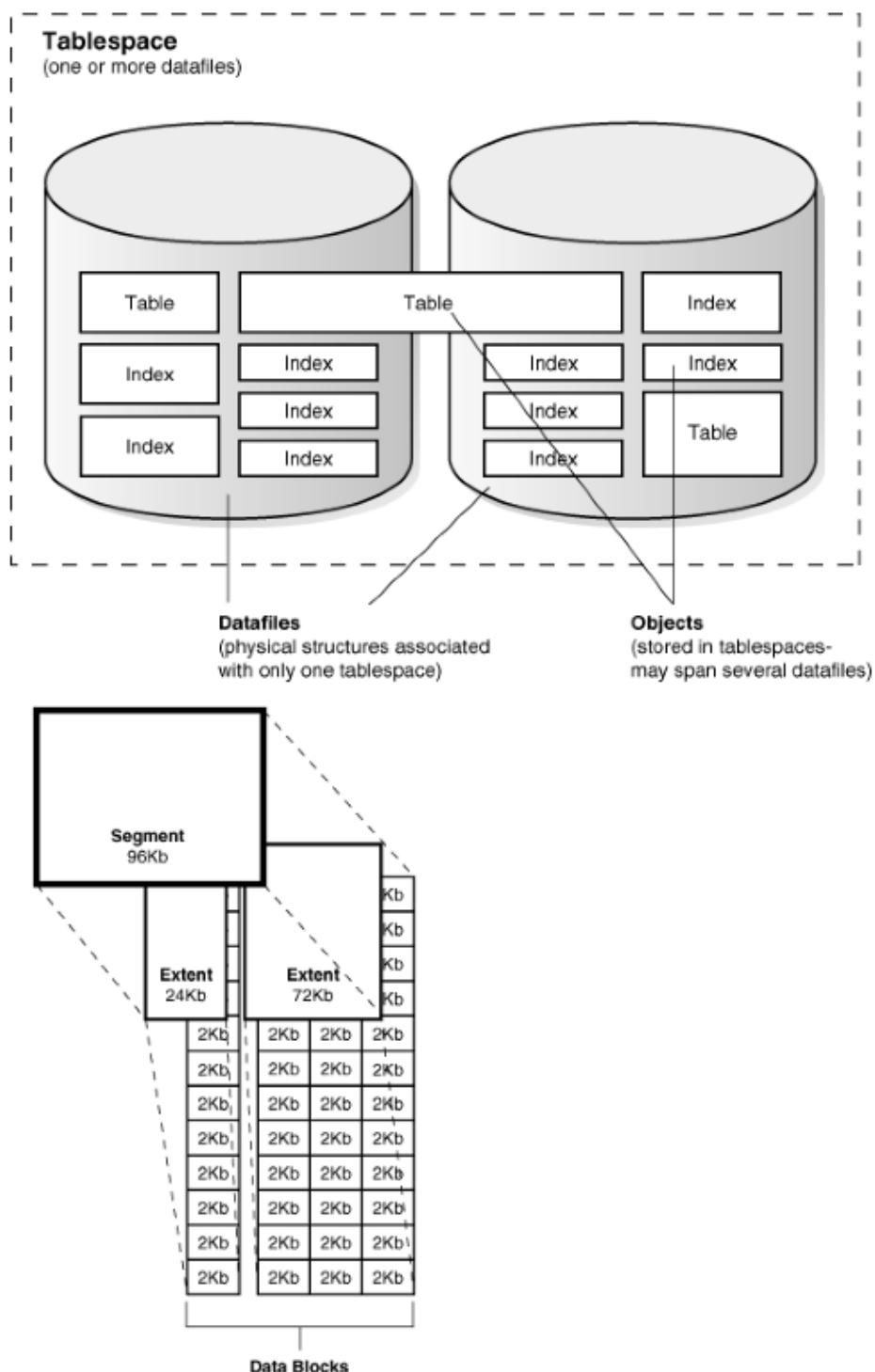
V Oracle je možné využít i **role**, což je pojmenovaná množina práv. Role lze strukturovat a uživateli jich lze přiřadit několik.

Fyzická struktura databáze

- Sekce obsahuje pouze princip ukládání dat pro Oracle, ostatní databáze byly probírány v předmětu Y36DBA.
- Data nemusí být uložena pouze v souboru, ale mohou být zapsána i na **raw device**, kde je využita přímo partition disku bez souborového systému z důvodu menší režie (a vyššího výkonu), rozdíl ovšem není příliš znatelný. Oracle při instalaci stále podporuje raw device.



- **Datový blok** je nejmenší alokovatelná jednotka, měl by odpovídat velikosti bloku filesystému nebo jeho násobkům.
- **Extent** je souvislá množina datových bloků, segmenty si alokují prostor po extentech z důvodu nižší fragmentace
- **Segment** je objekt v databázi (tabulka, pohled, index, procedura ...)
- **Databáze** používá k ukládání dat tzv. **tablespaces**, které se mohou skládat z více fyzických souborů.
Pokud dochází databázi volný prostor, není nic jednoduššího, než vytvořit další tablespace.



Další soubory:

- control file - obsahuje fyzickou i logickou strukturu DB
- redo log - obsahuje žurnál, transakční historii
- data files - .DBF soubory
- parameter file - obsahuje nastavení potřebné pro start instance
- password file - obsahuje systémové uživatele, lze autentizovat i proti OS

- archivované žurnály

MySQL a PostgreSQL používají adresáře jako tablespace, každá tabulka má alespoň jeden soubor (většinou je jich více: data, indexy a struktura).

Základní techniky ukládání záznamů do souborů a přístupu k datům v souborech

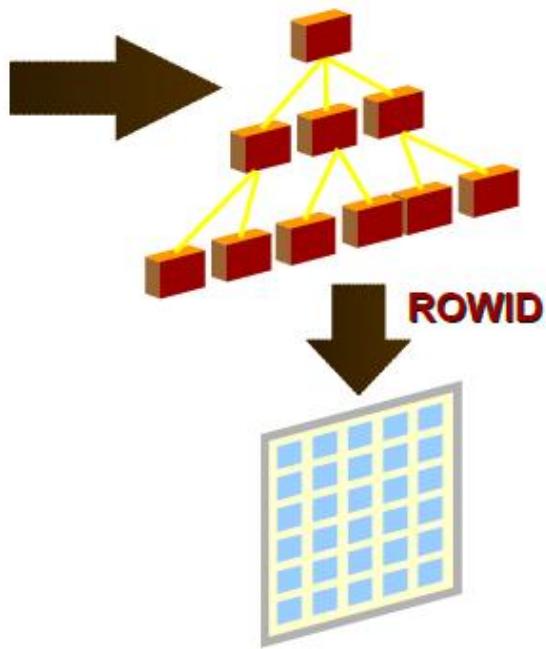
Schéma organizace souboru je popis logické paměťové struktury, do níž lze zobrazit logický soubor, spolu s algoritmy operací nad touto strukturou. Ta je obvykle tvořena z logických stránek (bloku pevné délky) a může být složena z více souborů (data ve více souborech, indexy, struktura tabulky ... závisí na použitém databázovém stroji).

Fyzická organizace tabulky:

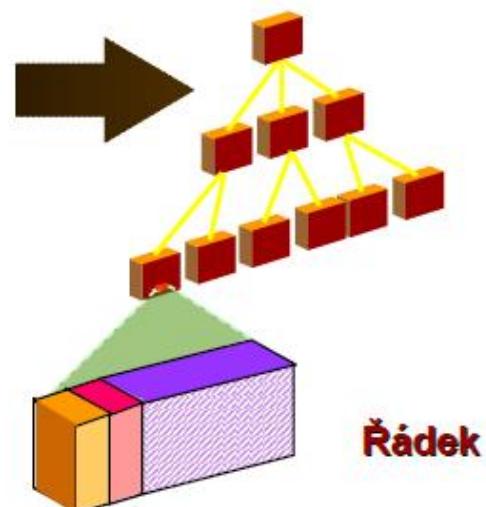
- **heap tabulka** (hromada/halda, neuspořádaný sekvenční soubor)
 - řádky nejsou ukládány v žádném určeném pořadí
 - při hledání dat je obvykle nutné projít celou tabulkou
 - pomalý sekvenční přístup
- **heap tabulka s indexy**
 - výrazně rychlejší přístup
 - indexy jsou obvykle oddělené od dat v heap tabulce (více v sekci Indexy)
 - při insert/update/delete operaci je potřeba znova vytvořit index
- **tabulka s indexovou organizací** (uspořádaný sekvenční soubor)
 - data jsou fyzicky organizována podle jednoho sloupce (indexu)

Indexově organizovaná tabulka

Heap tabulka s indexem



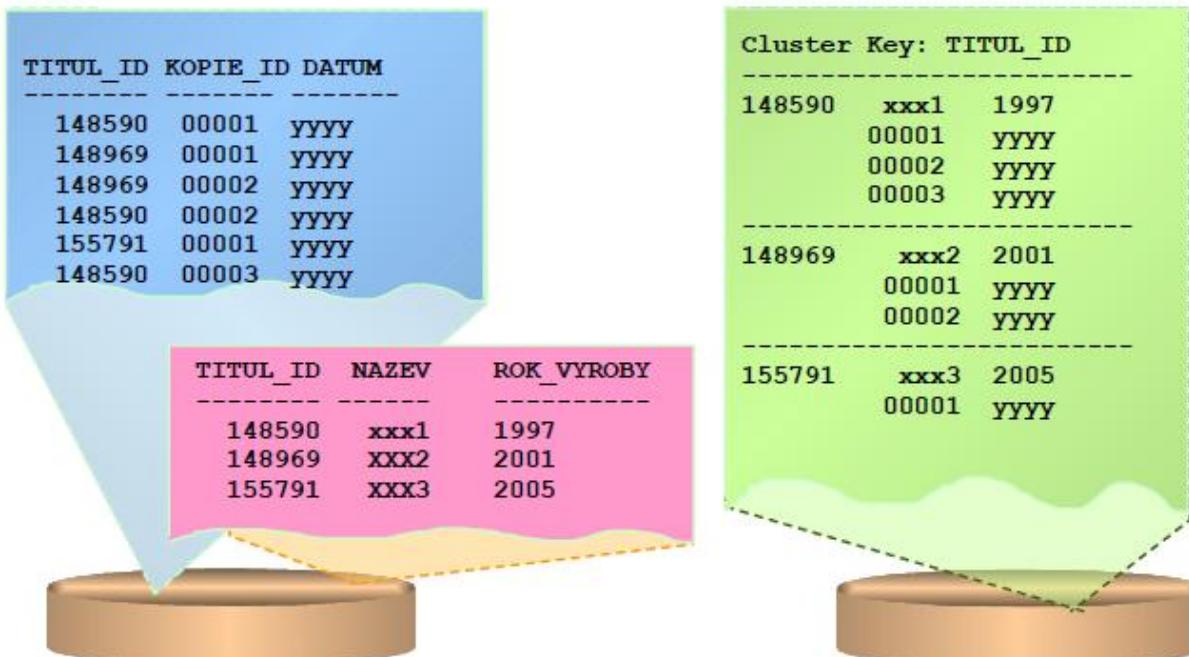
Indexově organizovaná tabulka



- **tabulka ve shluku (cluster)**

- sloučení několika tabulek do jedné
- data jsou seskupena a uspořádána podle *Cluster Key*, což je i klíč společný pro všechny tabulky

Shluk (Cluster)



**Samostatné tabulky s vazbou
přes cizí klíč**

Tabulky ve shluku

- **soubor s přímým přístupem**

- používá hashovací funkci pro nalezení a uložení záznamu
- záznamy jsou rozptýleny po celém souboru

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. Jejich použití zpomalí DML příkazy, ale výrazně urychlí čtení dat z tabulky.

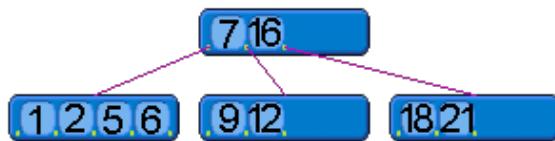
příklad vytvoření indexu: CREATE [UNIQUE] INDEX název ON tabulka (sloupec, ...)

typy indexů:

- PRIMARY - identifikuje záznam v tabulce, může být pouze jeden na tabulku
- UNIQUE - dovolí existenci pouze jedné hodnoty na tabulku
- INDEX - sekundární, použitý pro optimalizaci vyhledávání
- FULL-TEXT - implementace záleží na DB stroji, může např udržovat statistiku slov nebo vyhledávací tabulky

B-tree

- umožňuje vkládání, mazání a vyhledávání prvků se složitostí $O(\log N)$
- jedná se o vyvážený strom (Balanced v názvu) vhodný pro rozsáhlé struktury, které nemohou být v paměti
- pro B-tree řádu m , kde m je velikost plného uzlu, platí tyto pravidla:
 - kořen má nejméně dva potomky, pokud není listem
 - každý uzel má nejméně $m/2$ a nejvíce m potomků
 - každý uzel má nejméně $m/2 - 1$ a nejvíce $m/2$ datových záznamů
 - všechny cesty ve stromě jsou stejně dlouhé
 - záznamy v každém uzlu jsou vzestupně seřazeny



V databázích se častěji používá **B+tree**, kde poslední záznam každého listu ukazuje na první záznam dalšího listu a urychluje tím sekvenční procházení tabulky. Data jsou uložena pouze v listech.

Bitmapový index

- vytváří mapu bitů pro vybrané sloupce
- je vhodný pro sloupce s nízkou kardinalitou (nízkým počtem možných hodnot, např. ENUM v MySQL)
- je náročnější na údržbu než B-tree
- je výhodný pro dotazy v datových skladech (DSS), v praxi ale dosahuje stejných výsledků jako B-tree

Příklad

Tabulka osob, kde chceme vytvořit index podle pohlaví

Data		Bitmaps	
ID	Gender	F	M
1	Female	1	0
2	Male	0	1
3	Male	0	1
4	Unspecified	0	0
5	Female	1	0

Použití souborů při konstrukci databáze

Podle granta předmětu se pod tímto pojmem skrývá fyzická struktura databáze, různé databázové soubory (indexové, datové, clustery) nebo se nechá mluvit i o storage enginech různých databází, obzvlášť o MySQL, ale tohle už bylo náplní y36dba. Také se zmínil, že lze navázat na konceptuální a logický model. Jinak je tato část okruhu zpracována v otázkách okolo.

Odkazy

1. http://service.felk.cvut.cz/courses/36DBS/slides/8_transakce.pdf [http://service.felk.cvut.cz/courses/36DBS/slides/8_transakce.pdf] - přednáška
2. http://www.fit.vutbr.cz/study/courses/DSI/public/pdf/nove/9_2.pdf [http://www.fit.vutbr.cz/study/courses/DSI/public/pdf/nove/9_2.pdf] - transakční zpracování
3. http://en.wikipedia.org/wiki/Concurrency_control [http://en.wikipedia.org/wiki/Concurrency_control]
4. http://en.wikipedia.org/wiki/Two_phase_locking [http://en.wikipedia.org/wiki/Two_phase_locking]
5. [http://en.wikipedia.org/wiki/Schedule_\(computer_science\)](http://en.wikipedia.org/wiki/Schedule_(computer_science)) [http://en.wikipedia.org/wiki/Schedule_(computer_science)]
6. http://siret.ms.mff.cuni.cz/skopal/databaze/slidy/DBI025_09.ppt [http://siret.ms.mff.cuni.cz/skopal/databaze/slidy/DBI025_09.ppt] - Databázové systémy, transakce a rozvrhy

si:si4.txt · Poslední úprava: 2009/05/05 15:07 autor: Soran

Otázka 05 - Y36XML

Zadání: Základní principy formátů a nástrojů založených na technologii XML. Definice struktury pomocí schématu zapsaného v jazyce DTD nebo XML Schema. Reprezentace XML dat a dokumentů, rozhraní DOM a SAX

Základní principy formátů a nástrojů založených na technologii XML

- **XML 1.0** - je specifikace, která říká, co jsou tagy a atributy (viz. dále)
- **DTD** - je jeden z jazyků, pomocí nichž lze definovat strukturu XML dokumentu (viz. dále)
- **XML schema** - pomáhají vývojářům přesně definovat strukturu jejich vlastních formátů založených na XML (viz. dále)
- **jmenné prostory** - umožní návrhářům různých formátů použít značky se stejným názvem a uživateli pak používat tyto formáty dohromady (viz. dále)
- **rozhraní SAX** - rozhraní pro manipulaci s XML (viz. dále)
- **rozhraní DOM** - rozhraní pro manipulaci s XML prostredíctvom stromu (viz. dále)
- **XLink** - popisuje standartní cestu, jak přidat do XML souboru hypertextové odkazy
- **XSLT** - transformační jazyk používaný pro přidávání, odebírání i úpravu tagů a atributů
- **CSS** - štýl, dá se aplikovat na XML podobně jako na HTML
- **XSL** - je vylepšený jazyk pro zapisování stylů. Je založen na XSLT.
- **XPointer** - se ještě vyvíje, a budou sloužit k odkazování na části dokumentů

Jazyk XML

Jazyk HTML - pevně věřím - všichni známe. Stejně jako HTML i XML je značkovací jazyk, je však obecnější. Zatímco HTML má (v závislosti na verzi) pevně danou množinu značek, XML konkrétní značky nespecifikuje, definuje jen syntaxi, kterou je nutno dodržet, aby vznikl správný XML dokument.

Ukázka syntaxe:

```
<city name="Praha">
    <district>Karlín</district>
    <district>Strahov</district>
</city>
```

Celé XML je tedy jakýmsi *nadformátem*, konkrétní formáty realizované pomocí jazyka XML pak mají jasně definovanou podmnožinu značek. Takovým formátem může být třeba XHTML, které všichni známe, neboť je téměř k nerozeznání od HTML¹⁾.

Výhody:

- používá tagy pouze k ohrazení částí dat, a jejich interpretace je přenechána aplikaci, která data čte
- umožní vaše data uspořádat ve strukturách

- usnadňuje počítači tvořit, číst a zapisovat data
- plně vyhovuje standardu
- rozšiřitelné, nezávislé na platformě, a podporuje lokalizaci

Pojmosloví

Značka

Značka (tag) je řetězec mezi znakem < a znakem >

```
<značka>
```

Značky máme otevírací, viz výše, a ukončovací

```
</značka>
```

Pokud element nemá žádný obsah, je možno použít nepárovou značku, ta se pak zapisuje takto:

```
<značka />
```

Prvek

Prvek (element) je všechno od znaku < otevírací značky až po znak > ukončovací značky

```
<značka>Tento celý řetězec i se značkami je element</značka>
```

Atribut

Atribut je nějaká vlastnost dané značky, zapisuje se jako *jméno atributu* následované rovnítkem a řetězcem s *hodnotou atributu* v uvozovkách

```
atribut="hodnota atributu"
```

Definice struktury

Nabízí se otázka, jak specifikovat konkrétní formát. Většinou je potřeba definovat (výčet není vyčerpávající):

- jaké jsou přípustné názvy elementů
- jaké jiné elementy může daný element obsahovat, případně kolikrát a v jakém pořadí
- jaké atributy může element obsahovat
- jaké jsou povolené hodnoty těchto atributů

K tomu slouží schémata.

DTD

DTD je jeden z jazyků, pomocí nichž lze definovat strukturu XML dokumentu. Pojďme se podívat na nějaký příklad. Budeme chtít, aby XML dokument měl kořenový element **<cities>**, který bude moci obsahovat libovolný počet elementů **<city>**. Element **<city>** bude mít atribut **name** a dále bude obsahovat alespoň jeden element **<district>**. Specifikace takovýchto kritérií by mohla v DTD vypadat takto:

```
<!ELEMENT cities (city)*>
<!ELEMENT city (district)+>
<!ELEMENT district (#PCDATA)>
<!ATTLIST city
  name CDATA #IMPLIED >
```

Znaky ***** a **+** mají ve výše uvedených příklad funkci jako v regulárních výrazech, určují počty opakování příslušných částí. Pojďme si detailně ono DTD probrat: na prvním řádku říkáme, že budeme mít element **cities** jehož obsahem bude libovolný počet elementů **city** (tedy klidně i žádný). Na druhém řádku říkáme, že element **city** bude obsahovat alespoň jeden element **district**. Třetí řádek říká, že element **district** bude obsahovat nějaký text (PCDATA je zkratka pro Parsed Character DATA). Na čtvrtém řádku začíná specifikace atributů pro element **city**. Na pátém řádku říkáme že bude mít atribut **name** pak je datový typ (character data) a **#implied** znamená, že atribut není povinný (kdyby byl povinný, bylo by tam **#REQUIRED**).

Nyní rozšíříme ukázku XML dokumentu uvedenou na začátku článku tak, aby vyhovovala naší DTD specifikaci.

```
<?xml version="1.0" encoding="UTF-8"?>
<cities>
  <city name="Praha">
    <district>Karlín</district>
    <district>Strahov</district>
  </city>
</cities>
```

Za předpokladu, že máme výše uvedené DTD uloženo v souboru **definition.dtd** a chtěli bychom DTD k XML dokumentu explicitně připojit (aby např. programy, které budou dokument zpracovávat, mohly provést samy validaci), stačí přidat jediný řádek:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE cities SYSTEM "definition.dtd">
<cities>
  <city name="Praha">
    <district>Karlín</district>
    <district>Strahov</district>
  </city>
</cities>
```

DTD vs. CSS

Pokud má někdo pocit, že DTD je podobné CSS, pak bychom jej rádi vyvedli z mylu. CSS je navrženo pouze pro určení vizuálního (případně hlasového) formátování HTML. HTML samo o sobě (ve vyšších verzích) obsahuje jen samotnou informaci a CSS jí dodává nějakou konkrétní podobu. Důležité je si na tomto místě uvědomit, že máme jednu HTML stranku a k ní můžeme mít třeba 3 CSS předpisy. Určitě znáte nebo používáte nějaký web, na kterém jde přepnout vzhled. To je přepnutí CSS stylu. Ale důležité je, že obsah zůstává stále stejný, protože HTML se nezměnilo, mění se jen vzhled.

DTD naproti tomu slouží k tomu, abychom řekli, jaké značky vůbec budeme používat a jak mohou být do sebe vnořeny. Čili například existuje DTD, které definuje formát HTML. V něm se říká **musíte** mít značku **<head>** a v ní **může** být **<title>** atd. Kdybychom DTD upravili, vznikne úplně nový formát - nějaký paskvil odvozený od HTML. Čili jeden dokument nemůže mít více DTD. DTD zkrátka definuje konkrétní formát. Kdybychom se rozhodli vytvořit si nový formát založený na XML a chtěli jej prosadit do světa, vytvoříme definici v DTD a tu pak mohou vývojáři převzít a nás nový formát dodržovat 😊 DTD je tedy jakousi **formou** pro XML a konkrétní XML dokument je jakýsi **odlitek** z této formy. Snad se to dá pochopit.

XML Schema

Jazyk DTD má poměrně omezenou vyjadřovací schopnost. Navíc jeho velkou nevýhodou je, že sám není zapsán ve formátu XML, tím pádem se značně komplikuje jeho automatizované zpracování. Za největší nevýhodu pak bývá označována chybějící podpora jmenných prostorů (viz dále).

Popsané nevýhody řeší jazyk **XML Schema** - ten je sám o sobě XML. Navíc podporuje jmenné prostory a umožňuje specifikovat i datové typy.

Najlepší sposobom ako pochopiť XML Schema je na priklade

```
<zamestnanec id="101">
  <jmeno>Jan</jmeno>
  <prijmeni>Novák</prijmeni>
  <plat>25000</plat>
  <narozen>1965-12-24</narozen>
</zamestnanec>
```

V XML schematu můsime teda definovať elementy zamestanec, jmeno, prijmeni, plat, narozen a atribút id, pričom obsah elementov musí odpovedať určitému dátovému typu

```
<xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema">
<xss:element name="zamestnanec">
  <xss:complexType>
    <xss:sequence>
      <xss:element name="jmeno" type="xss:string"/>
      <xss:element name="prijmeni" type="xss:string"/>
      <xss:element name="plat" type="xss:decimal"/>
      <xss:element name="narozen" type="xss:date"/>
    </xss:sequence>
    <xss:attribute name="id" type="xss:integer"/>
  </xss:complexType>
</xss:element>
</xss:schema>
```

Čo nám teda XML schema vraví:

- celé schéma je XML dokument, ktorý používa špeciálne elementy
- všetky elementy patria do jmenného prostoru <http://www.w3.org/2001/XMLSchema> [<http://www.w3.org/2001/XMLSchema>] s obvyklým použitím prefixu xs alebo xsd
- cele schéma musí byť uzatvorené v elementu **schema**, obsahujúci definície elementu
- definice elementu sa zapisuje pomocou tágu **element**
- pre každý element musí byť určený jeho typ, rozlišujeme 2 druhy jednoduché a komplexné
 - **jednoduché typy** sa používajú skalárne hodnoty reťazec, číslo, dátum a pod. (môžme si

odvodiť vlastné)

- **komplexné typy** sa používajú ak element obsahuje iný element alebo atribút

- pomocou elementu **sequence**, že zamestnanec obsahuje 4 po sebe nasledujúce elementy a nutnosť výskytu(musia tam byť)
- počet jednotlivých elementov sa dá ovplivniť pridaním atributov **minOccurs,maxOccurs** npr. `<xss:element name=„plat“ type=„xs:decimal“ minOccurs=„0“/>` by dany element už neboli povinný, naopak ak by sme tam pridali `maxOccurs=„unbounded“` by sme určili, že zamestnanec môže mať ľubovoľný počet platov 😊
- pre každý element ďalej určujeme jeho jméno atribútom name a typ atribútom type
- atribút definujeme vnorené(to je medzi tágmy elementu,ktorému patrí), tágom attribute

Ak chceme v XML dokumente určiť kde je XML schema

Vloženie schématu v prípade, že nepoužívame jmenné prostory

```
<dokument
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="dokument.xsd">
    ...
</dokument>
```

- Ak chceme určiť kde má parser najst' XML schéma musíme k tomu použiť špecialné atribúty patriace do jmenného prostoru `http://www.w3.org/2001/XMLSchema-instance` [`http://www.w3.org/2001/XMLSchema-instance`]
- Atribút noNamespaceSchemaLocation určuje umiestnenie dokumentu

Vloženie schématu v prípade, že používame jmenné prostory

```
<dokument
    xmlns="urn:x-kosek:schemas:dokument:1.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:x-kosek:schemas:dokument:1.0
                        dokument.xsd">
    ...
</dokument>
```

- Pridali sme atribút schemaLocation, ktorý obsahuje dvojicu URI jmenneho prostoru a umiestnenie, môže obsahovať viac dvojíc
- Pribudol taktiež riadok `xmlns=„urn:x-kosek:schemas:dokument:1.0“`, ktorým definujeme jmenný prostor (viz. dále)

Well Formness vs. validita

Pokud chceme kontrolovat, zda je daný XML dokument správně zapsán, existují dvě úrovně kontroly. Ta nejzákladnější je kontrola tzv. Well-Formness. Dokument je Well-Formed, pokud odpovídá správné syntaxi XML. Čili značky se nám nekříží, jsou správně ukončeny atd.

Pokud XML dokument následuje nějaký konkrétní formát, který byl předem definován pomocí schématu, můžeme zkontovalovat validitu **vůči** tomuto schématu. Znamená to tedy, že se kontroluje, zda jsou správně názvy značek, zda je dodrženo jejich pořadí, zda jsou přítomny povinné atributy atd.

Jmenné prostory

Doufám, že z předchozího textu dostatečně jasně vyplynulo, že XML je jen obálka. Konkrétní formáty musí být definovány v nějakém schématu. Co se stane, když bychom chtěli použít onu obálku v podobě XML k uchování dvou a více formátů v jednom XML souboru? Konkrétní použití je třeba webová stránka psaná v XHTML (což je také XML) která má v sobě grafiku ve formátu SVG. SVG je vektorový formát zapsaný v XML. Oba formáty tedy můžeme v jednom dokumentu míchat. Problém by ale nastal, kdyby ony dva XML formáty měly nějakou značku se stejným názvem. Pak bychom nevěděli z kterého z nich je. A zde je řešení v podobě jmenných prostorů. V nadřazeném prvku nadefinujeme zkratky pro jednotlivé jmenné prostory a pak je používáme jako prefix v názvech značek.

Pokud vyrábíte HTML stránky, asi budete znát následující kus kódu.

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

Ten říká, že vše, co bude následovat, jsou elementy ze jmenného prostoru, jehož identifikátorem je řetězec <http://www.w3.org/1999/xhtml>. Pro jmenný prostor tam není definována žádná zkratka. Kdybychom nadefinovali zkratku např. **html** vypadal by pak kód takhle:

```
<html xmlns:html="http://www.w3.org/1999/xhtml">
```

Ovšem všude těle HTML stránky bychom pak museli psát **<html:table>**, **<html:ul>** atd.

Jmenné prostory tedy umožní návrhářům různých formátů použít značky se stejným názvem a uživateli pak používat tyto formáty dohromady. Deklarace probíhá vevnitř počátečního elementu atributem **xmlns**. Syntaxe je **xmlns:prefix_jmenného_prostoru=„URI_jmenného_prostoru“**.

- Prefix se využívá jako náhrada za dlouhé URI, které už není nutné dále psát
- Zápis **prefix:lokální_cast** se označuje jako **kvalifikované jméno**
- Elementy, které nelze v žádném jmenném prostoru nazýváme **nekvalifikované**
- Taktéž se dá definovat výchozí prostor **xmlns='URI'**, který se aplikuje na všechny elementy bez prefixu (viz příklad výše s HTML)
- Dalším významem jmenných prostorů je, že snadno určíme, které elementy patří k sobě.

Kvalifikované elementy

```
<pre:city xmlns:pre='mesta'>
  <pre:district>Karlín</district>
  <pre:district>Strahov</district>
</pre:city>
```

Kvalifikované elementy výchozím prostorem

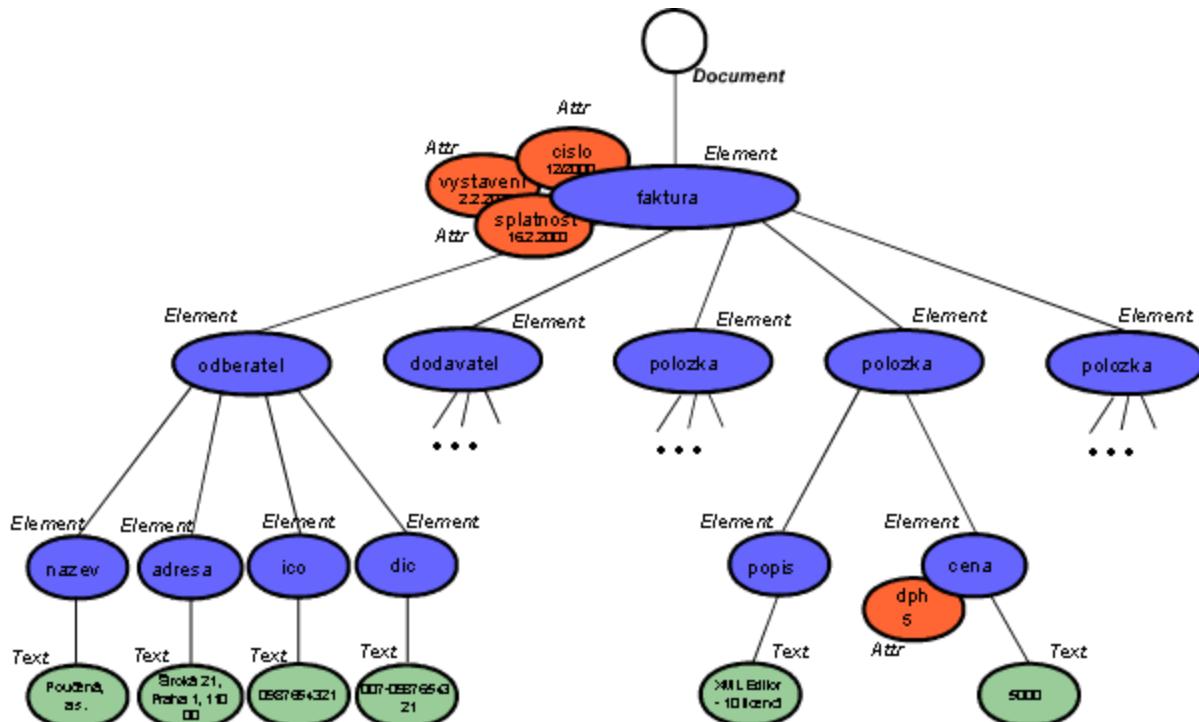
```
<city xmlns='mesta'>
    <district>Karlín</district>
    <district>Strahov</district>
</city>
```

Reprezentace dat

- Najčastejšou operáciou pri práci s XML je čítanie
 - Najlepší spôsob ako to spraviť je použiť hotové knihovne, ktoré sa nazývajú Parser
 - Parser číta XML zo súboru, alebo iného zdroja, a stará sa o nízkourovňovú syntaktickú analýzu (teda pozná, že za < prichádza element, vie že hodnota atribútu je v apostrofoch a pod.)

DOM

- najznámejšie rozhranie pracujúce so stromovou reprezentáciou dokumentu (XML dokument sa dá veľmi ľahko predstaviť ako strom)
 - XML dokument je sprístupnený pomocou objektov, ktoré zastupujú jednotlivé dôležité prvky (elementy, atribúty, textové prvky, a pod.)
 - každý objekt odpovedá jednému uzlu v strome XML dokumentu a ponúka metody pre zistenie jeho typu a hodnoty, jeho rodičov a detí
 - rozhranie vytvorilo W3C konzorcium a je nezávisle na použitom jazyku

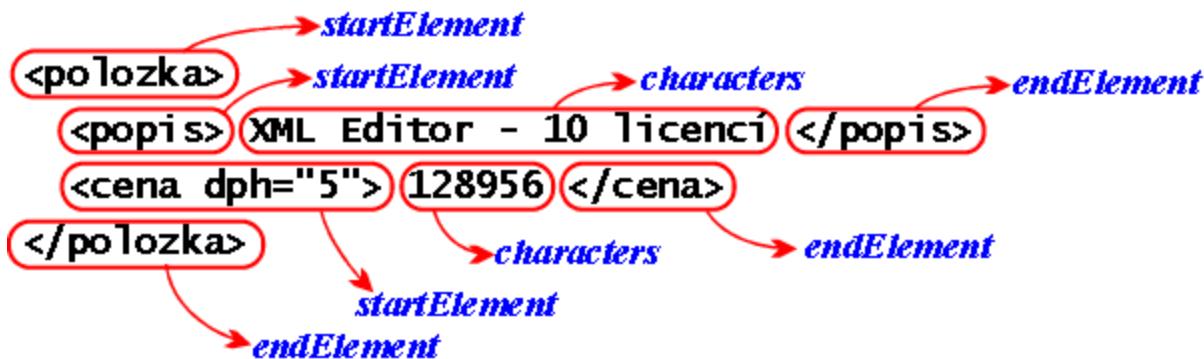


- #### ■ výhody DOM

- dokument môžeme prechádzať ľubovoľne a opakovane
- môžme manipulovať s objektmi
- **nevýhody DOM**
 - nutné načítať celý dokument vopred (pri veľkých dokumentoch = veľká časová náročnosť)
 - veľká pamäťová zložitosť (zaberie viac miesta v pamäti ako reálny dokument)
- **význam:**
 - poskytuje aplikáciám možnosť práce s XML

SAX

- je to udalosť-mi riadené rozhranie
 - parser postupne číta dokument
 - pre dôležité udalosti vola nami na definované funkcie (udalosti ako začiatok elementu, koniec, textový obsah elementu)



- nejedná sa o štandard W3C, ale vytvorila ho skupina ľudí okolo konferencie xml-dev
- **výhody** oproti DOM
 - vhodný pre veľké dokumenty, pretože ich nemusí ukladať v pamäti
 - je obecne rýchlejší pretože spracováva dokument okamžite
 - môžeme sa zamerať len na časť dokumentu, ktorá nás zaujíma (z tej potom vytvoriť strom)
- **nevýhody SAX**
 - nemôžeme sa vrátiť k predchádzajúcim dátam, už spracovaným (zabúda ich)
 - aktuálne je k dispozícii verzia SAX 2.0, ktorá priniesla podporu jemných prostoru
 - **význam:**
 - rovnako ako DOM poskytuje aplikáciám možnosť práce s XML, len sa jedná o iný spôsob

Slovníček pojmu

- **XML** - eXtensible Markup Language, česky rozširovateľný značkovací jazyk
- **DTD** - Document Type Definition, česky definice typu dokumentu
- **DOM** - Document Object Model, česky objektový model dokumentu
- **SAX** - Simple API for XML

Zdroje

1. <http://www.kosek.cz/xml/schema/uvod.html> [<http://www.kosek.cz/xml/schema/uvod.html>]
2. <http://www.kosek.cz/xml/api/> [<http://www.kosek.cz/xml/api/>]
3. <http://gis.zcu.cz/studium/pok/Materialy/Book/ar04s02.html> [<http://gis.zcu.cz/studium/pok/Materialy/Book/ar04s02.html>] - venuje sa jmenným prostorom.
4. <http://www.kosek.cz/xml/schema/wxs.html> [<http://www.kosek.cz/xml/schema/wxs.html>] - xml schéma detailne
5. <http://www.dynawest.cz/xml/XMLv10bodech.htm> [<http://www.dynawest.cz/xml/XMLv10bodech.htm>] - o XML v 10 bodoch, stručné a výstižné

¹⁾ Rozdíl mezi HTML a XHTML je ten, že původní HTML není validní XML, protože obsahuje např. neukončené značky apod.

Otázka 06 - Y36XML

Zadání: Jazyk XPath, Dotazovací jazyk XQuery, XML databáze a jejich vztah k jiným databázovým systémům.
Komprese XML dat.

Tento okruh se týká jazyka XML. Jazyk XML je značkovací jazyk podobný známému HTML, je však obecnější, můžeme v něm definovat vlastní značky. Jazykem XML se podrobněji zabývá otázka č. 5, doporučujeme ji nastudovat jako první.

Jazyk XPath

Jazyk XPath slouží k výběru elementů ze stromu XML dokumentu. Hned na začátek si uvedeme, kdy ho použijete. Například když budete chtít z XML dokumentu vybrat:

- všechny elementy daného jména
- element, který má nějakou konkrétní hodnotu atributu
(např. vyber element **kniha**, jež má jako hodnotu atributu **ISBN** daný řetězec)
- všechny elementy **kniha** jež jsou potomky elementu **knihovna**
- atd.

Výrazy v jazyce XPath se podobají cestám ve filesystému, což je poměrně intuitivní, protože i XML dokument je stromem, stejně jako filesystém.

Vezměme si ukázkový XML dokument

```
<?xml version="1.0" encoding="UTF-8"?>
<cities>
  <city name="Praha">
    <district>Karlín</district>
    <district>Strahov</district>
  </city>
  <city name="Bratislava">
    <district>Dúbravka</district>
    <district>Petržalka</district>
  </city>
</cities>
```

/

Když výraz začíná znakem / jde o absolutní cestu v XML dokumentu. Následující XPath výraz

```
/cities
```

vybere kořenový element, čili výsledkem bude

```
<cities>
</cities>
```

Všimněme si, že XPath výraz nám může vrátit i více výsledků, například tento

```
/cities/city
```

vrátí dva elementy **city** :

```
<city name="Praha"></city>
<city name="Bratislava"></city>
```

//

XPath výraz začínající na // vybírá daný element ať už se vyskytuje v dokumentu jakkoli hluboko. Např.:

```
//district
```

vrátí

```
<district>Karlín</district>
<district>Strahov</district>
<district>Dúbravka</district>
<district>Petržalka</district>
```

@

Zavináčem se v XPath označují atributy. Takže například následující výraz vybere všechny atributy name

```
//@name
```

Nebo všechny atributy name elementů city

```
//city/@name
```

Pozor ale, výše uvedené výrazy vrátí takzvaný uzel, čili celý atribut, nikoli jen jeho hodnotu.

Pokud bychom chtěli vypsat hodnoty atributů name v elementech city, pak musíme použít funkci data() (o funkcích níže):

```
data(//city/@name)
```

Výsledkem by bylo

```
Praha
Bratislava
```

Predikáty

Predikáty nám umožňují srovnávat. Zapisují se do hranatých závorek. Když známe nějakou hodnotu, podle které vyhledáváme, zapíšeme ji do predikátu. Výraz pak vrátí ty elementy, které predikát splňují. Např výraz:

```
//city[@name="Praha"]
```

vrátí celý příslušný element <city>.

Pokud predikát obsahuje pouze číslo, pak označuje pořadí prvku (**pozor**, indexované od 1, nikoli od 0). Výraz:

```
//city[2]
```

tedy vrátí Bratislavu. Stejně jako výraz

```
//city[last()]
```

Na závěr predikátů malinko složitější příklad:

```
string(//city[last()]/district[1])
```

vrátí textový obsah prvního elementu `<district>`, který je potomkem posledního elementu `<city>`. Tedy:

```
| Dúbravka
```

Osy

Ne, nebojte, nebudeme se zde dotýkat hrůzostrašného předmětu se stejným názvem 😊 Podíváme se na osy, které se vyskytují v XML stromě.

Název osy	Výsledek
ancestor	Vybere všechny předky (otce, dědečka...) daného uzlu.
ancestor-or-self	Vybere všechny předky nebo sebe sama.
attribute	Vybere všechny atributy daného uzlu.
descendant	Vybere všechny potomky (děti, vnoučata...) daného uzlu.
descendant-or-self	Vybere všechny potomky nebo sebe sama.
following	Všechno v dokumentu, co následuje po daném uzlu.
following-sibling	Všichni sourozenci daného uzlu.
namespace	Všechny uzly stejného jmenného prostoru jako je daný uzel.
parent	Vybere rodiče daného uzlu.
preceding	Všechno, co v dokumentu předcházelo danému uzlu.
preceding-sibling	Všechny předcházející sourozence
self	Vybere daný uzel

Celá cesta XPath výrazu se vyhodnocuje po krocích. Kroky jsou od sebe odděleny lomítky. Takže třeba výraz

```
//city/district
```

se vyhodnocuje tak, že se nejprve vyberou všechny elementy city, protože to je první krok (je ohraničen lomítky), pak se jede dál, takže v druhém kroku se vybírá mezi potomky elementů nalezených v prvním kroku.

Každý krok (řetězec mezi lomítky) má následující syntaxi:

```
| jméno-osy::uzel[predikát]
```

přičemž jméno osy i predikát můžeme vypustit. Nyní tedy už víte, k čemu využijete osy z výše uvedené tabulky.

Vezměme si výraz

```
| //district/ancestor::cities
```

a pojďme se podívat, jak funguje. V prvním kroku se vyberou elementy `<district>` kdekoli v dokumentu. V druhém kroku se hledají jejich předci, jenž se jmenují `<cities>`. Nevyberou se tedy všichni předci (přímý předek by byl například `<city>`), ale jen předci daného jména. Výsledkem tedy bude celý element `<cities>`.

Operátory

V XPath výrazech můžeme využívat následující operátory:

operátor	popis	příklad	návratová hodnota
	vyhodnotí oba výrazy	//book //cd	
+	sčítání	6 + 4	10
-	odčítání	6 - 4	2

*	násobení	$6 * 4$	24
div	dělení	$8 \text{ div } 4$	2
=	srovnání	price=9.80	true if price is 9.80 false if price is 9.90
!=	nerovná se	price!=9.80	true if price is 9.90 false if price is 9.80
<	menší než	price<9.80	true if price is 9.00 false if price is 9.80
≤	menší než nebo rovno	price≤9.80	true if price is 9.00 false if price is 9.90
>	větší než	price>9.80	true if price is 9.90 false if price is 9.80
≥	větší než nebo rovno	price≥9.80	true if price is 9.90 false if price is 9.70
or	disjunkce	price=9.80 or price=9.70	true if price is 9.80 false if price is 9.50
and	konjunkce	price>9.00 and price<9.90	true if price is 9.80 false if price is 8.50
mod	modulo	5 mod 2	1

Funkce

Občas je třeba použít nějakou funkci. V příkladech výše už jsme použili funkce `last()`, `string()` a `data()`, ale funkcí je celá řada. V tomto případě odkážu na externí zdroj, protože celý seznam je opravdu dlouhý:

http://www.w3schools.com/XPath>xpath_functions.asp [http://www.w3schools.com/XPath/xpath_functions.asp]

Použití XPath

XPath je poměrně obecným nástrojem, proto bývá spíše součástí dalších technologií (např. XQuery, viz níže) a nemá nějaké koncové uplatnění. Abyste XPath mohli použít, potřebujete k tomu příslušný software. V tomto ohledu doporučuji on-line nástroj pro zkoušení výrazů v XPath na adrese <http://www.whitebeam.org/library/guide/TechNotes/xpathtestbed.rhtm> [<http://www.whitebeam.org/library/guide/TechNotes/xpathtestbed.rhtm>]. Pokud nevíte jak začít, vložte do pole **XPath Expression** třeba řetězec `//child` a zmáčkněte tlačítko **Evaluate** a sledujte výpis XML o kus níže. Elementy, které se podbarví jsou ty, které výš XPath výraz právě „matchnul“ 😊

Dotazovací jazyk XQuery

Jazyk XQuery není jen dotazovacím jazykem, jak uvádí i zadání státnicového okruhu 😊, ale je i plnohodnotným funkcionálním programovacím jazykem. Primárně slouží k dotazování nad kolekcemi XML dokumentů. Můžete pomocí něj však vybudovat i plnohodnotnou webovou aplikaci.

Jazyk XQuery přímo využívá jazyka XPath. Kdybychom to měli vyjádřit zcela polopaticky, XPath se stará o tu část logiky, která říká **s čím** chcete manipulovat a XQuery pak řekne **co a jak** s tím budete dělat.

FLWOR

FLWOR je zkratka z FOR, LET, WHERE, ORDER BY, RETURN a čte se [flaur] - jako kytka. Tato zkratka naznačuje strukturu a pořadí příkazů typického XQuery dotazu. Možná jste si všimli slov WHERE a ORDER BY, ty nám napovídají, že pokládání dotazů v XQuery bude velmi podobné SQL dotazům.

Pojďme se podívat na ukázku. Předpokládejme, že výše uvedený XML dokument s městy je pojmenován `cities.xml`:

```
<html><head/><body><ul>
```

```
{
    for $city in doc("/db/wiki/nova/cities.xml")//city
    let $name := string($city/@name)
    return <li>{$name}</li>
}
</ul></body></html>
```

Výsledek bude

```
<html>
  <head/>
  <body>
    <ul>
      <li>Praha</li>
      <li>Bratislava</li>
    </ul>
  </body>
</html>
```

První, co si všimnete, že uvedený kód můžeme z jistého úhlu pohledu chápat jako šablonu. Vzpomeňte na své první pokusy v PHP a možná byste si místo XQuery dokázali představit právě nějaký ten PHP kód, který by vypisoval města z databáze do HTML stránky.

XML databáze a jejich vztah k jiným databázovým systémům

XML databáze je databáze navržená pro ukládání XML dokumentů. XML dokumenty můžeme rozdělit na dvě základní skupiny:

- XML dokumenty, které obsahují převážně data (anglicky *data based*)
- XML dokumenty, které obsahují převážně texty (anglicky *document based*)

XML databáze byly zprvu zaměřeny na druhý typ, tedy na skladování lidmi psaných textových dokumentů. S postupným rozšiřováním technologie XML se ale začaly orientovat na první typ, protože ten převládá. (Zkuste si představit, kolik sloupců ve vaší MySQL nebo PostgreSQL databázi obsahuje souvislý čitelný text - pravděpodobně minimum.)

Jen pro ilustraci a pro ujištění, že chápeme rozdíl mezi oběma typy:

Toto je *datově orientovaný XML dokument*:

```
<book id="358868">
  <price>256</price>
  <currency>CZK</currency>
  <stocks>5</stocks>
  <vat>19</vat>
</book>
```

a toto je *textově orientovaný XML dokument*:

```
<section>
  <title>Dnešní dopoledne</title>
  <para>To byste nevěřili, co jsem dneska zažil. Už od brzkého rána sedím u počítače a...</para>
</section>
```

Nativní XML databáze

Bývají také označovány NXD (Native XML Database). Jsou to databáze od začátku navržené pro ukládání XML, mezi takové patří například Apache Xindice [<http://xml.apache.org/xindice/>] - čte se [zin dý če] - nebo eXist [<http://exist-db.org/>].

Databáze s podporou XML

Mnoho relačních databázových systémů XML podporuje, ale znamená to, že interní forma uložení dat zůstává stále

relační, jen dochází k mapování XML na tuto strukturu. Sem patří Microsoft SQL Server, Oracle a další.

Vztah k jiným DB systémům

Napsal jsem Michalovi Valentovi co by si představoval jako náplň této části otázky. Zde je jeho odpověď:

Dobry den,
tahle otazka patri k predmetu Technologie XML. Nicmene, mel-li bych na tema
Vztah k jinym databazovym systemum odpovidat ja, asi se drzel techto bodu:
- databazove modely - z hlediska slozitosti a vyjadrovacich moznosti: relacni
model - XML - objektovy model
- z hlediska vyniku zrejme staci konstovat, ze XML databaze nejsou tak vykonne
jako relacni (jednak slozitejsi model a jednak kratsi doba vyvoje)
- aktualne to vypada, jakoby se opakovala situace s nastupem objektovych
databazi z 90. let - existuji OODB, ale z hlediska objemu nasazeni zvitezily
(nikoliv prekvapive) relacni databaze s objektovym rozsireniem, OOD rysy se
postupne dostaly do standardu jazyka SQL. Podobne dnes vetsina velkych
zabehnutych DBMS poskytuje XML rozsireni a prace s XML se take dostava do
standardu SQL

Zduraznuju, ze jsou to MOJE SUBJEKTIVNI nazory, nicmene soudim, ze polemikou
tohoto typu u statnic nikoho neurazite ;-).

Michal Valenta

Komprese XML dat

Jazyk XML byl navržen tak, aby byl relativně snadno čitelný a případně i editovatelný pro člověka. Nese to s sebou však i nevýhody a jednou z nich je větší datový objem. Kdyby ty samé informace byly uloženy binárně, zabraly by daleko menší objem.

Dobrou zvyklostí je značky v XML pojmenovávat pokud možno dostatečně jasně. Proto místo

<pt>1</pt>

použijeme raději

<paymentType>card</paymentType>

To je však o mnoho znaků delší. Když budeme mít v XML dokumentu dump s údaji o několika tisících položkách zboží a každý bude obsahovat tento element, nárůst objemu dat již bude znatelný.

Dobré je v tomto smyslu uvažovat o poměru délky celého XML dokumentu a délky skutečného datového obsahu v něm.

Nabízí se tedy myšlenka XML komprimovat. Musíme si však uvědomit, že standardní komprimační algoritmy (ZIP, RAR) nejsou specificky pro XML navrženy, takže nebudou dosahovat nejlepších možných výsledků.

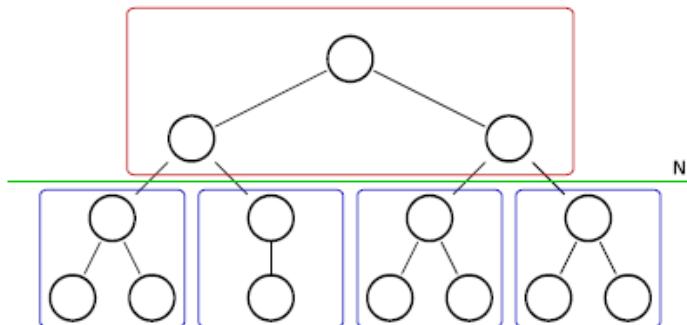
Dále můžeme uvažovat o tom, co budeme s komprimovaným XML vlastně dělat. Zda komprimujeme kvůli **přenosu**, nebo kvůli ušetření místa při **uložení**.

Jakákoli manipulace včetně čtení dokumentu by vyžadovala jeho dekomprimaci. Tedy případný XML databázový stroj, který by z důvodu úspory místa ukládal dokumenty komprimované, by spotřeboval velké množství výkonu na neustálou dekomprimaci. Skladování XML zkompresovaného pomocí běžných algoritmů tedy není příliš výhodné.

Existují však komprimační algoritmy přímo navržené pro XML a jsou mezi nimi dokonce takové, které podporují dotazování nad zkompresovaným dokumentem. Následující text je krácenou citací z diplomové práce Lukáše Skřívánka (viz zdroje):

XMLZip

XMLZip je kompresní program určený pro XML dokumenty založený na standardu DOM. Nejprve načte XML dokument pomocí parseru typu DOM a poté rozdělí vzniklou stromovou strukturu do několika komponent – kořenové komponenty, která obsahuje data do hloubky N stromu a jednu komponentu pro každý podstrom začínající v hloubce N.



XMLZip umožňuje uživateli zvolit hloubku N oddělující kořenovou komponentu od ostatních komponent. Kořenová komponenta obsahuje odkazy do jednotlivých nižších komponent. Poté jsou jednotlivé komponenty komprimovány zvlášť.

XMLZip vždy dekomprimuje pouze ty komponenty, ke kterým uživatel přistupuje.

XGrind

XGrind je kompresní program, který podporuje dotazování nad komprimovaným XML dokumentem, umožňuje však pouze pohyb ve stromu XML dokumentu od předka k potomkům. Používá různé kompresní algoritmy. Pro kompresi hodnot atributů přečte z DTD dokumentu zda se jedná o data výčtová či textová a podle toho dále volí kompresní algoritmus. XGrind komprimuje názvy elementu a atributu slovníkovou metodou a elementy ukončuje speciálním znakem.

Kvůli podpoře efektivního dotazování nad komprimovaným XML dokumentem potřebuje XGrind pro kompresi textových hodnot atributů a elementů kontextově nezávislý kompresní algoritmus. To znamená, že kódování znaku je nezávislé na pozici v textu. To umožňuje vykonání dotazu na rovnost a na shodu prefixu bez dekomprese a v případě intervalových dotazů a dotazů na podřetězec částečnou dekomprezí.

XPress

XPress je kompresní program, který přímo podporuje dotazování nad komprimovaným XML dokumentem, ale stejně jako XGrind umožňuje pohyb ve stromu XML dokumentu pouze ve směru od předka k potomkům. Přichází s novou metodou kódování zvanou reverzní aritmetické kódování určenou pro kódování cesty ve stromu XML dokumentu.

Pro ukládání hodnoty atributu a elementu používá XPress rozpoznávání datového typu. Podle něho volí kompresní metody přímo určené pro daný typ, čímž dosahuje lepšího kompresního faktoru.

Shrnutí

Poznámka vypracovávajícího: nedovedu si představit, že by nás někdo zkoušel na názvy těch komprimovačních programů, natož na jejich přesné principy. Proto připojuji moje polooficiální shrnutí.

- Komprimování XML je hezká věc, ale když se chceme **nad zkomprimovaným dokumentem dotazovat**, legrace přestává.
- Je třeba přijít s algoritmem, který dokáže porovnávat např. zkomprimované hodnoty atributů. to znamená, že **1)** ten algoritmus musí vědět mezi svými daty co je atribut a **2)** musí hodnotu zadanou v dotazu také nejprve zkomprimovat a pak binárně porovnávat s daty. Usměv už nás opouští, že?
- Komprimovační programy jsou navíc natolik sofistikované, že si umějí přechroustat DTD a určit si, pro jakou část dokumentu bude výhodnější použít jaký typ komprese. To dělají, aby ušetřily fakt každý možný byte.
- Hezky se dá ušetřit při použití **slovníkové metody**. Uvědomme si, že hodně znaků v dokumentu zabírají názvy

elementů. Slovníková metoda funguje jednoduše. Vytváří si takový malý slovníček elementů a přiřazuje jim ekvivalenty. Např. <paymentType> bude 1, <someOtherDeadlyLongElementName> bude 2 atd... V komprimované podobě jsou pak jen ekvivalenty. Ukončovací znaky se vůbec nepíšou, ty se nahrazují všechny jedním stejným znakem a program musí být tak chytrý, že si pamatuje, co za element má právě otevřeno.

- Suma sumárum komprimační program na XML bych fakt nechtěl psát jako semestrálku.

Slovníček pojmu

- **XML** - eXtensible Markup Language, česky rozšířitelný značkovací jazyk
- **NXD** - Native XML Database
- **DB** - Data Base, databáze

Zdroje

1. Stránky k předmětu X36XML na STM-WIKI
<http://stm-wiki.cz/index.php/Y36XML> [<http://stm-wiki.cz/index.php/Y36XML>]
2. Slidy (i nezkrácené) z předmětu Y36XML
<http://service.felk.cvut.cz/courses/Y36XML/slides/> [<http://service.felk.cvut.cz/courses/Y36XML/slides/>]

XPath

1. <http://interval.cz/clanky/zaklady-jazyka-xpath/> [<http://interval.cz/clanky/zaklady-jazyka-xpath/>]
2. http://www.zvon.org/xxl/XPathTutorial/General_cze/examples.html [http://www.zvon.org/xxl/XPathTutorial/General_cze/examples.html]
3. On-line testovací nástroj pro pokládání XPath výrazů - **doporučuji!**
<http://www.whitebeam.org/library/guide/TechNotes/xpathtestbed.rhtm> [<http://www.whitebeam.org/library/guide/TechNotes/xpathtestbed.rhtm>]
4. http://www.w3schools.com/XPath>xpath_axes.asp [http://www.w3schools.com/XPath>xpath_axes.asp]

XQuery

1. <http://en.wikipedia.org/wiki/XQuery> [<http://en.wikipedia.org/wiki/XQuery>]

XML Databáze

1. Vynikající referát Josefa Šína vypracovaný pro potřeby předmětu DBA - **doporučuji!**
http://www.cecak.cz/fel/dba/referaty/xml/teoreticke_zaklady_a_prakticke_vyuziti [http://www.cecak.cz/fel/dba/referaty/xml/teoreticke_zaklady_a_prakticke_vyuziti]

Komprese XML dat

1. Diplomová práce Komprese a dotazování nad XML dokumenty Lukáše Skrivánka
https://dip.felk.cvut.cz/browse/pdfcache/skrivl1_2007dipl.pdf [https://dip.felk.cvut.cz/browse/pdfcache/skrivl1_2007dipl.pdf]

Otázka 07 - Y36SI3

Zadání: Principy a techniky testování. Odchylky versus chyby, třídy ekvivalence chyb, hraniční testování. Typy odchylek a defektů.

Principy testování

Cílem testování je objevit chyby v softwaru a ne ukázat, že v něm chyby nejsou.

Základní principy testování můžeme shrnout do následujících několika bodů:

- Testy by se měly vztahovat k požadavkům zákazníka (většina defektů z pohledu zákazníka se projeví neshodou s požadavky).
- Testy by měly být plánovány v předstihu (plánovat je jakmile jsou specifikovány požadavky, navrhnout data, jakmile je proveden návrh programu.)
- Princip Pareto (pravidlo 80:20): většina všech neobjevených chyb má původ v několika málo modulech. Problémem je ty moduly nalézt.
- Testování by mělo začít testováním "v malém" a pokračovat testování "ve velkém". (Začít testováním modulů, pak integrovaných skupin (clusterů) modulů a nakonec celého systému)
- Úplné otestování není možné (nelze otestovat počet všech kombinací cest programem)
- Testování by mělo být vedeno nezávislou třetí stranou (jiný úhel pohledu, snaha nalézt chybu, ne dokázat, že tam není)

Výčet vlastností vedoucích k dobré testovatelnosti:

- **Spustitelnost (operability)** - čím lépe modul pracuje, tím účinněji může být otestován - chyby nebrání běhu programu, může být testován a vyvíjen současně.
- **Přehlednost (observability)** - testuješ, co vidíš - různé výstupy pro různé vstupy, stavy systému a proměnné jsou viditelné během chodu, faktory ovlivňující výstup jsou viditelné, nesprávné výstupy jsou lehce identifikovatelné, vnitřní chyby jsou automaticky detekovány a zaznamenávány, je dostupný zdrojový kód .
- **Kontrolovatelnost (controllability)** - čím lépe lze software kontrolovat/řídit, tím spíše lze testování automatizovat a optimalizovat - všechny možné výstupy jsou generovány nějakou kombinací vstupů, veškerý kód je spustitelný nějakou kombinací vstupů, vstupní a výstupní formát je konsistentní
- **Dekomponovatelnost** - kontrolováním rozsahu testování můžeme rychleji oddělit problémy a provést následné testy.
- **Jednoduchost** - čím méně máme testovat, tím rychleji to můžeme provést - jednoduchost funkce, struktury, kódu
- **Stabilita** - čím méně změn softwaru, tím lépe - nejsou časté, jsou řízené, neovlivně provedené testy.
- **Srozumitelnost** - čím víc máme informací, tím budou testy chytřejší - srozumitelný návrh, srozumitelné závislosti mezi vnitřními, vnějšími a sdílenými komponentami, dostupnost a

srozumitelnost technické dokumentace.

Techniky testování

4 stupně testování (podrobněji rozebrané v otázce č.8):

- **jednotkové(Unit) testy** -
 - provádí sám vývojář
 - white-box testing
- **integrační testy** -
 - verifikace programové konstrukce
 - black-box částečně white-box -pokrytí hlavních cest řízení
- **validační testy** -
 - ověření, že program vyhovuje požadavkům na funkci, chování a provedení(black-box)
- **systémové testy** -
 - test v kombinaci s ostatními systémovými prvky - HW, databáze, uživatelé

použité techniky

- **whitebox testing**
 - používá informace o vnitřní struktuře testovaného modulu a znalosti zdrojového kódu
 - potřebné informace:
 - Testovaný modul
 - Soubor specifikací výstupu pro specifický vstup.
 - Počet větvení ve zdrojovém programu.
 - Počet a typ cyklů ve zdrojovém programu.
 - Počet volání funkcí ve zdrojovém programu.
 - Počet nepodmíněných skoků ve zdrojovém programu.
 - Použití rekurze, které je třeba věnovat zvláštní pozornost.
 - prověřuje následující programové struktury:
 - všechny samostatné cesty uvnitř modulu budou provedeny alespoň jednou
 - všechny podmínky se projdou větví s hodnotou ANO (true) i větví s hodnotou NE (false)
 - projde všechny cykly
 - prověří vnitřní datové struktury
- **blackbox testing**
 - testování jednotlivých modulů bez znalosti vnitřní logiky
 - víme co by měly moduly produkovat za výsledky a ověřujeme, zda-li tomu tak je
- **smoke testování**
 - Krátký test zjišťující stabilitu sestavení programu a jestli je připraven pro další fázi testování

- Kontrola jestli jsou nainstalovány a naimplementované všechny potřebné části
- Stavíme-li potrubí, tak do něj pustíme kouř a zjistíme podle jeho úniku, že jsme zapoměli postavit některou část.
- **regresní testování**
 - opakované spouštění testů po provedení úprav
 - zjištění nebyla-li zavedena chyba do již otestované části
- **Zátěžové testování**
 - testování odezvy programu
 - cílem není testovat správnost systému, ale rychlosť odezvy při práci většího počtu uživatelů
- **Usability testy**
 - testy přívětivosti uživatelského rozhraní
 - typicky poskytnutí programu cizímu člověku a sledování jak se v něm dokáže orientovat
- **Validace**
 - Testování shody s požadavky zákazníka (Vytvořili jsme správný produkt?)
 - Typicky formou ručního testování, kdy testerovi dáme seznam požadavků a on podle nich zjišťuje, že je aplikace splňuje.
- **Verifikace**
 - Ověření, že software správně implementoval specifické funkce. (Implementovali jsme produkt správně?)

Mnemotechnická pomůcka: Pojmy Validace a Verifikace se často pletou. Pokud si představíme že u XML je DTD seznam požadavků na strukturu vytvářeného dokumentu, pak lze DTD označit jako požadavky zákazníka. A každý, kdo kdy měl co dočinění s XML, ví, že se proti DTD validuje

Odchylky versus chyby

- **Odchylky** - nesplnění specifikací (zákazník chtěl modré pozadí, ale v aplikaci je zelené)
- **Chyby** - logické, či v datových reprezentacích ($4 + 4 = 16$)

Třídy ekvivalence chyb

Matematická vsuvka

Pojem ekvivalence je v matematice používán pro binární relaci, která množinu, na které je definována, rozděluje na vzájemně disjunktní podmnožiny.

Relace ekvivalence určuje jednoznačně rozklad množiny na třídy ekvivalence. Rozkladem zde rozumíme takovou množinu podmnožin, že sjednocením této množiny je původní množina a každé dva prvky množiny podmnožin jsou disjunktní.

Zdroj: [http://cs.wikipedia.org/wiki/Ekvivalence_\(matematika\)](http://cs.wikipedia.org/wiki/Ekvivalence_(matematika))

[[http://cs.wikipedia.org](http://cs.wikipedia.org/wiki/Ekvivalence_(matematika))

Třídy ekvivalence v testování

Třída ekvivalence, neboli množina ekvivalentních případů, je taková množina testových případů, která testuje stejnou věc nebo odhaluje stejnou chybu. Tester / Návrhář testů hledá, jak vhodně seskupit podobné vstupní hodnoty, podobné výstupní hodnoty a podobné činnosti softwaru. Tyto skupiny pak tvoří potřebné třídy ekvivalence. Musí si však dávat pozor, aby počet testových případů nezredukoval pod únosnou mez.

Třídy ekvivalence odvozené ze vstupní podmínky:

Pokud vstupní podmínka specifikuje **rozsah**, jedna třída bude pro platná data a dvě pro neplatná. př.: X = <20,30>

- **platná** pro vstup v rozmezí 20 - 30
- **neplatná** pro vstup menší než 20
- **neplatná** pro vstup větší než 30

Pokud vstupní podmínka požaduje **specifickou hodnotu**, bude jedna platná a jedna neplatná. př.: X = 20

- **platná** pro vstup roven 20
- **neplatná** pro ostatní

Pokud vstupní podmínka specifikuje **množinu dat**, bude jedna platná a jedna neplatná. př.: X = {10,20,30}

- **platná** pro vstup roven 10, 20, nebo 30
- **neplatná** pro ostatní

Pokud vstupní podmínka je **booleovská**, bude jedna platná a jedna neplatná. př.: X=True

- **platná** platná pro vstup True
- **neplatná** pro vstup False

Hraniční testování

Hraniční testování znamená, že se testům nepředkládají jen nějaka obyčejná data z možné množiny vstupních dat, ale testuje se na okrajových hodnotách, kde dochází často k chybám (kdo by neznal chybu Index Out Of Bounds).

Vybírájí se data blízká hranicím tříd ekvivalence (viz. výše).

Uvažují se jak vstupní podmínky tak i výstupní podmínky.

1. Pokud vstupní podmínka specifikuje rozsah od A do B, testovací data mají být A, B, bezprostředně nad a pod A a B.
2. Pokud vstupní podmínka specifikuje počet hodnot, testovací data mají obsahovat minimální a maximální hodnotu.

Body 1 a 2 aplikuj pro výstupní specifikace. (např. výstup je tabulka hodnot, navrhněme test, který by produkoval maximální minimální hodnoty tabulky.)

Pokud vnitřní datová struktura má předepsané ohrazení (např. limit položek), navrhni test ohrazení.

Typy odchylek a defektů

- Chyby uživatelského rozhranní
- Chyby omezení
 - Chyby zpracování vyjímek
 - Chyby hraničních podmínek
 - Výpočetní chyby
- Procesní chyby
 - Chyby stavů (např.: špatná inicializace dat)
 - Chyby řízení (program provede špatný další krok)
 - Chyba řízení nastane, pokud program provede chybný příští krok.
 - Extrémní chyba nastane, pokud se program zastaví či naopak vymkne řízení
 - Chyby souběhu (paralelní běh vláken)
- Chyby vedení
 - Chyby Hardware - posílaná chybná data, přístup na neexistující zařízení, atd.
 - Dokumentace - Slabá dokumentace
- Chyby požadavků
 - Neúplné, nejednoznačné, či vzájemně si odpovídající
- Strukturální chyby
 - Chyby v řídících sekvencích
 - příkazy GOTO, kód ala špagety, kód ala pačinko,
 - většina chyb řízení (v novém kódu) se dá snadno testovat a je chycena během testování jednotek,
 - neupravený starý kód může mít řadu chyb v řídícím toku,
 - stlačování za účelem kratšího prováděcího času nebo menšího nároku na paměť je špatná praktika.
 - Chyby při vyhodnocování aritmetických operací
 - Špatné logické operátory
- Datové chyby
- Chyby implementace
- Srovnávací testy

Zdroje

- Skripta Y36SI2
- Materiály Y33TSW (Testování a kvalita software) - <http://rage.czweb.org/y33tsw.htm> [<http://rage.czweb.org/y33tsw.htm>] a http://labe.felk.cvut.cz/~marikr/teaching/Y33TSW_08/Y33TSW.htm [http://labe.felk.cvut.cz/~marikr/teaching/Y33TSW_08/Y33TSW.htm]
- stručný popis technik testování - <http://www.saenik.com/modules.php?name=News&file=article&sid=4> [<http://www.saenik.com/modules.php?name=News&file=article&sid=4>]
- další popisy technik testování - <http://ladyloba.blog.cz/0801/techniky-testovani-sw> [<http://ladyloba.blog.cz/0801/techniky-testovani-sw>]

si/si7.txt · Poslední úprava: 2009/05/21 15:52 autor: Kabakov

Otázka 08 - Y36SI2

Zadání: Nástroje a postupy testování, návrh testovacích plánů. Řízení procesu testování, monitorování, analýza chyb.

slovnicek pojmu

- **Antibugging** - předvídání chybného chování při návrhu cesty pro zpracování chybného stavu, které ukončují výpočet s chybovým hlášením. Je třeba testovat, zda nenastane některý z následujících případů:
 1. Popis chyb není příliš inteligentní.
 2. Oznámené chyby nekorespondují se skutečnými.
 3. Chyba způsobí zásah do systému dříve, než je ošetřena.
 4. Nesprávné ošetření výjimečných podmínek.
 5. Popis chyby neposkytuje dostatečné informace k lokalizaci chyby.
- **whitebox testing**
 - používá informace o vnitřní struktuře testovaného modulu a znalosti zdrojového kódu
 - potřebné informace:
 - Testovaný modul
 - Soubor specifikací výstupu pro specifický vstup.
 - Počet větvění ve zdrojovém programu.
 - Počet a typ cyklů ve zdrojovém programu.
 - Počet volání funkcí ve zdrojovém programu.
 - Počet nepodmíněných skoků ve zdrojovém programu.
 - Použití rekurze, které je třeba věnovat zvláštní pozornost.
 - prověřuje následující programové struktury:
 - všechny samostatné cesty uvnitř modulu budou provedeny alespoň jednou
 - všechny podmínky se projdou větví s hodnotou ANO (true) i větví s hodnotou NE (false)
 - projde všechny cykly
 - prověří vnitřní datové struktury
- **blackbox testing**
 - testování jednotlivých modulů bez znalosti vnitřní logiky
 - víme co by měly moduly produkovat za výsledky a ověřujeme, zda-li tomu tak je
- **smoke testování**
 - Krátký test zjišťující stabilitu sestavení programu a jestli je připraven pro další fázi testování
 - Kontrola jestli jsou nainstalovány a naimplementované všechny potřebné části
 - Stavíme-li potrubí, tak do něj pustíme kouř a zjistíme podle jeho úniku, že jsme zapoměli postavit některou část.
- **Regresní testování** - Testuje se, zda nenastal vedlejší efekt přidáním nového modulu (zavlečené chyby) - opakováním předchozích testů.
 - Když napíšeme nový kus kódu, tak napíšeme většinou test na novou funkci

(progresní test), tím sice ověříme jestli funguje nová funkce, ale nevíme, jestli jsme tím neovlivnili starou funkcionalitu. Tudíž je potřeba spustit i všechny předchozí testy na staré funkce.

- V praxi pro každé sestavení programu (build) proběhnou všechny testy (třeba přes noc)

Nástroje

- **Seznam požadavků**
 - Základním předpokladem pro testování je seznam požadavků na program.
- **Cause and Effect diagram (rybí kost)**
 - Při výskytu chyby umožňuje zobrazit možné příčiny
 - viz. sekce Monitorování a analýza chyb (níže)
- **Bug tracking system**
 - Systém pro sledování zadávání objevených chyb a distribuci odpovědným osobám za jejich vyřešení.
- **IDE**
 - IDE rozhraní často umožňuje správu a automatické spouštění testů
- **Automatické testy**
 - Dumb monkey (hloupá opice)
 - Náhodné generování vstupů pro aplikaci
 - Smart monkey (chytrá opice)
 - Generování vstupů aplikace dle určitých pravidel (stavové automaty, statistické modely)
 - kombinace
 - Smart monkey ovládaná Dumb monkey - náhodně kombinuje posloupnost rozumných testů
- **Správa paměti**
 - Testování správy operační paměti (memory leaky, přístup na nealokovaná místa, atd.)
 - Například nástroj Rational Purify Plus
- **Využití metod v programu**
 - Otestování jak moc jsou které metody využívány a kolik procesorového času spotřebují
 - Vhodné pro případnou optimalizaci
 - Například nástroj Rational Quantify
- **Code coverage (pokrytí kódu testy)**
 - Zjištění jak velká část programového kódu je pokryta našimi testy

Postupy testování

Jednotkové testy (unit tests)

- Testování zaměřené na verifikaci malých jednotek softwarového návrhu - modulů.
- prováděno metodami **white-box** testování paralelně pro více modulů

- testuje:
 - rozhraní
 - lokální datové struktury (zda dočasně uložená data zachovávají svou integritu)
 - okrajové podmínky (zda modul pracuje správně na hranicích, omezujících výpočet)
 - nezávislé cesty (zaručující, že každý příkaz bude proveden alespoň jednou)
 - Není možné otestovat všechny možné cesty průchodu programem (výpočetně velmi náročné)
 - Zvolení minimálního počtu cest, které dohromady pokryjí celý kód
 - cesty pro zpracování chyb

Integrační testy

Testování interakce jednotlivých modulů jako celku.

Prováděno metodami **black-box** testing.

2 přístupy:

- velký třesk
 - vše se spojí rovnou a otestuje
 - v záplavě chyb se většinou nedá orientovat ⇒ nepraktické
- inkrementální integrace
 - opak velkého třesku
 - integrace shora-dolu
 - integrace zdola-nahoru

Integrace shora-dolů

Napíše se základní modul aplikace a podřízené moduly jsou nahrazeny maketami (náhražky budoucích modulů vracející testovací data), které jsou postupně nahrazovány moduly.

Při každém nově přidaném modulu dojde k otestování, jestli nedošlo k zavlečení nové chyby.

Nevýhodou je když mají makety poskytovat složitější operace.

Integrace zdola-nahoru

Napíší se jednotlivé dílčí moduly, které jsou postupně pospojovány pomocí řídících modulů (driverů).

Po otestování se drivery odstraní a shluk modulů se připojí ke kompletní struktuře.

Validační testy

Testování jestli program splňuje požadavky zákazníka.

Alfa testování

provádí zákazník v řízeném prostředí dodavatele ("programátor se mu dívá přes rameno")

Beta testování

se provádí u jednoho nebo více zákazníků. Vývojář není přítomen. Zkouší se to v "živých" podmínkách. Zákazník zapisuje všechny problémy (skutečné i imaginární) a určitých intervalech je posílá dodavateli.

Systémové testy

Systémové testování je série různých testů, která prověřuje celý počítačový systém

- Testování obnovy (Recovery testing)
 - Byly poruchy ošetřeny v řádném čase?
 - Byla obnovena potřebná data?
 - Jak proběhla reinicializace?
- Bezpečnostní testování (Security testing)
 - Tester spluluje roli útočníka
 - Je cena na proniknutí do systému dostatečně vysoká?
- Zátěžové testování (Stress testing)
 - Jak dlouho mohu systém namáhat, než se zhroutí?

Testování GUI

série obecných testů:

- Bude okno správně otevřeno klávesovými příkazy či příkazy menu?
- Může být okno zmenšeno/zvětšeno, přesunuto, schováno?
- Jsou všechna data uvnitř okna dostupná myší, funkční klávesou, šípkami a klávesnicí?
- Je okno správně obnovené, když se překryje a znova zavolá?
- Jsou všechny funkce v okně dostupné, když je potřeba?
- Jsou všechny funkce v okně spustitelné?
- Jsou všechny roletová menu, nástrojové lišty, dialogová okénka, knoflíky, ikony a ostatní řídicí prvky správně zobrazeny?
- Je aktivní okno správně vysvícené?
- Jsou všechny roletová menu, nástrojové lišty, dialogová okénka, knoflíky, ikony a ostatní řídicí prvky správně zobrazeny?
- Je aktivní okno správně vysvícené?
- Pokud se užívá multitasking (zpracování několika úloh současně), jsou všechna okna správně a ve správný čas aktualizovaná?
- Nezpůsobí vícenásobné kliknutí myši, nebo kliknutí mimo okno neočekávaný vedlejší efekt?
- Objevují se správně zvuková nebo barevná upozornění?
- Zavře se okno správně?
- Je zobrazena správná příkazová lišta v odpovídajícím kontextu?
- Jsou správně zobrazeny doplňující údaje (např. čas)?

- Pracuje správně stahování menu?
- Jsou všechny funkce v menu a případné podfunkce správně zobrazeny?
- Jsou všechny funkce v menu správně ovladatelné myší?
- Dají se příslušné funkce spustit správně odpovídajícím klávesovým příkazem?
- Jsou funkce správně vysvíceny nebo neosvíceny podle daného kontextu?
- Dělá každá funkce, co má?
- Jsou názvy funkcí názorné a srozumitelné (self-explanatory)?
- Je pro každou položku menu dostupný help a je kontextově závislý?
- Jsou operace myši správně rozpoznány?
- Pokud je požadované vícenásobné kliknutí, je správně rozpoznáno?
- Je správně zobrazen a změněn kurzor v daném kontextu?

Návrh testovacích plánů

Testování probíhá po celou dobu vývoje software (v agilních metodikách programování se dokonce nejdříve napíše test a teprve poté se napiše metoda, která jím má projít) .

Vývojáři v průběhu vývoje průběžně testují svůj kód pomocí Unit testů. V pokročilejší fázi vývoje nastupuje nezávislá skupina testerů, kteří se vrhnou na otestování celku.

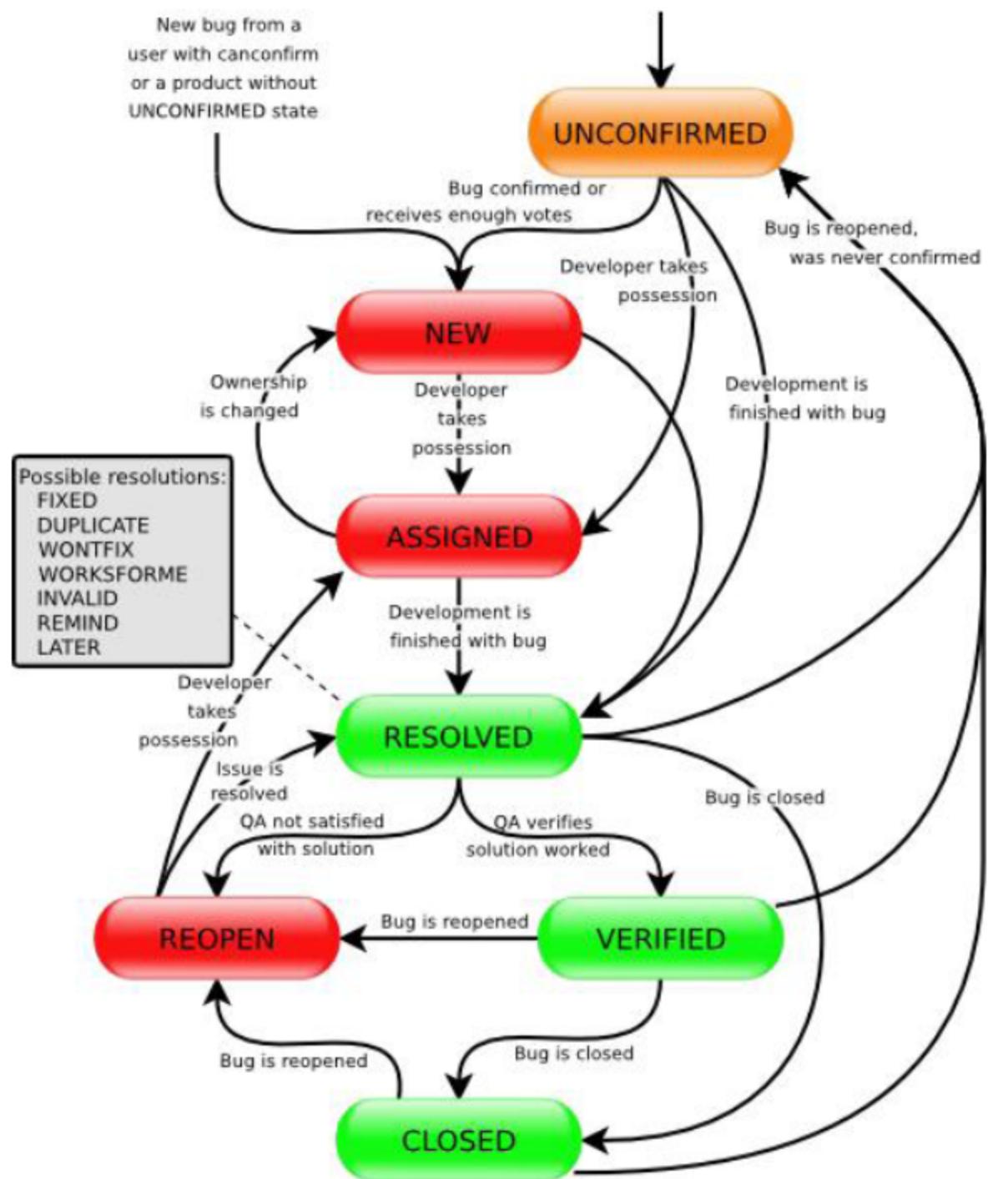
Hierarchie testovaných částí pro objektově orientované programovací jazyky:

- třída
 - Unit testy
- interakce mezi třídami
 - Integrační testy
- subsystém
 - Integrační testy
- systém
 - Validační testy
 - Systémové testy
- Testování uživatelské dokumentace a helpu
 - Popisuje dokumentace správně provedení změny módu?
 - Je dobře popsána posloupnost interaktivní komunikace?
 - Jsou uvedené příklady správné?
 - Je terminologie, popisy menu, a systémových odezv popsána konsistentně s programem?
 - Je relativně snadné nalézt v dokumentaci případnou radu, návod?
 - Je popsáno chování v případě problému? (troubleshooting)
 - Má dokumentace správný obsah a rejstřík?
 - Je grafická úprava přehledná (a není matoucí)?
 - Jsou všechna chybová hlášení podrobně popsána v dokumentaci?
 - Jsou hypertextové odkazy správné?

Řízení procesu testování

Životní cyklus chyby (Bugzilla)

1. Při nahlášení chyby (testerem, či zákazníkem) se musí ověřit, jestli se skutečně jedná o chybu
2. Pokud ano, vytvoří se nový ticket popisující chybu (**New**)
3. Ticket se přiřadí řešiteli (**Assigned**)
4. Řešitel problém vyřeší, nebo předá někomu jinému (když neví co s tím, nebo se to k němu dostalo omylem)
5. Po vyřešení se ticket přesune do stavu vyřešený (**Resolved**), kde je potřeba ověřit, jestli je problém skutečně odstraněn.
6. Pokud ano, je označen jako ověřený (**Verified**), jinak znova otevřen a vrácen řešiteli (**Reopen**)
7. Pokud je vše v pořádku je ticket uzavřen a problém je považován za vyřízený (**Closed**)



Kdy ukončit testování?

Jelikož v každém kódu lze nalézt nějaká chyba, tak odladit program do odstranění všech chyb je prakticky nemožné. Na počátku testování máme obvykle velké množství chyb a jejich postupným odstraňováním bude počet nově nalezených chyb klesat. Je tedy potřeba nalézt hranici, kde již množství nově nalezených chyb bude zanedbatelné a vypustit program do světa (a případné další kritické chyby řešit pomocí patchů)

Například: Logarithmic Poisson execution-time model: (Dle mého názoru tento vzoreček nikdo u stánic chtít nebude)

$$f(t) = (1/p) \ln (I_0 pt + 1)$$

kde $f(t)$ je kumulativní počet poruch, které se očekávají, že nastanou po testování softwaru po určitou dobu t ,

I_0 je počáteční softwarová intenzita poruch (poruchy za časovou jednotku) na začátku testování

p je exponenciální snížení intenzity poruch po odhalení chyb a jejich odstranění

Okamžitá intenzita poruch $I(t)$:

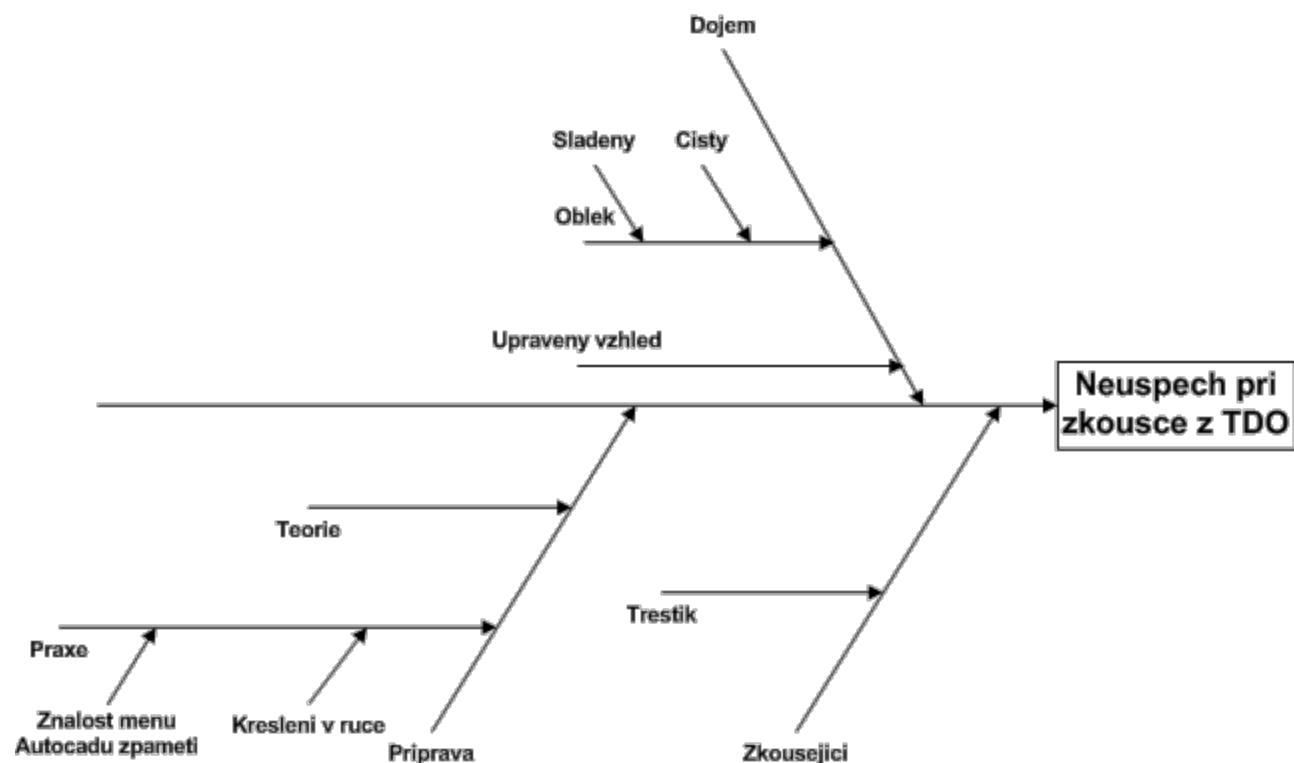
$$I(t) = I_0 / (I_0 pt + 1)$$

Monitorování a analýza chyb

Pokud test odhalí chybu, neznamená to nutně, že je chyba v programu, může být chybně navržen samotný test.

Cause and effect diagram (Rybí kost)

Diagram problému a možných příčin. Příčiny se hledají formou brainstormingu a hledá se nejpravděpodobnější.



Ladění

Oprava nesouhlasu mezi očekávaným a skutečným výstupem spočívá v nalezení příčiny daného symptomu.

Přístupy k ladění

Při ladění programů jde o kombinaci systematického vyhodnocování, intuice a štěstí. Existuje několik různých přístupů:

- hrubá síla - nejčastější metoda a nejméně efektivní
- backtracking - postupujeme zpětně (manuálně) od daného symptomu po všech cestách a hledáme chybu (To je možné jen pro malé programy).
- eliminace příčin - návrh testů má eliminovat možné příčiny.

Všechny přístupy mohou být podpořeny ladícími nástroji (debugging tools) :

- **trasování** - zastavení programu za chodu a sledování aktuálního stavu proměnných a vykonávaných metod, případně vypisování hodnot proměnných na výstup.
- **automatické generátory testů** - některé testy se dávají vygenerovat bez zásahu vývojáře testera (smart/dumb monkey, getters a setters, atd.)
- **výpis paměti** - aktuální stav proměnných a alokovaného místa
- **ladící nástroje překladačů**

Často stačí vysvětlit problém jinému kolegovi (jiný pohled) a chyba se objeví.

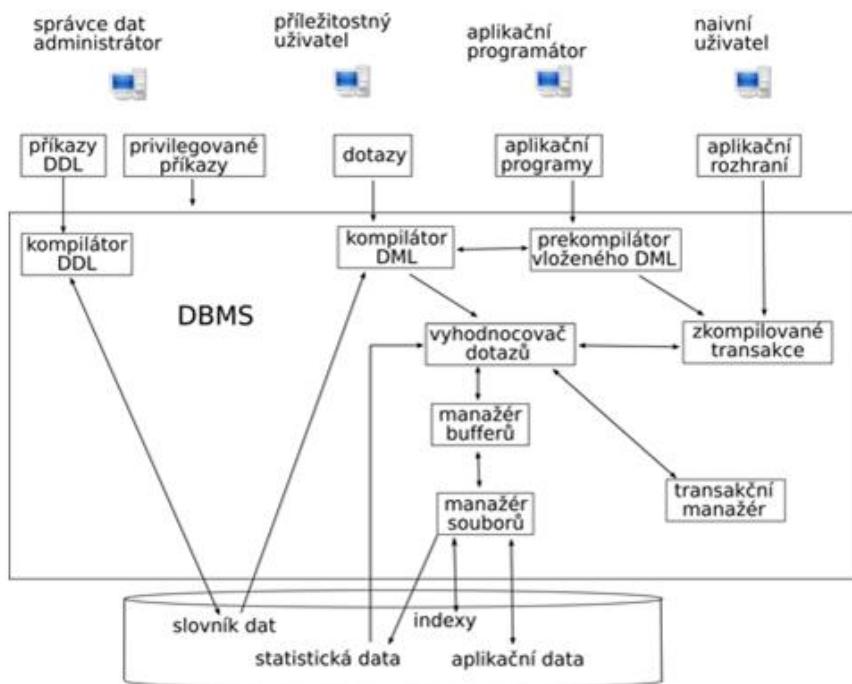
Zdroje

- Skripta Y36SI2
- Materiály Y36SISI3
 - přednáška č. 2 - Systém pro správu požadavků - <http://service.felk.cvut.cz/courses/Y36SI3/files/2SI3.zip> [<http://service.felk.cvut.cz/courses/Y36SI3/files/2SI3.zip>]
 - přednáška č. 13 - Automatické testování - <http://service.felk.cvut.cz/courses/Y36SI3/files/13SI3.zip> [<http://service.felk.cvut.cz/courses/Y36SI3/files/13SI3.zip>]
- Materiály Y33TSW (Testování a kvalita software) - <http://rage.czweb.org/y33tsw.htm> [<http://rage.czweb.org/y33tsw.htm>] a http://labe.felk.cvut.cz/~marikr/teaching/Y33TSW_08/Y33TSW.htm [http://labe.felk.cvut.cz/~marikr/teaching/Y33TSW_08/Y33TSW.htm]

Otázka 09 - Y36DBS

Zadání: Architektura databázového stroje, typičtí uživatelé, práva přístupu k DB objektům (DCL SQL), relační DB struktury sloužící k optimalizaci dotazů - indexy, clustery, indexem organizované tabulky.

Architektura databázového stroje



Příkazy DDL	Příkazy pro definici struktury tabulek/databáze
Privilegované příkazy	Přidělování uživatelských práv, nastavování parametrů SŘBD, apod.
Kompilátor DDL	Zpracovává definici schématu a ukládá jí do slovníku dat.
Kompilátor DML	Kompiluje do programů nižší úrovni přístupu k databázi. Tento kód je připojen ke zbytku aplikáčního programu či předdefinované uživatelské transakci.
Prekompilátor vloženého DML	Zpracovává příkazy DML vyskytující se v hostitelském jazyku.
Vyhodnocovač dotazů	Je zřízen pro dotazy v dotazovacím jazyku. Interpretuje nebo kompiluje dotaz, případně zařizuje jeho optimalizace (za použití např. statistických dat) a komunikuje přímo přes manažér bufferů (a souborů) s databází.
Zkompliované transakce	Zkompliované a uložené transakce.
Manažer souborů	Prostředník mezi operačním systémem a SŘBD v souvislosti s přenosem dat mezi diskem a vyrovnávacími paměti (buffery) ve vnitřní paměti počítače.
Manažer bufferů	Prostředník mezi vyhodnocovačem dotazů a manažerem souborů. Stará se o správné načítání dat ze souborů do bufferů a předávání již načtených dat vyhodnocovači dotazů.
Transakční manažer	Zpracování dotazů a dalších uživatelských transakcí.
Slovík dat	Množina metadat, ve kterých jsou organizovány veškeré definice týkající se logického a fyzického schématu databáze. Z hlediska SŘBD se jedná o skupinu tabulek (a pohledů), do kterých nelze zapisovat. Datový slovník mimo jiné obsahuje: přesnou definici datových prvků, uživatelská jména, role a privilegia, schémata, integrativní omezení, uložené procedury a uložené procedury (stored procedures) a spouštěče.

	(triggers), obecnou strukturu databáze, přidělení místa (space allocation). Datový slovník umožňuje zachovat konzistenci dat v různých tabulkách
Statistická data	Statistická údaje o rozložení dat v tabulce.
Indexy	Datová struktura, která umožňuje rychlejší prohledávání tabulek. (viz níže)
Aplikační data	Data uložená v databázi.

Typičtí uživatelé

Databázový administrátor

- Instalace DBMS
 - často vyžaduje hlubší zásahy do konfigurace OS – nastavení semaforů, velikost sdílené paměti, parametry FS,....
- údržba verzí, „patchování“
- konfigurace DBMS, konfigurace klient-server
- vytvoření a údržba databáze
- zálohování a obnova databáze při pádu
- monitoring a plánování růstu databáze
- audit
- Když databáze nefunguje, je to jeho starost
- Má na starost replikaci dat na sekundární stroje a load-balancing

Administrátor - správce dat

- Obvykle ne jednotlivec, ale odborný útvar (osoby zodpovědné za autorizovaný přístup do databáze). Může obsahovat i projektanty IS, kteří v průběhu života IS modifikují aplikaci v souladu s měnícími se skutečnostmi.
- Vytváří strukturu databáze/tabulek
- Ručí za integritu dat
- Když jsou chyby v datech, je to jejich starost

Příležitostný uživatel

- Vyžadují data z databáze v různých, předem nepředvídatelných souvislostech
- Obvykle ovládají silnější dotazovací jazyk (např. SQL)
- Mění své požadavky v závislosti na svých okamžitých potřebách, tj. aplikují interaktivní či ad hoc dotazy

Aplikační programátor

- Vytváří aplikace pro použití naivními uživateli
- Zná strukturu databáze a píše nad ní dotazy
- Využívá pouze DML, nevytváří strukturu databáze/tabulek

Naivní uživatel

- Využívá aplikační rozhraní vytvořené programátory
- Nemá žádné znalosti o databázích (pro svou práci je nepotřebuje, všechno si 'nakliká' v aplikaci)

Práva přístupu k DB objektům (DCL SQL)

- DCL = Data Control Language
 - Příkazy Grant, Revoke
- Umožňuje řízení přístupu k databázovým objektům pro jednotlivé uživatele databáze
- Identifikace uživatele se skládá z uživatelského jména, hesla a volitelně také z místa připojení (např. je možné rozlišit uživatel user123 připojeného z lokálního počítače a dát mu vyšší práva, než když se přihlásí odněkud jinud)
- Přidělování práv k jednotlivým objektům záleží na konkrétní databázi, ale obecně se dá nastavit přístup typu „číst a zapisovat do tabulky x, číst z tabulky y,...“
 - Např. MySQL umožňuje nastavovat práva i pro jednotlivé sloupce tabulky

Root bývá obvykle nejvyšší správce a může přidělovat libovolná oprávnění prakticky komukoliv (pozor - nemusí platit vždy, např. v MySQL je možné omezit mu práva). Jedním z možných práv, která může přidělit je také právo přidělovat privilegia dalším uživatelům. Takovýto uživatel smí dále přiřazovat jiným uživatelům práva, která sám vlastní. Manuál MySQL v tomto bodě doporučuje dávat pozor na lidi, kterým tato práva přidělujeme – je to z toho, důvodu, že dva uživatelé s různými právy (a možností tato práva přidělovat) si mohou navzájem tyto své pravomoci doplnit a získat tak oprávnění, jež by mít neměli.

V hierarchii uživatelských práv se MySQL odchyluje od standardu SQL. Mějme tedy např. uživatele user1 s nějakými právy (a umožněme mu tato práva dále distribuovat). Ten přidělí tato práva uživateli user2. Pokud v tomto bodě odebere libovolná práva uživateli user1, tak na rozdíl od standardu SQL user2 tyto změny nijak nepocítí.

Konkrétní ukázky přidělování a odebírání oprávnění

Pokud jsme tedy přihlášeni jako uživatel, který má dostatečná oprávnění, můžeme provádět např.:

```
GRANT ALL ON *.* TO 'user1'@'localhost';
```

Tento příkaz přiřadí veškerá práva uživateli 'user1', ovšem ten je nesmí přidělovat nikomu dalšímu. Takového efektu bychom dosáhli příkazem:

```
GRANT ALL ON *.* TO 'user1'@'localhost' WITH GRANT OPTION;
```

MySQL samozřejmě umožňuje mnohem jemnější rozlišení práv, než pouhé ALL:

```
GRANT SELECT, INSERT, UPDATE ON myDatabase.* TO 'user';
```

Tímto příkazem umožníme uživateli user číst, zapisovat a modifikovat data v libovolné tabulce databáze myDatabase. Samozřejmě je možné určovat práva i pro jednotlivé tabulky:

```
GRANT SELECT, INSERT, UPDATE ON myDatabase.myTable TO 'user';
```

V praxi jsem se s tím sice nikdy nesetkal, ale MySQL umožňuje definovat práva i na jednotlivé sloupce tabulky:

```
GRANT SELECT(col1), INSERT(col2, col3), UPDATE(col2) ON myDatabase.myTable TO 'user';
```

Nyní může uživatel user číst data ze sloupce col1, zapisovat nová data do sloupců col2 a col3 a měnit data ve sloupci col2 v tabulce myTable v databázi myDatabase. Odebírání práv funguje velice obdobně:

```
REVOKE ALL ON myDatabase.* FROM 'user';
```

Tento kód odebere veškerá práva všech tabulek v databázi myDatabase uživateli user.

Relační DB struktury sloužící k optimalizaci dotazů

Indexy

Index je databázová konstrukce, sloužící ke zrychlení vyhledávání a dotazování a definování unikátních hodnot nad databázovými tabulkami. Indexová většinou vypadá jako datová struktura, která obsahuje hodnotu klíče, nad níž je index vytvořen a odkaz na záznam (řádek), ke kterému patří.

Databázové indexy (či indexekvenční moduly) dělíme do různých druhů podle toho, co chceme při přístupech k primárním datům příslušné databázové tabulky optimalizovat. Označení druhů indexů se může různit, nejčastěji se používají tyto hlavní druhy:

Primary

Je tvořen většinou jedním (ale může být i více) sloupcem, který obsahuje primární klíč. Jedná se o speciální druh indexu, který se může v každé tabulce vyskytovat nanejvýš jednou. Je definován sloupcem (nebo sloupcí), které svou hodnotou jednoznačně identifikují záznam v tabulce. Databázový stroj musí zajistit, aby nebylo možné vložit řádek se stejnými daty (ve sloupcích určujících primary index), která už v tabulce existují. Jedná se tedy o speciální případ Unique indexu. Je zařitou konvencí vytvářet primary index nad sloupcem nazvaným Id s celočíselným datovým typem.

Unique

Je velice podobný primary indexu (primary index je vlastně podtypem unique) s tím rozdílem, že se v tabulce může vyskytovat vícekrát.

Index (secondary, vedlejší)

Definicí jednoho či více indexů tohoto typu v tabulce zajišťujeme optimalizaci vyhledávání podle dalších sloupců, mimo primární nebo unikátní indexy. Databázový server vytvoří a nadále udržuje vnitřní konstrukci odkazů na řádky tabulky, jež poskytuje uspořádání podle příslušných hodnot ve sloupci, k němuž je index logicky vázán (podle hodnot sekundárního klíče). Udržování takto uspořádané konstrukce urychluje vyhledávání záznamů v databázi (je možno použít některé matematické interpolační numerické metody), logické či fyzické řazení záznamů jakož i jiné další datové operace s tabulkou, jež se mají provést na podmnožině záznamů z tabulky vymezené podmínkou položenou na hodnoty v sekundárním klíči. Na rozdíl od předchozích indexů PRIMARY a UNIQUE lze do tabulky vkládat záznamy, které nejsou v sekundárním indexu unikátní. U některých databázových systémů se může jednat i o sloupec tzv. fiktivní, tedy sloupce odvozené respektive vypočtené z hodnot sloupců fyzických resp. uložených.

Full-text

Používá se pro optimalizaci full-textového vyhledávání v textových sloupcích. Implementace závisí na zvoleném databázovém stroji, jednou z metod je např. udržování statistiky slov, které se vyskytují ve sloupcích s full-text indexem.

Indexy - implementace

Bitmap index

Tento druh indexu je vhodný pro sloupce, kde se vyskytuje relativně málo hodnot, ale ve velkém množství opakování. Krásným příkladem je např. tabulka uživatelů, o kterých si chceme pamatovat jejich pohlaví. Takový sloupec nabývá buď hodnot muž x žena.

Pro každou z těchto dvou hodnot je vytvořena bitmapa (pole bitů), která obsahuje informace o tom, které řádky obsahují tyto hodnoty a její nad ní možné provádět bitové operace. Výhodou tohoto přístupu je relativně malá datová

náročnost na uchovávání indexu, protože každý záznam v tabulce zabere 1bit * počet hodnot, které sloupec nabývá. Ukázkový index pohlaví by tak v tabulce s více než osmi miliony uživatelů zabral 2 MiB. I přes relativně malou náročnost na data je možné index dále komprimovat, typicky pomocí nějaké verze run-length kódování, které jsou velice nenáročné na kompresi/dekompresi. Zde se nejčastěji používá Byte-aligned Bitmap Code (BBC) a Word-aligned Hybrid Code (WAH), které navíc umožňují provádět bitové operace přímo nad komprimovanými daty.

BBC uchovává svoje bitmapy v bytech, zatímco WAH v wordech, které lépe vyhovují dnešním procesorům a dosahují větších výkonu. Dle zveřejněných testů by pak WAH mělo (na reálných i syntetických datech) zabírat zhruba o 50% více datového prostoru, ale dosahovat až 12x rychlejších výsledků než BBC. Obě metody jsou však silně závislé na seřazení dat v tabulce. Jednoduché lexikální seřazení může změnit index až 9x a výrazně zvýšit výkon. Čím více dat

Bitmap index

Identifier	Gender	Bitmaps	
		F	M
1	Female	1	0
2	Male	0	1
3	Male	0	1

tabulka obsahuje, tím více roste nutnost správného řazení.

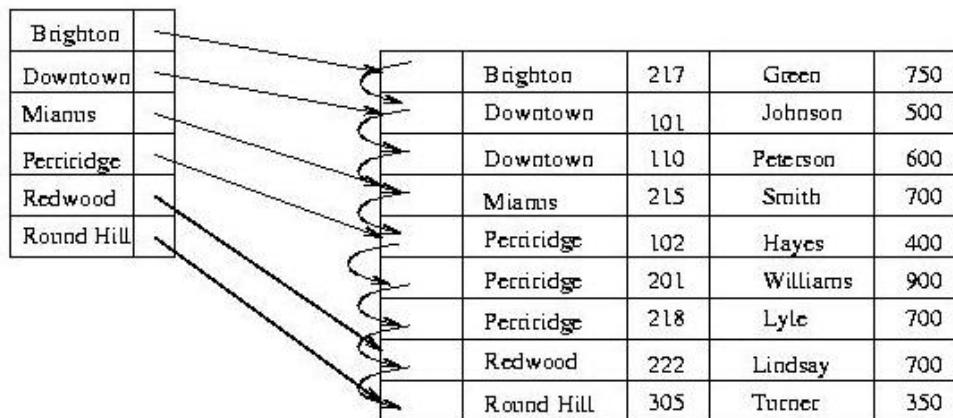
Dense a Sparse index

Dense index obsahuje všechny hodnoty klíče, které se v tabulce vyskytují spolu s odkazem na relevantní záznamy. Při hledání se pak najde v indexu požadovaný klíč a z něho se přečte umístění požadovaných záznamů.

Sparse index neuchovává záznamy o všech klíčích, ale pouze o určité podmnožině dostupných klíčů. Vyhledávání probíhá tak, že se najde největší klíč, který menší nebo roven tomu hledanému a od tohoto záznamu se sekvenčně prohledává tabulka, než jsou nalezeny požadované záznamy.

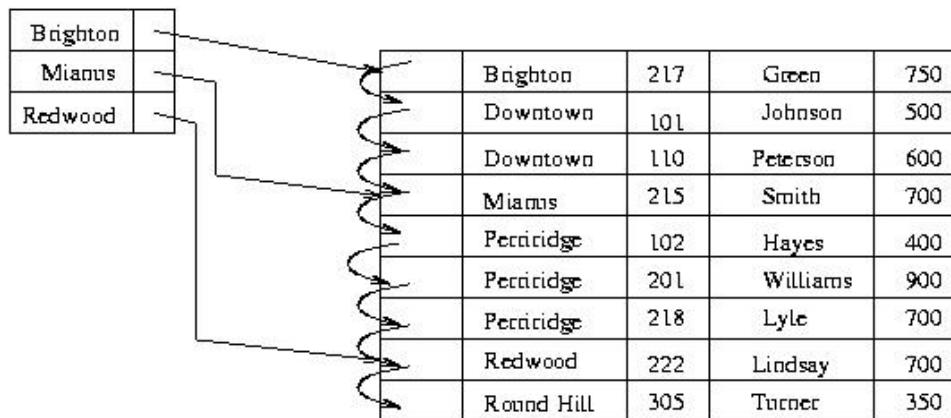
Dense indexy jsou obecně o něco rychlejší, ale zabírají více diskového prostoru, než sparse indexy, ale mají větší režii

Dense index



pro přidávání a mazání.

Sparse index



Clustery

V rámci optimalizace rychlosti dotazů umožňují některé databázové enginy ukládat tabulky do tzv. clusterů. Jedná se o formu fyzického uložení na disk, kdy jsou tabulky se souvisejícími daty (např. spojení přes reference) uloženy u sebe. Tím se zrychlí např. dotazy tyto JOIN na tyto tabulky. Pro uživatele (tedy aplikáčního programátora) se však nadále chovají zcela transparentně jako normální tabulky.

Shluk (Cluster)

TITUL_ID	KOPIE_ID	DATUM
148590	00001	YYYY
148969	00001	YYYY
148969	00002	YYYY
148590	00002	YYYY
155791	00001	YYYY
148590	00003	YYYY

TITUL_ID	NAZEV	ROK_VYROBY
148590	xxx1	1997
148969	XXX2	2001
155791	XXX3	2005

Cluster Key: TITUL_ID

148590 xxx1 1997

00001 YYYY

00002 YYYY

00003 YYYY

148969 xxx2 2001

00001 YYYY

00002 YYYY

155791 xxx3 2005

00001 YYYY

**Samostatné tabulky s vazbou
přes cizí klíč**

Tabulky ve shluku

Indexem organizované tabulky

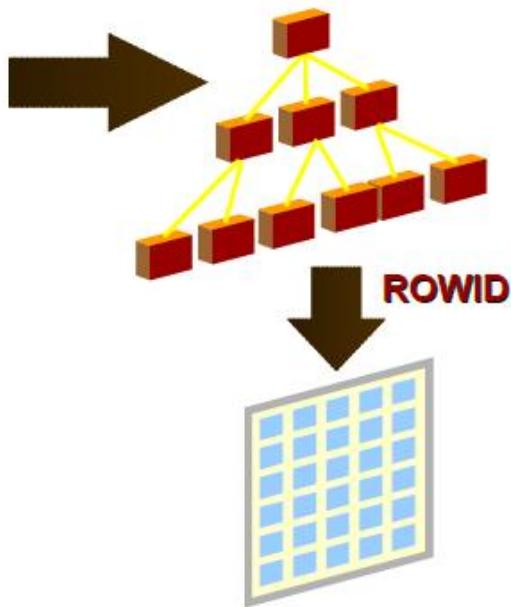
Standardní tabulka ukládají řádky do svých datových bloků přibližně následovně (může se lišit dle implementace): Každý řádek má neměnnou pozici na datovém úložišti. V momentě, kdy je dostane svoji pozici a na té zůstane, dokud nebude smazán a to i v případě, že bude např. přesunut a doplněn o další data - v takovém případě na původním místě alespoň část dat, podle kterých systém dokáže dohledat zbytek. Index v normální tabulce obsahuje data a rowid (interní identifikátor řádku), podle kterého se dohledá správný datový blok.

V indexově organizované tabulce jsou data na úložišti organizována jako B-tree index postavený z primárních klíčů. V takovéto tabulce nemá řádek stejnou pozici a v čase se může přesouvat, aby bylo zachováno správné seřazení B-tree. Výhody

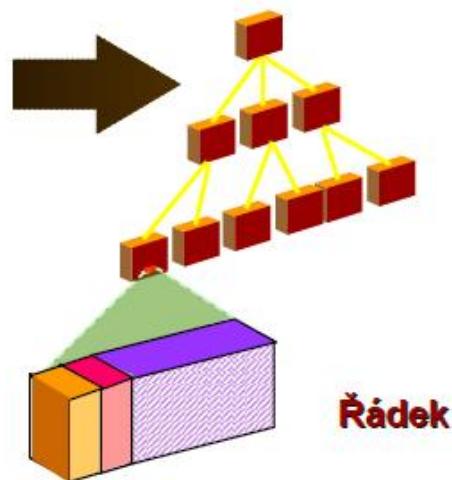
- Rychlý přístup k datům při dotazech na přesnou hodnotu nebo rozsah primárního klíče (např. WHERE id = 5, WHERE BETWEEN 10 AND 20, ...)
- Pro 24x7 aplikace: Při reorganizaci dat na datovém úložišti (např. defragmentace) není třeba znova vytvářet index

Indexově organizovaná tabulka

Heap tabulka s indexem



Indexově organizovaná tabulka



si/si9.txt · Poslední úprava: 2009/06/16 02:22 autor: Staral

Otázka 10 - Y36PJC

Zadání: Typ reference, přetěžování funkcí, typ reference, vstup a výstup, třídy, staticky vázané metody, dědění, dynamicky vázané metody, abstraktní třídy, polymorfní datové struktury.

Slovníček pojmu

- **Procedura:** funkce s prázdnou návratovou hodnotou
- **Skutečný parametr:** Výraz použitý jako parametr při volání funkce.
- **Staticky alokovaná proměnná:** obvyklá proměnná.
 - o vrácení paměti se stará program
 - **int x = 2;**
- **Dynamicky alokovaná proměnná:** proměnná vytvořená ručně operátorem **new** (nebo malloc...) za běhu programu.
 - je třeba dát pozor na uvolnění paměti (delete, free...)
 - **int * x;**
- **VMT:** Tabulka virtuálních metod (*Virtual Method Table*).

Typ ukazatel

- Ukazatel (Pointer) ukazuje na místo v paměti. Jeho hodnotou je adresa v paměti.
- **Doménový typ** ukazatele je to, na co ukazuje (datový typ, objekt, funkce, ...).

Příklady deklarací proměnných typu ukazatel:

```
int *p; /* p je ukazatel na int */
char *q; /* q je ukazatel na char */
```

Související operace

Reference (adresa)

`& X` kde `X` je označení datového objektu nebo funkce

```
Je-li X typu T, pak &X je typu T*, tzn. ukazatel na T
```

Dereference (zpřístupnění objektu, na který ukazuje ukazatel)

`* X` kde `X` je výraz typu ukazatel

```
Je-li X typu T*, pak *X je typu T
```

Příklady

```
int i, *pi = &i; /* pi obsahuje adresu i */
char c, *pc = &c; /* pc obsahuje adresu c */
*pi = 25; /* do i se uloží 25 */
*pc = 'a'; /* do c se uloží 'a' */

char str[] = "Hello world!";
char *ptr = &str[2];
*ptr = 'l'; /* změní se str[2] */
```

Ukazatelová aritmetika

Ukazatele mohou být operandy sčítání, odčítání a všech relačních operátorů.

Dovolené kombinace a typ výsledku:

```
T* + int -> T*
T* - int -> T*
T* - T* -> int
T* <relační operátor> T* -> int
```

Příčtení n k ukazateli typu T* znamená jeho změnu o n-násobek délky typu T. Podobně odečtení a rozdíl ukazatelů

Příklady

```
int a[10], *p = &a[0];
*(p + 3) = 10; /* do a[3] se uloží 10 */

/* vynulování pole a */
for (p = &a[0]; p <= &a[9]; p++) *p = 0;

/* nebo vynulování pole a */
for (p = &a[0]; p <= &a[9]; *p++ = 0);
```

Pole a ukazatele

Jméno pole prvků typu T = konstantní ukazatel typu T* ukazující na prvek s indexem 0

```
int a[10], *pa, i;
pa = a; totéž co pa = &a[0]
*(a + 2) = 3; totéž co a[2] = 3
*(a + i) = 4; totéž co a[i] = 4;

for (pa = a; pa < a + 10; *pa++ = 0); /* smaže pole */
a++; /* chyba, protože a je konstantní ukazatel ( tj. nelze měnit místo kam ukazuje ) */
```

Upřesnění indexace

X [Y] kde X je typu ukazatel na T a Y typu int, výsledek je typu T

```
Výraz X [ Y ] je ekvivalentní s *( ( X ) + ( Y ) )
pa[3] = 10; totéž co *(pa + 3) = 10;
```

Přetěžování funkcí (overloading)

Definici funkce tvoří:

- hlavička funkce, což je deklarace funkce nezakončená středníkem
- tělo funkce, což je blok

Přetěžování funkcí je používání stejného názvu funkce pro více různých funkcí. Konkrétní volaná funkce je pak vybrána podle počtu a typu parametrů. (neplatí pro C)

Příklad

```
void f(char x)
{
    cout << x << endl;
}
```

```
void f(int x)
{
    cout << x << endl;
}
```

Párování (matching)

Při volání přetížené funkce se vyvolá ta, jejíž parametry se nejlépe spárují se skutečnými parametry.

1. přesná shoda typů
2. shoda typů po roztažení (promotion)
 - char → int
 - short → int
 - enum → int
 - float → double
3. shoda typů po standardní konverzi
 - int → float
 - float → int
 - int → unsigned
 - ...
 - int → long
4. shoda typů po uživatelské konverzi

Pokud nejlepší párování má více než jedna funkce, ohlásí se chyba

Příklady

```
void f(float, int);
void f(int, float);
f(1,1); // chyba
f(1.2,1.3); // chyba

void f(long);
void f(float);
f(1.1); // chyba
f(1.1F); // f(float)

void f(unsigned);
void f(int);
void f(char);
unsigned char uc;
f(uc); // f(int)
```

Implicitní parametry

Umožňují při volání funkce nezadat všechny parametry. Místo nezadaných hodnot budou použity hodnoty uvedené v hlavičce funkce.

Implicitní hodnoty je nutné zadávat od konce. Vynechávat skutečné parametry je nutné rovněž od konce.

Příklady

Povolené

```
void f(int x, int y=1, int z=3)
{
    cout << "x=" << x << " y=" << y << " z=" << z;
}

main()
```

```
{
    f(10, 20, 30);
    f(10, 20);
    f(10);
}
```

Nesmí se

```
f(int x=1, int y); // chyba

f(int x, int y=10);
f(int x);
f(20); // nejednoznačnost
```

Typ reference

Reference je ukazatel na proměnnou. Jeho výhodou je, že není potřeba dereference při používání, takže mizí spousty ** z kódu. Mezi nevýhody naopak patří nemožnost ukazovat nikam (není možné neukazovat na žádnou proměnnou či NULL). Rovněž je potřeba věnovat větší pozornost kódu, jelikož ** lépe upozorní že se nejedná o obyčejnou proměnnou.

Příklad

```
int a = 0;      // Proměnná typu int
int &b = a;      // "b" je inicializovaná a odkazuje na proměnnou "a"
a = 9;          // se změnou hodnoty "a" se změní i hodnota "b", jelikož "b" na "a" odkazuje
b++;           // po provedení této řádky bude "a" i "b" mít hodnotu 10
```

Použití ve funkci

Mnohem užitečnější je používání referencí při psaní funkcí. Pomocí nich je možné zpřístupnit funkci parametry pro zápis. Skutečným parametrem však musí být proměnná stejného typu, nikoliv literál nebo konstanta.

Příklad

Funkce zapíše do parametrů „pmin“ a „pmax“ minimální a maximální hodnotu z čísel „x“ a „y“. *Příklady oba shrnují zápis funkce a příklad volání, pro spustitelnost potřebují drobné úpravy.*

Realizace pomocí referencí

```
void minmax(int x, int y, int& pmin, int& pmax)
{
    if (x < y)
    {
        pmin = x; // Přiřazení do referované proměnné. Jedná se o okopírování hodnoty nikoliv přiřazení reference. (Pokud bychom tu použily pouze =, bylo by to přiřazení referenčního adresy)
        pmax = y;
    }
    else
    {
        pmin = y;
        pmax = x;
    }
}

int main()
{
    int a, b, min, max;
    ...
    minmax(a, b, min, max); // po této řádce budou proměnné "min" a "max" obsahovat minimum a maximum z hodnot "a" a "b"
    ...
    return 0;
}
```

Realizace pomocí ukazatelů (pro porovnání)

```

void minmax(int x, int y, int *pmin, int *pmax)
{
    if (x < y)
    {
        *pmin = x;
        *pmax = y;
    }
    else
    {
        *pmin = y;
        *pmax = x;
    }
}

int main()
{
    int a, b, min, max; c
    ...
    minmax(a, b, &min, &max);
    ...
    return 0;
}

```

V hlavičce funkce je možné použít klíčové slovo „const“ a potom je parametr předán referencí, nicméně není možné ho měnit (komplier při pokusu hlásí chybu).

```

int f( const int &number )

```

Vstup a výstup

- vstupní a výstupní operace jsou v C a C++ dány knihovnami
- v C++ je možné používat C funkce na vstup a výstup (následuje seznam některých C funkcí)
 - formátované funkce
 - **scanf**: int scanf (const char * format, &promenna1, &promenna2, ...); pro standardní vstup * **fscanf**, **sscanf**: obdobky pro čtení ze souboru a stringu * **printf**: int printf (const char * format, promenna1, promenna2, ...); pro standardní vstup
 - **fprintf**, **sprintf**: obdobky pro čtení ze souboru a stringu
 - neformátované
 - **getc**: přečte znak ze vstupu
 - **ungetc**: odpřečte znak ze vstupu (příští volání getc vrátí tento znak, pozor na opakování volání)
 - **putc**: zapíše znak na výstup
 - **fread**: přečte blok ze souboru
 - **fwrite**: zapíše blok dat do souboru
 - **feof**: test na konec souboru
 - **ferror**: test na chybu
 - **clearerr**: vynuluje chybové indikátory

Následující část kapitoly se zabývá hlavně C++.

- prostředkem pro vstup nebo výstup je **datový proud** (stream)
- vstupní proud standardního vstupu je k dispozici pod názvem **cin**
- výstupní proudy jsou:
 - **cout** na standardní výstup
 - **cerr** na chybový výstup
 - **clog** na výstup do logu
- datový proud (dále jen proud) realizuje tok dat od zdroje ke spotřebiči
 - je-li spotřebičem program, jde o vstupní proud (zdrojem může být soubor nebo řetězec)
 - je-li zdrojem program, jde o výstupní proud (spotřebičem může být soubor nebo řetězec)
- vstup a výstup může být formátovaný nebo neformátovaný

- **formátovaný vstup:** zdrojová data jsou posloupnosti znaků, která se konvertují do vnitřní (binární) reprezentace
- **formátovaný výstup:** zdrojová data ve vnitřní reprezentaci se konvertují na posloupnosti znaků
- **neformátovaný vstup a výstup:** zdrojová data proudí do spotřebiče jako posloupnosti bytů

Třída istream

- Společné vlastnosti všech vstupních proudů definuje třída istream.
- Metody pro neformátovaný vstup:

```
int get()
istream& get(char *, int len, char = '\n')
istream& get(char&)
istream& getline(char *, int len, char = '\n')
istream& ignore(int n = 1, int delim = EOF)
int peek()
istream& putback(char)
istream& read(char *, int)
long tellg()
istream& seekg(long)
```

Příklady

Operátor » pro formátovaný vstup

```
istream& operator>>(istream&, T&) // pro všechny standardní typy T
```

Příklad přetížení operátoru pro uživatelskou třídu

```
istream& operator>>(istream& is, Complex &x)
{
    is >> x.re >> x.im;
    return is;
}
```

Formátovaný vstup lze řídit manipulátory

```
int n;
...
cin >> oct >> n; // vstup v osmičkové soustavě
```

Chyby

Nastane-li při čtení ze vstupního proudu chyba, další vstupní operace s daným proudem se až do vymazání příznaku chyby ignorují.

```
bool fail() // kontroluje zda-li se operace zdařila
void clear() // odstraní příznak chyby
```

Příklad čtení celého čísla s kontrolou chyby:

```
int ctiInt(istream& is)
{
    int x;
    bool chyba = false;

    cout << "zadejte cele cislo: ";
    do
    {
        is >> x;
        chyba = is.fail();
        if (chyba)
        {
            cout << "spatne zadane cislo, zadejte znova\n";
            is.clear();
        }
    } while (chyba);
}
```

```

        is.ignore(100, "\n");
    }
}
while(chyba);
return(x);
}

```

Soubory

- Konstruktory tříd **ifstream** a **ofstream** s jedním parametrem otevírají soubory jako textové.
- Textové soubory se člení na řádky.
 - Oddělovačem řádků v paměti (v řetězcích) je jediný znak '\n' s kódem 10 (LF)
 - Oddělovačem řádků v souboru je:
 - v OS Windows dvojice znaků CR LF s kódy 13 a 10
 - v OS Unix (Linux a pod.) je jediný znak LF
 - Rozdílné reprezentace oddělovače řádků v souboru řeší knihovny C++ tak, že v OS Windows:
 - Při čtení textového souboru se dvojice bytů s hodnotami 13 a 10 přečte jako jediný znak s kódem 10 (LF).
 - Při zápisu do textového souboru se znak s kódem 10 zapíše jako dvojice bytů s hodnotami 13 a 10.
- Při otevírání binárních souborů, jejichž obsahem nejsou znaky a nečlení se na řádky, je třeba výše uvedenou transformaci potlačit uvedením příznaku **binary**.

```

ofstream out("data.bin", ios::out|ios::binary);

```

Příklad

Zápis a přečtení binárního souboru po blocích.

```

#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    int pole[5] = {1,2,3,4,5};           // vytvorí pole
    ofstream out("pole.bin",ios::out|ios::binary); // otevte soubor
    out.write((const char*)pole,5*sizeof(int)); // zapise pole
    out.close();                         // zavreni souboru

    for (int i=0; i<5; pole[i++]=0);     // vynulovani pole

    ifstream inp("pole.bin",ios::in|ios::binary); // otevreni pro cteni
    inp.read((char*)pole,5*sizeof(int)); // cteni
    inp.close();                         // zavreni

    for (i=0; i<5; cout << pole[i++] << ' ');
    // vypis pole

    return 0;
}

```

Třídy

V jazyku C++ třída slouží především jako popis strukturovaného datového typu a specifikuje, z jakých složek (položek, atributů) se skládají objekty (instances) dané třídy a jaké operace lze s objekty dané třídy provádět.

Příklad

Příkladem je třída čítač, zapouzdřující jednoduché počítadlo.

Citac	název typu
hodn	datové položky
pocHodn	
zvetsit	operace
zmensit	
nastavit	
hodnota	

definice přímo v jednom souboru

```
class Citac
{
    int hodn;
    int pocHodn;
public:
    Citac(int ph) {pocHodn = ph; nastavit();}
    void zvetsit() {hodn++;}
    void zmensit() {hodn--;}
    void nastavit() {hodn = pocHodn;}
    int hodnota() {return hodn;}
};
```

použití hlavičkovéhou souboru

hlavičkový soubor obsahuje pouze hlavičky funkcí a třídní proměnné

```
/* citac.h */
class Citac
{
    int hodn;
    int pocHodn;
public:
    Citac(int ph);
    void zvetsit();
    void zmensit();
    void nastavit();
    int hodnota();
};
```

soubor s implementací obsahuje těla metod

```
/* citac.cpp */
#include "citac.h"
Citac::Citac(int ph) {pocHodn = ph; nastavit();}
void Citac::zvetsit() {hodn++;}
void Citac::zmensit() {hodn--;}
void Citac::nastavit() {hodn = pocHodn;}
int Citac::hodnota() {return hodn;}
```

příklad použití

```
#include "citac.cpp"

int main()
{
```

```

int volba;
cout << "citac\n";
Citact * pcitac = new Citac(0); // vytvorí instanci tridy citac

do
{
    cout << "Hodnota = " << pcitac->hodnota() << endl; // vypis hodnoty citace
    volba = menu(); // nacteni integeru (implementovano jinde...)
    switch(volba)
    {
        case 1: pcitac->zvetsit(); break; // operace s citacem
        case 2: pcitac->zmensit(); break;
        case 3: pcitac->nastav(); break;
    }
}
while(volba > 0);

cout << "Konec\n" << endl;
delete pcitac; // uvolneni pameti
return 0;
}

```

obecná deklarace

```

class T {
    deklarace položek;
    deklarace metod typ jméno( ... );
    deklarace konstruktorů T( ... );
    deklarace destruktoru ~T();
};

```

Přístup k položkám třídy

Řízení přístupu

Přístup k proměnným (datovým položkám), metodám (i konstruktorům) lze omezit pomocí modifikátorů.

- **public** viditelné odevšud
- **private** viditelné pouze v rámci jedné třídy
- **protected** viditelné v dané třídě a všech potomcích z ní vycházejících

Implicitní přístupnost pro třídu je private.

Zápis přístupu

- Přístup ke statickému objektu (struktuře)

```

Class objekt;
int a;
a = objekt.polozka;
a = objekt.getA();

```

- Přístup k dynamicky vytvořenému objektu ($T \rightarrow x()$; je zjednodušený zápis $(*T).x()$;)

```

Class * objekt = new Class();
int a;
a = objekt->polozka;
a = objekt->getA();
a = (*objekt).getA();

```

Konstruktory

Implicitní konstruktor

Používá se všude tam, kde je volán konstruktor a nejsou uvedeny parametry.

```
T a;
T b[10];
T *p = new T;
```

Nemá-li třída žádný konstruktor, je automaticky vygenerován.

Kopírující konstruktor

deklaraci (kde x je typu T)

```
T y = x;
```

interpretuje překladač jako

```
T y(x);
```

ve které se volá tzv. kopírující konstruktor

```
T(const T&)
```

V případě potřeby je vygenerován automaticky a to tak že:

- položky, které jsou objekty (nebo pole objektů) zkopiřuje pomocí kopírujících konstruktorů příslušné třídy
- ostatní položky bitovou kopíí

Konstruktor uživatelské konverze

- Konstruktorem uživatelské konverze typu T1 na T je každý konstruktor třídy T, který lze volat s jediným parametrem typu T1.
- Začíná-li deklarace konstruktoru uživatelské konverze klíčovým slovem **explicit**, je jím umožněna pouze explicitní konverze, bez tohoto slova konstruktor definuje explicitní i implicitní konverzi.

```
class T1 { ... };
class T
{
    ...
    T(int);
    explicit(const char *, int = 0);
    T(T1);
    ...
};

// pouziti
T a = 1      // a = T(1);
T1 c;        // c = T1();
a = c;       // a = T(c);
```

Statické položky

- Lze deklarovat tzv. statické položky, které nejsou součástí objektů, ale jsou součástí třídy (třídní proměnné).
- Deklarace těchto položek začíná klíčovým slovem **static**.

```
class T
{
    int a;           // je součástí každého objektu
    static int pocet; // není součástí žádného objektu
public:
    T(int x) {a = x; pocet++;}
    ~T() {pocet--;}

    static kolik() {return pocet;} // statické metody mohou používat statické proměnné
```

```

};

int T::pocet = 0; // statickým položkám je třeba přidělit paměť deklarací mimo třídu
...
cout << T::kolik(); // ke statickým metodám je možné přistupovat i bez instance

```

Přístupnost statických položek se řídí stejnými pravidly, jako přístupnost ostatních členů třídy.

Staticky vázané metody

- Obvyklé metody nálezející třídě.
- Fyzická adresa metody je známa již při překladu.
- Při přetypování se volá metoda příslušící výslednému typu. (*dále vysvětleno na příkladech*)

Dědění

Deklarace podtřídy

- podtřída má implicitně všechny vlastnosti jako nadřazená třída
- je možné přidat nové položky
- je možné přidat či předefinovat metody
- je možné omezit přístup k metodám nadřazené třídy
 - private změní vše na private
 - public zachová původní přístupnost

```

class T1 : public T
{
    deklarace nových položek
    prototypy nových (předefinovaných) metod
};

```

- Potomek může být přiřazen předkovi.
- Předek nemůže být přiřazen potomkovi.

Reprezentace v paměti

Příklad

```

class Citac
{
    int hodn;
    int pocHodn;
public:
    Citac(int ph);
    void zvetsit();
    void zmensit();
    void nastavit();
    int hodnota();
};

class CitacMod:public Citac
{
    int modul;
public:
    CitacMod(int ph, int mod);
    void zvetsit() { hodn = ++hodn % modul; }
    void zmensit(){ hodn = --hodn % modul; }
};

```

```
Citac x(0);
```

hodn	0
pocHodn	0

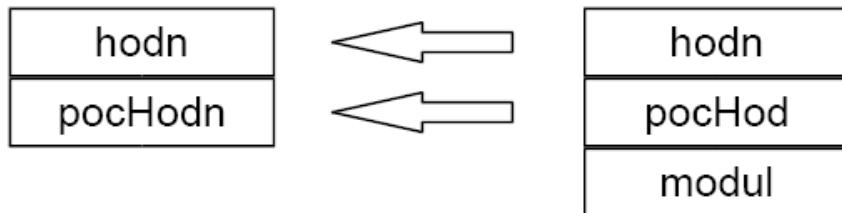
```
CitacMod y(0, 10);
```

hodn	0
pocHod	0
modul	10

Paměťová reprezentace čítače modulo je obohacena jeden int (modulo).

Přetypování

```
Citac c(0);
CitacMod cm(0,5);
...
c = cm;
```



```
c.zvetsit(); // vyvolá se vždy Citac::zvetsit()
cm.zvetsit(); // vyvolá se vždy CitacMod::zvetsit()
```

Při přetypování se použije paměťová reprezentace přetypovaného objektu. V předchozím případě je objekt třídy *CitacMod* přetypován na *Citac*. Příslušný k němu bude probíhat za použití *Citac* metod jakoby se jednalo o obyčejný *Citac*, ačkoliv byl vytvořen konstruktorem *CitacMod*.

Dynamicky vázané metody

- Dynamicky vázané metody začínají klíčovým slovem **virtual**.
- Obvykle jsou vnitřně řešeny pomocí **tabulky virtuálních metod** (*VMT: Virtual Method Table*). Ta pro každou hlavičku virtuální metody vytvoří odkaz na metodu. Příslušná metoda je na místo odkazu vyplňena až za běhu při spuštění konstruktoru příslušného objektu.
- Volání virtuální metody je časově mírně náročnější oproti staticky vázané.

Příklad

```
class T
```

VMT

```

{
    int a,b;
}

public:
    void P();
    virtual void Q();
};

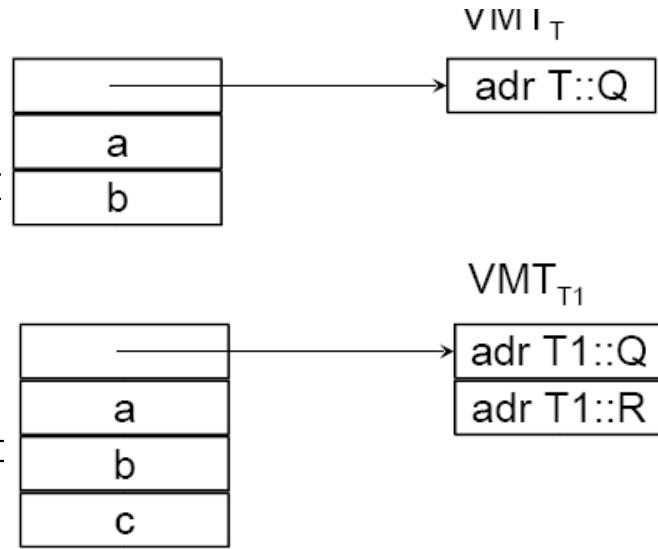
class T1 : public T
{
    int c;

public:
    void P();
    virtual void Q();
    virtual void R();
    void S();
};

T *t = new T();
T *t1 = new T1();

t->P(); // zavola T::P()
t1->P(); // zavola T::P()
t->Q(); // zavola T::Q()
t1->Q(); // zavola T1::Q() (pomoci tabulky virtualnich metod)
t1->R(); // prekladova chyba, T nema metodu R

```



Abstraktní třídy

- jejich základní vlastností je, že **není možné z nich vytvářet instance** (volat konstruktor) a to ani staticky ani dynamicky
- lze ale pracovat s ukazateli a referencemi typu abstraktní třída
- mají alespoň jednu **abstraktní metodu**
 - abstraktní metody mohou být pouze v abstraktních třídách - jakmile třída obsahuje alespoň jednu abstraktní metodu, stává se automaticky abstraktní
 - je dáno **pouze rozhraní metody** (jméno, parametry, ...)
 - **tělo** metody je definováno v **potomcích**
 - metoda však **existuje i v předkovi** (aby se vyhradil prostor v VMT)
 - musí být vždy **virtual** (aby byla při volání byla zavolána správná metoda pomocí Dynamické vazby - viz výše)
 - lze vytvořit pouze abstraktní instanční metodu
 - nelze vytvořit abstraktní konstruktor, destruktor a třídní metodu
 - potomek musí implementovat všechny abstraktní metody, v opačném případě se jedná opět o abstraktní třídu
- může obsahovat implementaci metod, které jsou společné všem potomkům
- **použítí**
 - výchozí bod (jakousi šablonu) pro složitější datové struktury (základ pro dědění)
 - poskytují jednotný pohled na více heterogenních objektů
 - využití rozhraní vyšší úrovně, není třeba rozlišovat detaily implementace v jednotlivých podtřídách - poskytuje rozhraní společné všem potomkům
 - pro implementaci Polymorfních datových struktur - viz dále

Příklad

Geometrické tvary.

```

class Shape
{
public:
    virtual int Area() = 0; // cisate virtualni metoda
};

class Rectangle: Shape

```

```
{
    int a;
public:
    Rectanlge(int size) { a = size; }
    virtual int Area() { return a * a; }
}
```

Zaměstnanci.

```
class CPerson
{
protected:
    string name;
    int born;
public:
    CPerson ( string _name, int _born ) : name (_name), born(_born) { }
    virtual ~CPerson ( void ) { }
    virtual int retired ( int year ) const = 0;
};

class CWoman : public CPerson
{
protected:
    int childs;
public:
    CWoman ( string _name, int _born, int _childs ): CPerson ( _name, _born ), childs(_childs) { }
    virtual int retired ( int year ) const { return year > born + 63 - 2 * childs; }
};

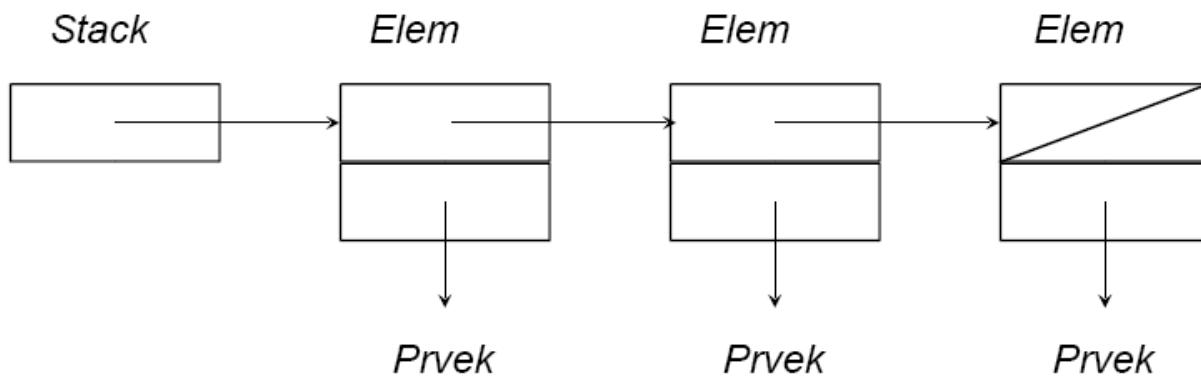
class CMan : public CPerson
{
protected:
    int milSvc;
public:
    CMan ( string _name, int _born, int _milSvc ) : CPerson ( _name, _born ), milSvc(_milSvc) { }
    virtual int retired ( int year ) const { return year > born + 65 - milSvc; }
};
```

Polymorfni datové struktury

- jedná se o datové struktury, které v sobě mohou uchovávat instance odvozené od různých tříd
- typickým příkladem je třeba abstraktní datový typ **zásobník, fronta, tabulka, množina, strom, halda** apod
- datová struktura obsahuje **prvky**, které **podporují určité operace**, jinak řečeno implementují definované rozhraní
- polymorfni datové struktury jsou z pravidla **implementovány pomocí abstraktních tříd** (viz výše), které právě definují příslušné rozhraní. **Typy prvků**, které je taková struktura schopna ukládat jsou **podtřídami** (ve smyslu dědičnosti) dané abstraktní třídy
- **dělí se na**
 - **homogenní** - obsahují **prvky stejného typu** ⇒ operace se provádí pro všechny prvky stejně
 - **heterogenní** - obsahuje **prvky různých typů** ⇒ operace se mohou provádět různě v závislosti na typu praktu

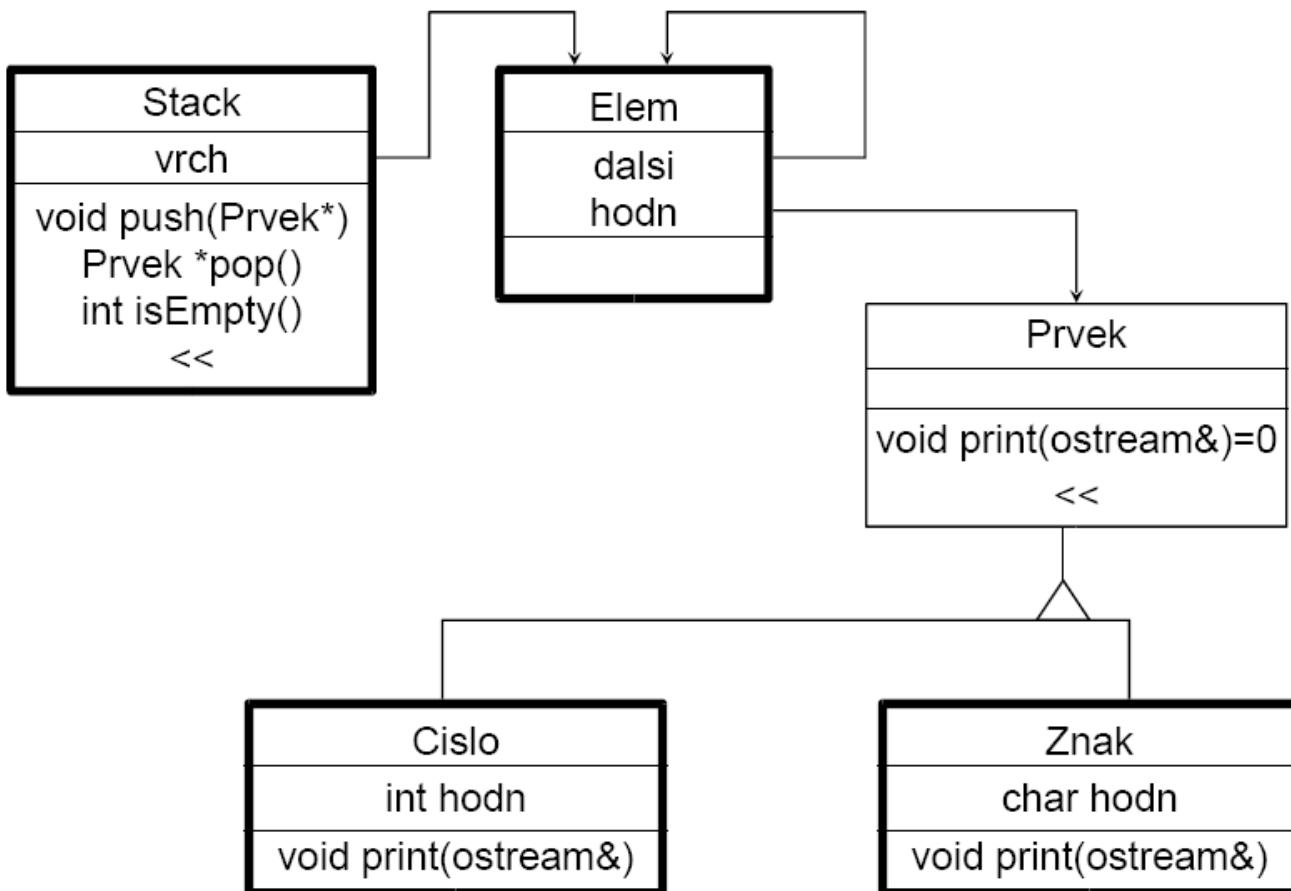
Příklad - polymorfni zásobník

- implementován spojových seznamem dynamických proměnných
- tyto proměnné obsahují ukazatel na prvek zásobníku, situace je znázorněna na následujícím obrázku



- typy prvků, které bude zásobník schopen uchovávat budou podtřídami abstraktní třídy CPrvek - v této třídě bude pro všechny prvky přetížen operátor výstupu «, který bude volat abstraktní metodu Print.
- vše výše popsané zachycuje následující obrázek

Polymorfní zásobník (pokračování)



Zdrojový kód

```
class Stack
{
    struct Elek
    {
        Prvek * hodn;
```

```

        Elek * dalsi;
        Elek (Prvek *h, Elek *d){hodn = h; dalsi = d;
    };
    Elek * vrch;
public:
    Stack ()
    {
        vrch = 0;
    }
    void push ( Prvek * p)
    {
        vrch = new Elek (p, vrch);
    }
    Prvek * pop ()
    {
        Elek *pom = vrch;
        Prvek *prvek = vrch->hodn;
        vrch = vrch->dalsi;
        delete pom;
        return prvek;
    }
    int isEmpty()
    {
        return vrch == 0;
    }
    friend ostream& operator<<(ostream&, Stack&);
};

/* stack.cpp */
#include "stack.h"
ostream& operator<<(ostream& s, Stack& stc)
{
    Stack::Elek *pom = stc.vrch;
    while (pom)
    {
        s << *pom->hodn << endl;
        pom = pom->dalsi;
    }
    return s;
}

/* main.cpp */
/* Stack s astraktním prvkyem */
#include "stack.h"
#include <iostream>
using namespace std;
class Cislo : public Prvek
{
    int hodn;
public:
    Cislo(int h) {hodn = h;}
    virtual void print(ostream& s) const {s << hodn;}
};
class Znak : public Prvek
{
    char hodn;
public:
    Znak(char h) {hodn = h;}
    virtual void print(ostream& s) const {s << hodn;}
};
void main()
{
    Stack s;
    Znak z('b');
    Cislo c(222);
    cout << "\nPolymorfni zasobnik\n";
    s.push(new Cislo(10));
    s.push(new Znak('a'));
    s.push(&z);
    s.push(&c);
    cout << "Vypis zasobniku\n" << s;
    cout << "Odebrani ze zasobniku\n";
    while (!s.isEmpty())
    {
        cout << *s.pop() << endl;
    }
}

```

Poznámky k parametrům

- Je-li parametr funkce (metody, konstruktoru) typu reference na třídu T, přípustným skutečným parametrem je objekt typu T', kde T' je T nebo podtřídou T. Ve druhém případě se předá odkaz na objekt, aniž by se prováděla konverze na T.
- Funkce operator« definovaná pro třídu Prvek se vyvolá i pro objekt typu Cislo nebo Znak; vlastní výpis provede metoda print, která je definovaná pro každou podtřídu jinak.
- Parametr typu reference na nadtíedu může být též výstupním parametrem

Výběr ukazatele z polymorfní datové struktury

- Při výběru ukazatele na prvek polymorfní datové struktury je někdy třeba jej uložit do proměnné typu ukazatel
- Obsahuje-li datová struktura ukazatele na instance podtříd abstraktní třídy T, lze vybraný ukazatel uložit přímo (bez přetypování) pouze do proměnné typu T*

Příklad 1:

```
Stack s;
s.push(new Cislo(10));
s.push(new Znak('a'));
Znak *pc = s.pop(); // chyba při překladu
Cislo *pc = s.pop(); // chyba při překladu
```

Příklad 2:

```
Stack s;
s.push(new Cislo(10));
s.push(new Znak('a'));
Prvek *p1 = s.pop(); // O.K.
Prvek *p2 = s.pop(); // O.K.
```

Dynamické přetypování na ukazatel na podtřídu

- Ukazatel na nadtíedu lze dynamicky přetypovat na ukazatel na podtřídu

Příklad: Součet čísel v polymorfním zásobníku

```
int main()
{
    Stack s;
    s.push(new Cislo(10));
    s.push(new Cislo(29));
    s.push(new Znak('a'));
    cout << s << endl;
    Cislo *pc;
    int součet = 0;
    while (!s.isEmpty())
    {
        pc = dynamic_cast<Cislo*>(s.pop());
        if (pc!=NULL)
        {
            součet += pc->hodn;
        }
    }
    cout << "součet cisel v zásobníku = " << součet << endl;
    return 0;
}
```

Dynamický test typu objektu

- Někdy stačí pouze zjistit typ objektu, na který ukazuje proměnná typu nadtíeda objektu

Příklad: počet čísel a znaků v polymorfním zásobníku

```
#include <typeinfo>
int main ()
{
```

```
Stack s;
s.push(new Cislo(10));
s.push(new Cislo(29));
s.push(new Znak('a'));
cout << s << endl;
Prvek *p;
int pocetCisel = 0, pocetZnaku = 0;
while (!s.isEmpty())
{
    p = s.pop();
    if (typeid(*p) == typeid(Cislo))
    {
        pocetCisel++;
    }
    else if (typeid(*p) == typeid(Znak))
    {
        pocetZnaku++;
    }
}
cout << "pocet cisel = " << pocetCisel << endl;
cout << "pocet znaku = " << pocetZnaku << endl;
return 0;
}
```

Zdroje

- Wikipedia [<http://wikipedia.org>]
- přednáškové materiály doc. Mullera [http://www.stm-wiki.cz/images/7/7c/X36PJC_slajdy.pdf]
- http://www.builder.cz/art/cpp/cpp_oop.html [http://www.builder.cz/art/cpp/cpp_oop.html]
- http://www.infocom.cqu.edu.au/Staff/Mike_Turnbull/Home_Page/LectureTts/Toc.htm [http://www.infocom.cqu.edu.au/Staff/Mike_Turnbull/Home_Page/LectureTts/Toc.htm]
- <http://www.cplusplus.com/> [<http://www.cplusplus.com/>]

Otázka 11 - Y36PJC

Zadání: Objektově orientované programování v C++, přetěžování operátorů, generické funkce a třídy, výjimky, knihovny.

Objektově orientované programování

Nejdříve trochu k OOP a jeho základním rysům, které později probereme v pohledu C++. Základní pojmy z OOP jsou:

- **Třída** - definuje předmět, který modelujeme - jeho vlastnosti (atributy) a schopnosti (metody). Dohromady se atributy a metody třídy nazývají členové třídy
- **Objekt** - je konkrétní exemplář určité třídy. Již existuje v paměti, program může volat jeho metody atd.
- **Instance třídy X** - je označení určitého objektu, který je třídy X
- **Metoda** - funkce určité třídy. Funkce se v programu vyvolává buď pro třídu (statická metoda), nebo pro instanci třídy

Základní ideje OOP:

- **Dědičnost** - Třída může být „poděděná“ od jiné třídy, což znamená, že má všechny členy (atributy, metody...) tříd, ze kterých dědí (předků), a může přidat nové, případně upravit existující (z této možnosti vychází koncept polymorfismu). V programovacích jazycích může být podporována i několikanásobná dědičnost, umožňující třídě být potomkem několika tříd (které mezi sebou nemají žádný vztah) najednou.
- **Abstrakce** - Díky dědičnosti je možné pracovat s třídami na libovolné vhodné úrovni ve stromu jejich předu (např. máme pole zvířat *Zvire[]* - nezájímá mě jakých - ale každý prvek pole může být nějaký (jiný) potomek třídy *Zvire*)
- **Zapouzdření** - Třídy nám umožňují nevidět konkrétní detaily fungování třídy (metoda *Stekni()* může využívat jiné metody atd., ale to nás nemusí zajímat)
- **Polymorfismus** - OOP nám umožňuje pracovat s potomkem třídy X jako se třídou X samotnou - ale potomek může provádět při práci s jeho členy akce, které mu odpovídají (např. třída *Zvire* má metodu *DelejZvuk()*, kterou má *Slon* implementovanou jako troubení a *Vlk* jako vytí. A když voláme u nějakého objektu, se kterým pracujeme jako s třídou *Zvire*, metodu *DelejZvuk()*, díky polymorfismu se ozve správný zvuk - troubení, vytí atd..)

OOP v C++

C++ zapojuje všechny výše zmíněné pojmy a koncepty. Jejich konkrétní podobu popisuje otázka 10.

Přetěžování operátorů

Standardní operátory jsou přetížené: např: $10 + 1$, $1.5 + 2$, etc...

V C++ lze na rozdíl od většiny ostatních jazyků operátory dále přetěžovat pro jiné typy parametrů:

- nelze zavést nové operátory,
- nelze předefinovat stávající operátory,
- nelze měnit aritu operátorů,
- nelze měnit prioritu operátorů

Přetížení operátorů:

- klíčové slovo **operator**,
- přetížení pomocí funkce,
- přetížení pomocí metody.
- Metodou lze přetížit všechny operátory kromě `::...* ?`:
- Funkcí dále nelze přetížit operátory → `→* = () []`
- Přetížit funkcí či třídní metodou lze i operátory: `new new[] delete delete[]`
- Metodou lze přetížit i operátory přetypování

Přetěžování operátorů funkcí

Příklad přetěžování operátoru +

```
struct TCplx
{
    double re, im;
};

TCplx operator + ( TCplx a, TCplx b )
{
    TCplx c;
    c . re = a . re + b . re;
    c . im = a . im + b . im;
    return c;
}
```

Problém: Tento přetížený operátor nebude použit u `TCplx a, b; a = b + 3; a = 4 + b;` Abychom mohli použít přetížený operátor zde, musíme ho přetížit i takto:

```
TCplx operator+ ( TCplx a, double b ) { ... }
TCplx operator+ ( double a, TCplx b ) { ... }
```

Druhou možností jak řešit tento problém je ponechat pouze jeden přetížený operátor + a využít konstruktor uživatelské konverze z typu double na typ TCplx

```
TCplx::TCplx ( double a )
{
    re = a;
    im = 0;
}

TCplx a, b;
a = 4 + b; // a = TCplx ( 4 ) + b;
```

Přetěžování operátorů metodou

- Přetížení operátoru funkcí:
 - nelze pro všechny operátory,
 - problémy s přístupem ke členským proměnným objektů.
- Řešení – přetížení metodou:
 - přístup ke členským proměnným,
 - použitelné pro všechny operátory, které lze přetížit.
- Realizace:
 - jméno metody – operator ...,
 - parametry – o jeden méně, než je arita operátoru,
 - levý operand – instance nad kterou je metoda spuštěna.

Příklad přetěžování operátorů metodou

```
class CCplx
{
    double re, im;
public:
    CCplx ( double r, double i=0 ) : re(r), im(i) {}
    CCplx operator - ( void ) const;
    CCplx operator + ( const CCplx & x ) const;
};

CCplx CCplx::operator - ( void ) const
{ // unarni minus – 0 parametr
    return ( CCplx ( -re, -im ) );
}

CCplx CCplx::operator + ( const CCplx & x ) const
{ // binarni plus – 1 parametr
    return ( CCplx ( re + x. re, im + x . im ) );
}
```

Generické funkce a třídy (Templates)

Pod generickými funkcemi a třídami si lze představit způsob, jak vytvořit šablonu (funkce nebo třídy), která pracuje s určitým typem (resp. hodnotou), aniž je konkrétně specifikováno, o jaký typ (resp. hodnotu) konkrétně jde. Následně můžeme tuto šablonu vzít a deklarovat, že pracuje např. nad našim typem Zvíře (nebo pro hodnotu 10)

Výhody:

- Vyhne se vytváření funkčně identického kódu pro různé typy.
- Zvyšujeme typovou bezpečnost (v některých případech bychom šablony nahrazovali různými operacemi s ukazateli, které by nebyly ověřitelné kompilatorem).

Jak to funguje

Před deklaraci metody nebo funkce vložíme klíčové slovo template s parametry šablony. Parametry šablony mohou být dvou typů:

- Typové parametry - (uvorené kl. slovem **class** nebo **typename**)
- Hodnotové parametry - (uvorené typem parametru - chceme specifikovat číslo, použijeme int)

Parametry šablony následně použijeme v třídě/funkci na místě, kde bychom použili konkrétní typ (resp. hodnotu příslušného typu)

Vytvoření naší šablony pro konkrétní parametry je provedeno automatický komplátorem při prvním použití generického bloku s daným parametrem (resp. jejich kombinací). Kompilátor se pokusí zjistit parametry z použití, nebo musí být specifikovány explicitně pomocí < a > (viz ukázka šablony metody). S konkrétními parametry pak vytvoří samostatnou implementaci funkce, resp. třídy. Pro třídy to tedy znamená, že stejná generická třída pro různé typy nesdílí statické proměnné.

Šablona může parametrů mít i více (oddělených čárkou)

To vše se provádí při komplaci a je ověřována typová bezpečnost.

Šablony metod

Slouží pro specifikování šablony jedné metody.

Ukázka:

```
template <class T> // typový parametr šablony T
T max(T a, T b)
{
    return a > b ? a : b ;
}
void main()
{
    //Funkce je vytvořena pro int - tedy je to funkce int max(int a, int b)
    cout << "max(10, 15) = " << max(10, 15) << endl ;

    // Funkce je vytvořena pro char
    cout << "max('k', 's') = " << max('k', 's') << endl ;

    //Kompilátor by nebyl schopen určit typ - specifikován explicitně a fce vytvořena pro double
    cout << "max<double>(10, 15.2) = " << max<double>(10, 15.2) << endl ;
}
```

V příkladu jsme deklarovali funkci, která pro obecný typ T vrátí větší z dvou parametrů. Následně se v main funkci použila pro parametry typu int, char a double. Při překladu kompilátor vytvořil tedy tři funkce (ve zkompilovaném souboru bude funkce max 3x, ale vždy pracovat s jinými typy). Pro poslední použití bylo nutné explicitně specifikovat typový parametr šablony, jelikož kompilátor by nebyl schopen z parametrů funkce určit, pro jaký typ šablony vytvořit - 1. parametr je typu int a 2. typu double (pokud ale vytvoříme max, která operuje s double, nevadí to - díky implicitním konverzím jazyka).

Šablony tříd

Slouží pro specifikování šablony jedné třídy. Template hlavičku definující parametry šablony je nutné dávat jak před deklarací třídy, tak i před implementace jednotlivých metod třídy. Navíc musí být generické třídy deklarovány a implementovány v jednom souboru - nemohou být rozděleny do .h a .cpp souborů, jako je u tříd zvykem.

Při použití šablony třídy uvádíme parametry explicitně vždy, protože kompilátor nemá z čeho je poznat.

Ukázka:

```
template <class T, int jeOtevrena> //Pro ilustraci je zde druhý parametr. 1. je typový, 2. hodnotový
class Garaz
```

```

{
public:
    void zaparkuj(T* auto);
    T* vyparkuj();
private:
    T* parkovaciMisto;
} ;

template <class T>
T* Garaz<T>::vyparkuj()
{
    T* temp = parkovaciMisto;
    parkovaciMisto = NULL;
    return temp;
}

template <class T>
void Garaz<T>::zaparkuj(T* auto)
{
    if(jeOtevrena == 0) return;
    parkovaciMisto = auto;
}

void main()
{
    Garaz<Porsche, 1> garaz; //Garáž pro objekty typu Porsche s jeOtevreno = 1
    Porsche *auto = new Porsche();
    garaz.zaparkuj(auto);
    Porsche *vyparkovanyAuto = garaz.vyparkuj();
    delete auto;
}

```

Specializace šablon

V některých je třeba pro konkrétní hodnoty parametrů šablony upravit - změnit jak se bude při využití s daným parametrem šablony chovat. Ukazka, která upravuje funkci max z prvního příkladu:

Mám-li například funkci *max* z příkladu šablon metod:

```

template <class T> // typový parametr šablony T
T max(T a, T b)
{
    return a > b ? a : b ;
}
void main()
{
    //Vytvoří se funkce char* max(char* a, char* b) a bude porovnávat adresy obou stringů!
    // (ale my chceme porovnat řetězce na těch adresách)
    cout << "max(\"Aladdin\", \"Jasmine\") = " << max("Aladdin", "Jasmine") << endl ;
}

```

Mohu provést specializaci šablony pro typ *char** a dělat žádoucí porovnání.

```

template <>
char* max(char* a, char* b)
{
    return strcmp(a, b) > 0 ? a : b ;
}

```

Výchozí parametry

Tak jako pro parametry funkcí, je i pro parametry šablon možné zadat výchozí hodnoty parametru. Je tedy

možné pak třídě předat méně parametrů, nebo i žádný.

Hlavíčka šablony třídy *Garage* by mohla vypadat např. takto:

```
template <class T = Porsche, int jeOtevrena = 1>
```

A její použití pak :

```
Garage garaz; //Otevřená Porsche garáž
```

Výjimky

- Výjimka je programová entita označující chybu při výpočtu. Různé chyby se označují různými výjimkami.
- Výskyt chyby se signalizuje vyvoláním výjimky. Může jí vyvolat jak interpretace programu tak program pomocí speciálního příkazu.
- Jsou zavedeny strukturované příkazy, které předepisují provední dílčích příkazů a dále obsluhu výjimek, které mohou být vyvolány při provádění dílčích příkazů. Každá obsluha je označena identifikací výjimky a tvoří ji příkazy, které se mají provést jako reakce na danou chybu.
- Obsluhy výjimek tvoří dynamický řetězec. Při vstupu do příkazu s obsluhou výjimek se tento řetězec rozšíří o nový záznam obsluhy, po řádném ukončení se záznam zruší.
- Při vyvolání výjimky se v řetězci obsluh výjimek hledá nejbližší záznam obsahující obsluhu dané výjimky. Přitom se zruší aktivační záznamy podprogramů, které obsluhu výjimky neobsahují.

V C++ se výjimky rozlišují pomocí datových typů. Vyjímkou tedy může reprezentovat jak číslo, znak nebo řetězec, tak instance tříd (objekty). Pro vyvolání vyjímkky slouží příkaz **throw**, jehož argumentem je výraz určující typ a hodnotu dané výjimky. Řídící strukturou je příkaz **try**. Vyjímkou se zachytává v části **catch(...){...}**.

```
void* AlokujPamet(int size){
    void* ptr;
    ptr = malloc((size_t) size;
    if (ptr==0) throw("nepodařilo se alokovat pamet");
    return ptr;
}

int main(){
    try{
        AlokujPamet(100);
    }
    catch(char* v){
        printf("%s\n", v);
    }
}
```

Knihovny

Programovací jazyk C sám o sobě obsahuje minimum funkcí - většina funkcí se nachází v knihovnách na které se musíme na začátku programového kódu odvolat. Zde si připomeneme základní knihovny jazyka C.

- **string.h** - obsahuje funkce pro práci s řetězci.
 - **char *strcat(char *dest, const char *src)** - Funkce připojí řetězec src k řetězci dest. Funkce vrací ukazatel na řetězec dest.
 - **int strcmp(const char *s1, const char *s2)** - Porovnává řetězce s1 a s2. Pokud je s1 < s2, vrací hodnotu menší než 0, pokud jsou si rovny, vrací 0, pokud je s1 > s2 vrací hodnotu větší jak 0.
 - **char *strcpy(char *dest, const char *src)** - Zkopíruje řetězec src do řetězce dest. Vrací ukazatel na dest.
 - **size_t strlen(const char *s)** - Vrací délku řetězce s.
- **stdio.h** - knihovna pro standardní vstup a výstup. Slouží především pro práci s příkazovou řádkou a soubory.
 - **int scanf(const char *format, ...)** - čte data ze standardního vstupu.
 - **int printf(const char *format, ...)** - slouží k formátovanému výstupu do stdout.
 - **FILE *fopen(const char *path, const char *mode)** - otevře soubor aby s ním bylo možné pracovat jako s datovým proudem
 - **int fclose(FILE *stream)** - uzavře (a vyprázdní) datový proud stream.
 - **size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)** - načte nmemb dat o velikosti size ze souboru stream do paměti kam ukazuje ptr. Návratovou hodnotou je počet správně načtených položek.
 - **size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)** - zapíše nmemb dat o velikosti size do souboru stream z paměti kam ukazuje ptr. Návratovou hodnotou je počet správně zapsaných položek.
- **time.h** - funkce pro práci s časem.
 - **char *asctime(const struct tm *timeptr)** - Převede strukturu tm na řetězec („Wed Jun 30 21:49:08 1993\n“). Návratovou hodnotou je ukazatel na tento řetězec.
 - **double difftime(time_t time1, time_t time0)** - Vrací rozdíl mezi časem time0 a time1 ve vteřinách.
 - **time_t time(time_t *t)** - Vrací počet sekund reprezentující čas (v Unixech počet sekund od 1.1.1970). Tuto hodnotu také uloží tam, kam ukazuje ukazatel t, pokud nemá hodnotu NULL). Je to jediná funkce, která nám vrací aktuální čas.
- **stdlib.h** - převody řetězců a čísel, ukončování programu, třídění pomocí qsort(), funkce abs(), div(), systémová volání atp.
- **ctype.h** - funkce pro testování znaků. isalnum(x), isalpha(x), iscntrl(x), isdigit(x), tolower(x), toupper(x)
- **math.h** - matematické operace. sin(x), cos(x), ..., exp(x), log(x), sqrt(x),...
- **errno.h** - knihovna <errno.h> definuje jedinou proměnnou, a to extern int errno - Tuto proměnnou můžeme použít k nalezení chyb.

Otázka 12 - Y36SI2

Zadání: Jak zajistit kvalitu SW a jak ji ověřit. Potřeba kultivovat kvalitu. Vyhýbání se chybám a ostatním problémům s kvalitou. Inspekce a revize, efektivní vedení efektivních inspekcí.

Slovníček pojmu

jakost (= kvalita)

- podle normy ISO/DIS 9000:2000 - schopnost souboru inherentních znaků výrobku, systému nebo procesu plnit požadavky zákazníků a jiných zainteresovaných stran
 - **inherentní znak** - je nějaká rozlišující vlastnost, která je pevně svázána s výrobkem (např: pevnost, funkčnost, bezporuchovost). Cena není inherentní znak, jelikož se může měnit a není pevně svázána s výrobkem.
 - **Výrobek (produkt)** - hardware, software, služby a zpracované materiály
 - **Požadavky** - jsou potřeby nebo očekávání, které se předpokládají nebo jsou závazné
 - závazné požadavky na software jsou dány jeho specifikací
 - předpokládané požadavky nejsou v žádném závazném dokumentu, ale bez nich by výsledný produkt nebyl kvalitní (software je stabilní, nekoliduje s ostatními programy etc.)
- jiná definice - jakost softwaru je shoda s explicitně stanovenými požadavky na funkci a chování, explicitně dokumentovanými vývojovými standardy a implicitními charakteristikami, které se očekávají od každého profesionálně vyvinutého softwaru
 - **Vývojový standard** - kvalitní produkt musí být vytvořen kvalitním procesem
 - kvalitu nemůžeme kontrolovat až potom, co byl vytvořen programový kód, ale během celého vývoje softwaru

Jak zajistit kvalitu SW

Management jakosti (podle ISO/DIS 9000:2000)

- jsou koordinované činnosti pro nasměrování a řízení organizace s ohledem na jakost
- zahrnuje (z hlediska řízení projektu jsou důležité body 2-4):
 1. stanovení politiky jakosti a cílů jakosti
 2. **plánování jakosti**
 - spočívá ve stanovení cílů jakosti
 - specifikování procesů pro splnění těchto cílů
 - výsledkem je plán řízení jakosti
 3. **řízení jakosti**
 - splnování požadavků na jakost
 - sledování konkrétních výsledků projektu a posuzování, zda odpovídají standardům a stanoveným požadavkům
 - pokud projekt neodpovídá požadavkům, navrhují se způsoby odstraňování příčin nevyhovujícího plnění

4. zabezpečování jakosti

- poskytování důvěry, že jsou splněny požadavky na jakost
- ujištění zákazníka nebo třetí strany o kvalitě

5. zlepšování jakosti

Ověření jakosti softwaru

Přezkoumání, revize (review)

- činnost prováděná k zajištění vhodnosti, přiměřenosti, efektivnosti a účinnosti předmětu s cílem dosáhnout stanovených cílů
- tři základní metody přezkoumání:
 - inspekce
 - procházení (walk-throughs)
 - osobní přezkoumání

Inspekce

- formální metoda týmového přezkoumání podle přísných pravidel
- má 3 části
 - 1. **Příprava**
 - nazačátku je krátká schůzka, seznámení účastníků inspekce s problematikou
 - individuální studium podkladů a odkrývání problémů a otázek
 - 2. **Vlastní inspekce**
 - je setkání inspekčního týmu s realizátory
 - probírají se nalezené problémy a vyjasňují případné otázky
 - 3. **zpráva**
 - specifikují se nalezené defekty

Formální technická revize (FTR – Formal Technical review)

- inspekci se také někdy nazává formální technická revize
- **cíle** formální technické revize
 - odkrýt chyby ve funkci, v logice, v implementaci
 - ověřit, že software vyhovuje požadavkům
 - ujistit se, že software je reprezentován podle definovaných standardů
 - zajistit, aby software byl vyvinut jednotným způsobem
 - docílit, aby projekt byl lépe ředitelný
- v týmu o třech až pěti pracovnících
- rozsáhlý produkt je třeba rozdělit na části a ty zkoumat samostatně
- každou FTR musí někdo vést a každé jednání musí někdo zapisovat
- **závěr** FTR stanoví zda
 - akceptovat produkt bez modifikací
 - odmítnout produkt kvůli závažným chybám (budou-li opraveny má se uskutečnit nová revize)

- předběžně akceptovat s tím, že menší chyby budou opraveny (nová revize nebude prováděna)

Procházení (walk-through)

- méně formální než Inspekcce
- Realizátor prochází programem a vysvětluje, jak program řeší jednotlivé situace
- Cílem je objevit případná nedorozumění a nedostatky

Osobní přezkoumání

- provádí tvůrce programu
- autor prochází jednotlivé modely (analýzu, návrh, implementaci) a snaží se zjistit chyby
- je dobré mít i zde svůj plán, jak zkoumáme - třeba kontrolní seznam

Testování

- prověrování funkčnosti produktu (programu je to spuštění za účelem nalezení chyb)
- pokud testování chyby neobjeví, znamená to, že jsme neprovedli správný test 😊

Simulace

- ruční nebo poloautomatické procházení částí programu a kontrolování správné funkčnosti

Formální důkaz správnosti

- metoda, která má formou matematického důkazu ověřit, že je program správný

Evaluace

- celkové zhodnocení rozsáhlých materiálů
- provádí se přezkoumáním nebo inspekcí.
- může jít např. o studie proveditelnosti (feasibility study) - zjištění, zda jsou požadavky úplné, bezesporné, ve shodě s cíly projektu

Verifikace

- ověření, zda výstupy etapy splňují požadavky vstupních dokumentů
- provádí se přezkoumáním, inspekcí, procházením nebo čtením kódu
- dále se sleduje dodržování termínů, rozpočtu, norem a standardů

Validace

- praktické ověření správné činnosti testováním

Audit

- prověření dodržování a stavu plnění úkolů nezávislou skupinou
- některou specifickou oblast (např. účetní audit) může prověřovat jen akreditovaná organizace

Jakost versus cena

- každý další test může odkrýt další chyby, ale také něco stojí
- je vždy výhodnější vynaložit náklady na kvalitu na začátku procesu než na jeho konci
- náklady na kvalitu dělíme na
 - **náklady na prevenci** - plánování jakosti, školení týmu, pořízení nástrojů na testování
 - **náklady na provádění** - cenu za přezkoumání, inspekce a testování
 - **náklady na odstranění chyb** - se liší, je-li chyba objevena před dodáním zákazníkovi (vnitřní chyba) nebo po dodání (vnější chyba)
- náklady rostou od prevence k detekci chyb a od odstraňování vnitřních k odstraňování vnějších chyb
- existuje několik srovnání, jaká je relativní doba identifikace chyby v různých fázích vývoje softwaru
 - jedna z nich říká:

Fáze vývoje	relativní doba identifikace chyby
během stanovení požadavků	1
během návrhu	3-6
během kódování	10
při vývojovém testu	15-40
při akceptačním testu	30-70
během provozu	40-1000

- cena za opravu chyby u zákazníka je tím větší, kolik zákazníků produkt používá
 - cena zahrnuje vyřízení reklamace, vrácení produktu, jeho opravy a poskytnutí náhrady

Model zesilování defektů

- model, říká, že v každém kroku vývoje softwaru můžeme a obvykle uděláme nějaké chyby
- jestliže je neobjevíme, přenáší se do dalšího kroku
 - v dalším kroku buďto přetrávají bez změny
 - nebo v častějším případě jsou zdrojem následných chyb – zesilují se

Statistiky jakosti softwaru

- pro zlepšení je potřeba dělat si statistiky o defektech, vyhodnocovat výsledky a poučit se z minulých chyb
- **statistický přístup k jakosti softwaru**
 1. sběr a kategorizace informací o softwarových defektech
 2. nalezení příčiny pro každý defekt (např. chyba návrhu, špatná komunikace se zákazníkem)

3. identifikace závažných příčin

■ **Pareto princip**

- velkou část následků způsobuje jen malé procento příčin
- typický je poměr 80:20 (zdrojem 80% všech chyb je jen 20% možných říčin)

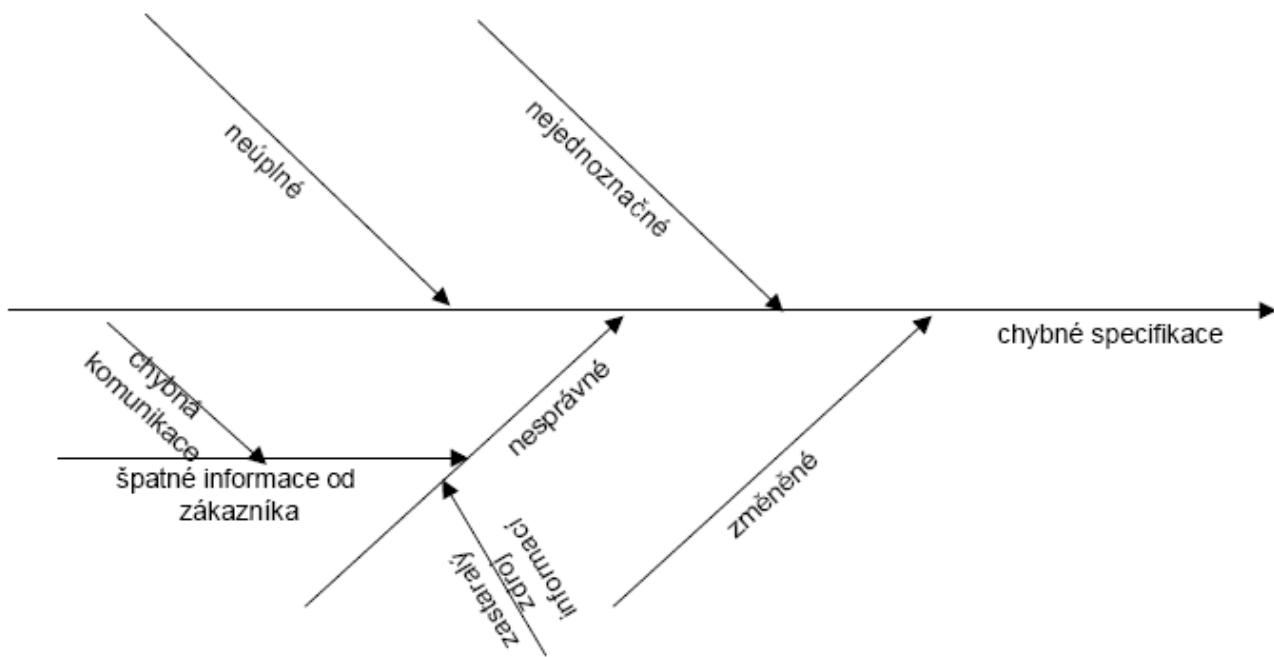
4. ošetření problémů, které způsobují závažné příčiny

■ Příklad:

- Po určitý časový úsek (např. rok) jsme ve firmě zaznamenávali informace o chybách (bod 1)
- Ke každé nalezené chybě jsme našli její příčinu (bod 2)
- Stanovili jsme závažnost každé chyby (bod 3)
- zjištěné hodnoty jsou:

	Příčina	Vážné	Střední	Malé	Celkem
1	chybná specifikace	6	12	20	38
2	nedorozumění v komunikaci se zákazníkem	1	12	12	25
3	chyba návrhu	30	32	70	132
4	implementační chyba	56	77	105	238
5	neúplné testování	5	20	19	44
6	nejednoznačné nebo nekonsistentní uživatelské rozhraní	1	15	5	21
7	ostatní	3	5	30	38
	součet	102	173	261	536

- z tabulky zjistíme, že 69% všech chyb a 84% závažných chyb nastalo v návrhu a implementaci. Řešením by mohlo být nákup CASE nástroje pro zlepšení procesu návrhu. Pro zkvalitnění implementace můžeme zařídit školení programátorů nebo zaměstnat zkušenější (bod 4).
- při hledání příčin problémů nám může pomoci **fishbone diagram** (**Diagram příčin a následků, Ishikawův diagram**)
 - na obrázku je příklad rozkladu příčin, které vedou k chybě specifikací (skripta str. 75)



Chybový index (El error index)

- je softwarová metrika sloužící k indikaci zlepšování jakosti softwaru
 - v každém kroku vývoje spočítáme fázový index

$$PI_i = w_v \left(\frac{V_i}{E_i} \right) + w_s \left(\frac{S_i}{E_i} \right) + w_m \left(\frac{M_i}{E_i} \right)$$

E_i - celkový počet chyb objevených v i-tém kroku procesu, z toho

V_i - počet vážných chyb

S - počet středních chyb

M_i - počet malých chyb

PS - velikost produktu (v počtu LOC, řádek návrhu, stran dokumentace) v i-tém kroku

w_1, w_2, w_m - váhový faktor (doporučené hodnoty jsou 10, 3, 1)

- chybový index je pak:

$$EI = \text{suma} \frac{(i PI_i)}{PS}$$

- každý krok má větší váhu (chyby nalezené později mají závažnější důsledky, než chyby objevené v dřívějších krocích)

Spolehlivost softwaru

- je důležitým znakem kvality softwaru
- je pravděpodobnost bezporuchového fungování programu v daném prostředí a daném čase
 - Poruchou (failure) programu se rozumí neshoda s jeho požadovanými specifikacemi
- u softwaru nevznikají poruchy v průběhu času (např. opotřebení), ale budoucí poruchy jsou zapříčiněny chybami při jeho vývoji
 - to vyžaduje specifický přístup k pojmu spolehlivost a také tomu přizpůsobené metody měření

- **střední doba mezi poruchami**
 - metrika pro měření spolehlivosti softwaru

$$MTBF = MTTF + MTTR$$

MTBT - střední doba mezi poruchami (mean time between failure)

MTTF - střední doby do poruchy (mean time to failure)

MTTR - střední doba opravy (mean time to repair)

- tato metrika je z pohledu uživatele vhodnější, než například součet všech chyb na jednotku programu (počet defektů/KLOC)
 - některé chyby se totiž projeví krátce po nasazení, ale jiné mohou přetrvávat roky. Odstranění chyb v častěji používaných částech programu má větší dopad na spolehlivost.

- **použitelnost software (availability)**
 - jiná metrika pro měření spolehlivosti softwaru
 - pravděpodobnost, že program bude pracovat podle požadovaných specifikací v daném čase

$$A = \frac{MTTF}{(MTTF + MTTR)} \cdot 100\%$$

Benchmarking

- je technika pro sledování procesu a porovnávání jej s podobnými procesy, prováděnými jinými jedinci, skupinami nebo organizacemi
- podmínkou je, aby procesy byly měřitelné metrikami nezávislými na procesu ale odrážejícími jeho schopnosti a robustnost
- Postup benchmarkingu
 - Měřit schopnost procesu produkovat vysokou kvalitu produktu
 - poskytovat jasné uspořádání vlastností procesu od nejlepšího k nejhoršímu
 - indikovat schopnost procesu odolat krizovým situacím
 - poskytovat požadovaná data objektivně měřitelná
 - poskytovat data včas

Nejdůležitější body

- kvalitu produktu nemůžeme pouze kontrolovat na konci výrobního procesu, musíme ji řídit po celou dobu procesu
- řízení kvality softwaru je levnější a účinnější, čím dříve v procesu vývoje začneme provádět účinné kontroly a přezkoumání
- kvalitní produkt vytvoříme jenom kvalitním procesem
- řídit kvalitu procesu můžeme jen tehdy, pokud jej budeme měřit
- máme-li zlepšovat proces, musíme hledat příčiny minulých chyb a poučit se z nich
- normy nám pomáhají využívat zkušeností a vyzkoušených praktik úspěšných organizací.

Zdroje

- Slidy z přednášky [<http://service.felk.cvut.cz/courses/Y36SI2/Prednasky/SI2%20prednasky%20pdf/zajisteni%20kvality.pdf>]
- Příklad z přednášky [<http://service.felk.cvut.cz/courses/Y36SI2/Prednasky/SI2%20prednasky%20pdf/rizeni%20kvality.pdf>]

- Formal Technical Review Methods [<http://www.osix.net/modules/article/?id=186>]
- Software review [http://en.wikipedia.org/wiki/Category:Software_review]
- Benchmarking [http://en.wikipedia.org/wiki/Benchmarking#Metric_Benchmarking]
- Fishbone diagram [http://en.wikipedia.org/wiki/Ishikawa_diagram]

si/si12.txt · Poslední úprava: 2009/05/02 14:51 autor: Budas

Otázka 13 - Y36SIN

Zadání: Standardy pro dokumentaci požadavků. Trasovatelnost. Lidský faktor. Požadavky v kontextu agilních metodik. Správa požadavků, správa změn požadavků.

Slovníček pojmu

- **požadavek** (requirement)
 - v softwarovém inženýrství znamená požadavek jednotlivou dokumentovanou potřebu, jaký má konkrétní produkt nebo služba být nebo co má dělat
 - jedná se o údaj, který identifikuje nezbytné vlastnosti, schopnosti, charakteristiky nebo kvality systému tak, aby tento systém měl hodnotu a byl užitečný pro uživatele
- **specifikace požadavků na software** (software requirements specification, SRS)
 - kompletní popis chování systému, který má být vyvýjen
 - zahrnuje množinu případů užití (use case), které popisují veškeré interakce uživatelů se systémem
 - případy užití bývají také často označovány jako **funkční požadavky**
 - kromě případů užití obsahuje SRS také nefunkční (nonfunctional/supplementary) požadavky
 - nefunkční požadavky zavádějí omezení pro návrh nebo implementaci systému (výkon, kvalita, návrhová omezení)
- **IEEE** (Institute of Electrical and Electronics Engineers)
 - „Institut pro elektrotechnické a elektronické inženýrství“
 - mezinárodní nezisková profesionální organizace usilující o vzestup technologie související s elektrotechnikou
 - organizace stála za vznikem mnoha standardů, pro oblast řízení požadavků je důležitý standard **IEEE 830-1998** (rozebrán dále)
- **trasovatelnost** (traceability)
 - úplně obecně - schopnost chronologicky propojit jedinečně identifikovatelné entity přesně určeným způsobem
 - z hlediska softwarového inženýrství - pod pojmem trasovatelnost rozumíme schopnost popsat a sledovat „život“ požadavků a to jak ve směru dopředném tak i zpětném (např: od jejich specifikace až po návrh systému a dodání systému a naopak bychom měli být schopni dojít od již hotového systému přes přesně definovanou dokumentaci na počátek k jednotlivým požadavkům)
 - jiná definice tohoto pojmu uvádí - „Na poli inženýrství softwarových požadavků je trasovatelnost o porozumění, jakým způsobem jsou vysokoúrovňové požadavky - cíle, záměry, účely, snahy, očekávání, potřeby - transformovány do nízkoúrovňových požadavků. Týká se tedy primárně vztahů mezi vrstvami informací.“
- **matice trasovatelnosti** (traceability matrix)
 - tabulka, která určuje vztahy mezi dvěma dokumenty (představujícími různé úrovně řešené

- problematiky - např. dokument popisující analýzu a dokument popisující návrh systému)
- k úplnému vyjádření vztahů mezi dokumenty je často potřeba vztahů M:N (many to many) - například jeden funkční požadavek (use case) vyústí v několik různých modulů a naopak několik modulů může být nakonec dodáváno jako jedna komponenta
- je často používána pro propojení vysokoúrovňových požadavků (též „marketingových požadavků“) s detailními požadavky na softwarový produkt a se souvisejícími částmi vysokoúrovňového návrhu, nízkoúrovňového návrhu, testovacích plánu a testovacích případů (test case)
- **lidský faktor** (human factor)
 - lidský činitel
 - musí být brán v úvahu při návrhu softwarových systémů
 - týká se zejména návrhu uživatelských rozhraní a rozhraní člověk-stroj (usability)
 - představuje slabý článek ve většině procesů (je potřeba ochrana proti selhání lidského faktoru jako součást požadavků na software)
- **metodika pro vývoj software** (software development methodology)
 - souhrn doporučených praktik a postupů, pokrývajících celý životní cyklus vytvářené aplikace
 - existuje více různých metodik pro vývoj software, různé metodiky jsou různě vhodné pro různé typy projektů (unified process (UP), rational unified process (RUP), extreme programming (XP) ...)
 - metodika sestává z „vývojové filozofie“ (způsob přístupu k procesu vývoje softwaru) a nástrojů, modelů a metod, které mají s vývojem software pomoci (dohromady tvoří kostru vývojového procesu)
- **agilní metodiky** (agile methodologies)
 - označuje skupinu metodik pro vývoj software založených na iterativním vývoji
 - požadavky a řešení se vyvíjí skrze spolupráci mezi samoorganizujícími se týmy, v nichž jsou lidé z různých oblastí znalostí (např: kromě programátorů jsou v týmu i lidi z marketingového oddělení, z oddělení financí, ...)
 - agilní metodiky obecně prosazují časté revize kódu a adaptaci na nové podmínky
 - velký důraz je kladen na týmovou práci a zodpovědnost
- **správa požadavků** (requirements management)
 - proces objasnění, zdokumentování, analýzy, prioritizace a odsouhlasení požadavků a následné řízení změn a komunikace s relevantními osobami (zadavatelem, investory)
 - jedná se o proces probíhající po celou dobu projektu
 - požadavek je schopnost, kterou musí produkt (nebo služba) splňovat

Standardy pro dokumentaci požadavků

Dokumentace požadavků se týká standard IEEE 830-1998. Ten udává formát dokumentu shrnujícího všechny požadavky na nově vyvíjený systém. Výsledný dokument se označuje a anglickém jazyce jako **Software Requirements Specifications (SRS)** (hrubý překlad tohoto pojmu může být „specifikace požadavků na software“).

SRS sestává (mimo jiné) hlavně z funkční specifikace (specifikace případů užití), nefunkční (doplňkové)

specifikace a případných diagramů případů užití.

Ukázková kostra SRS

```
1. Introduction
  1.1 Purpose
  1.2 Document conventions
  1.3 Intended audience
  1.4 Additional information
  1.5 Contact information/SRS team members
  1.6 References

2. Overall Description
  2.1 Product perspective
  2.2 Product functions
  2.3 User classes and characteristics
  2.4 Operating environment
  2.5 User environment
  2.6 Design/implementation constraints
  2.7 Assumptions and dependencies

3. External Interface Requirements
  3.1 User interfaces
  3.2 Hardware interfaces
  3.3 Software interfaces
  3.4 Communication protocols and interfaces

4. System Features
  4.1 System feature A
    4.1.1 Description and priority
    4.1.2 Action/result
    4.1.3 Functional requirements
  4.2 System feature B

5. Other Nonfunctional Requirements
  5.1 Performance requirements
  5.2 Safety requirements
  5.3 Security requirements
  5.4 Software quality attributes
  5.5 Project documentation
  5.6 User documentation

6. Other Requirements
  Appendix A: Terminology/Glossary/Definitions list
  Appendix B: To be determined
```

Jednotlivé odstavce jsou číslovány a toto číslování je dále využíváno pro propojování požadavků s následujícími fázemi vývoje systému (aby byl projekt dobře trasovatelný).

Jedná se o ukázku, jak může struktura SRS vypadat. Jinou strukturu najeznete např. na http://en.wikipedia.org/wiki/Software_Requirements_Specification [http://en.wikipedia.org/wiki/Software_Requirements_Specification]. Standard nevyžaduje naprosto přesnou strukturu, ale spíše uvádí, co by dokument měl obsahovat a jak mají být zaznamenávány vztahy mezi jednotlivými entitami SRS.

Způsob zápisu požadavků

Ačkoliv neexistuje UML standard pro zápis požadavků, existuje jistá konvence. Požadavky je potřeba

jedinečně číslovat:

```
<id> The <system> shall <function>  
e. g.: "32 The ATM system shall validate the PIN number."
```

U jednotlivých požadavků se navíc zaznamená zdroj požadavku. Protože jsou požadavky číslovány je možné propojit tyto požadavky pomocí těchto jednoznačných identifikátorů s informacemi o jejich původu. Tyto informace poté umožňují lepší trasovatelnost, neboť v případě změn víme, s kým máme tyto změny řešit a koho/co dalšího tyto změny ovlivní.

Oblasti, které by mělo SRS řešit

Spravný SRS by měl (dle IEEE) řešit následující oblasti:

1. rozhraní (interfaces)
2. funkční schopnosti (functional capabilities)
3. datové struktury/elementy (data structures/elements)
4. bezpečné použití (safety)
5. spolehlivost (reliability)
6. bezpečnost/ochrana soukromí (security/privacy)
7. kvalitu (quality)
8. rezervy a omezení (constraints and limitations)

Vlastnosti dobrého SRS

- **jednoznačnost** - jeden každý požadavek má právě jednu interpretaci (pozor na to, že přirozený jazyk je od principu nejednoznačný)
- **úplnost** - obsahuje všechny důležité požadavky (na funkce, vlastnosti, omezení, ...), definuje odezvy na všechny realizovatelné kombinace vstupních dat (platných i neplatných), splňuje případné příslušné normy, všechny diagramy apod. mají řádné titulky, neobsahuje odkazy typu „bude stanoveno později“ (pokud je tyto odkazy nutno zahrnout, musí dokument popsát jejich důvod a způsob + termín odstranění)
- **verifikovatelnost (ověřitelnost)** - jeden každý požadavek je verifikovatelný, tj. „existuje cenově efektivní proces, pomocí něhož je možno ručně nebo strojově ověřit, že software odpovídá tomuto požadavku“ (s důrazem na kvantifikovatelnost ⇒ měřitelnost požadavků)
- **konzistence** - neobsahuje navzájem konfliktní množinu požadavků (ten může nastat při použití různých termínů pro stejný objekt, při konfliktních požadavcích na takový objekt, nebo při časově resp. logicky konfliktních specifikacích dvou akcí)
- **modifikovatelnost** - jeho struktura a styl dovoluje snadné, úplné a konzistentní dpolňování případných změn (dokument proto by měl být psaný konzistentním stylem bez redundancí a obsahovat seznamy pro křížové reference jako obsah a index)
- **trasovatelnost** – je zřejmé, odkud vzešly jednotlivé požadavky, a na každý požadavek je možné se explicitně odkazovat v další dokumentaci (zpětná trasovatelnost = u každého požadavku jsou uvedeny jeho zdroje v předchozí dokumentaci, dopředná trasovatelnost = každý požadavek má

jednoznačnou identifikaci, např. číselnou, důležité při údržbě a dohadech).

Trasovatelnost

Přesné definice, co je to trasovatelnost byly uvedeny již dříve ve slovníku pojmu. Zjednodušeně se můžeme na trasovatelnost dívat jako na vlastnost dobře zdokumentovaného a vedeného projektu, kdy jsme schopni přesně sledovat spojitosti od uživatelských požadavků až po finální testy prováděnými nad hotovým softwarem. Obecně se jedná o M:N vztahy mezi jednotlivými položkami (částmi dokumentace, odstavci, číslovanými požadavky, komponentami, funkcemi, ...). Jeden funkční požadavek se totiž může v fázy analýzy rozpadnout na několik aspektů nebo třeba ve fázi návrhu opět srovnout v jednu komponentu.

Při pohledu na vývoj software jako transformaci mezi jednotlivými modely (analytický model, návrhový model, ...) předsavuje trasovatelnost právě záznam zmíněné transformace (na co se převedou entity z předchozí fáze a opačně z čeho vznikly entity s nimiž se pracuje v následující fázi).

Moderní CASE nástroje mají pro trasovatelnost podporu. Mapování prvků z jedné fáze do druhé lze znázornit prostřednictvím matice trasovatelnosti (traceability matrix). Ta má obecně formu tabulky (může vypadat různě - závisí to na prvcích, které na sebe mapujeme).

	<u>SEQ_01</u>	<u>REQ_01</u>	<u>REQ_02</u>	<u>REQ_03</u>	<u>REQ_05</u>	<u>TC_01</u>	<u>TC_02</u>	<u>TC_03</u>	<u>TC_04</u>	<u>TC_05</u>
<u>SEQ_01</u>		↗							↗	
<u>REQ_01</u>						↗				
<u>REQ_02</u>							↗	↗		
<u>REQ_03</u>								↗	↗	
<u>REQ_05</u>										
<u>TC_01</u>										↗
<u>TC_02</u>										
<u>TC_03</u>										
<u>TC_04</u>										
<u>TC_05</u>										

Výše uvedená ukázka znázorňuje, že z požadavku SEQ_01 byl derivován požadavek REQ_01 a TC_04. Obecně může matice mapovat jakékoli prvky na jakékoli jiné.

Lidský faktor

Lidský prvek má při specifikaci požadavků nezanedbatelný dopad. Již při návrhu musíme počítat s některými omezeními, která pro systém člověk představuje:

- **udžovatelnost systému** (maintainability)
 - specifikace požadavků na udržovatelnost (aby produkt byl pro administátory zvladatelný)
- **návrh uživatelského rozhraní**
 - uživatelské rozhraní musí respektovat potřeby uživatele (někdy se hovoří o user-centered návrhu, kdy v centru pozornosti stojí potřeby uživatele)

- požadavky na snadnost ovládání, dostupnost pro lidi se zvýšenými nároky (accessibility)
- **testování uživatelského rozhraní**
 - jedná se o nákladnou záležitost (časově, personálně, ...)
 - různé laboratoře pro testování uživatelských rozhraní - usability lab
- **analýza spolehlivosti lidského prvku**
 - lidský faktor představuje v systémech slabý článek (nelze spoléhat na to, že uživatel zadá všechno a správně, ...)
 - je potřeba zajistit ochranu proti chybám

Požadavky v kontextu agilních metodik

Tradiční metodiky kladou velký důraz na sbírání požadavků v počátečních fázích projektu. V následujících fázích se již uvažuje jen zpřesňování a doplňování existujících funkčních specifikací (které je ale spíše nežádoucí).

V agilních metodikách je očekáváno, že se požadavky budou v průběhu projektu ještě upřesňovat a doplňovat.

Na začátku procesu v agilních metodikách stojí vytyčení vysokoúrovňových požadavků. Následuje vytvoření modelu použití (usage model), který může být např. skupina use case diagramů.

Po vytvoření základního modelu použití se vytvoří základní doménový model - v agilních metodikách je typické použití CRC karet. Tyto karty znázorňují jednotlivé třídy, jejich zodpovědnosti a dále třídy, s nimiž spolupracují na dosahování určitých cílů:

Class Name	
Responsibilities	Collaborators

Další možností záznamu základního doménového modelu je klasický UML diagram.

Poslední částí úvodní etapy je naprototypování uživatelského rozhraní - to mohou být buď nějaké nákresy, screenshoty vyvořené s použitím specializovaného nástroje pro návrh uživatelského rozhraní nebo eventuálně kompletní prototyp uživatelského rozhraní.

Následuje další velmi podobná iterace. Při každé iteraci se hojně komunikuje se zadavateli a zpřesňují se požadavky - zadavateli se snadněji specifikují požadavky, když vidí jak takový systém bude zhruba vypadat.

U požadavků se v agilních metodikách vždy hledí na jejich původ. Požadavky by měly být vždy ze strany

zadavatelů (vývojáři mohou něco navrhnut, ale zadavatel musí tento požadavek vzít za své) a měly by být prioritizovány. V agilních metodikách se počítá s aktivním přístupem zadavatele. Požadavky s vyšší prioritou jsou modelovány do větších detailů.

Správa požadavků, správa změn požadavků

Správa požadavků

Základním požadavkem na správu požadavků je jejich trasovatelnost. Všechny požadavky tedy musí být jednoznačně označeny a musí být dokumentovány veškeré změny.

Správa požadavků v různých fázích projektu:

- „**průzkum**“ (investigation)
 - shromažďování požadavků
 - nikdy nelze shromáždit všechny požadavky na začátku projektu
- „**analýza životoschopnosti**“ (feasibility)
 - je ekonomicky smysluplné implementovat všechny požadavky?
 - má projekt smysl?
 - jaká je cena této konkrétní vlastnosti systému?
- „**návrh**“ (design)
 - v průběhu návrhu se kontroluje, zda navrhovaný systém souvisí s požadavky
- „**konstrukce a testování**“ (construction and test)
 - ověření, že implementovaný systém funguje dle požadavků
 - přezkoumání rozpočtu a cen jednotlivých požadavků (nyní již jsou známy člověkohodiny a další metriky vytvářených atributů systému)
- **uvolnení projektu** (release)
 - zdánlivě konec řešení požadavků
 - po nasazení se teprve začnou ukazovat nové požadavky, které jsou vstupem do první fáze

Správa změn požadavků

V průběhu projektu podávají jednotliví účastníci softwarového projektu (investoři, zadavatelé, zainteresované osoby, ...) požadavky na změny. Jedná se o přirozený jev, s nímž je nutné počítat a je potřeba jej zvládnout, protože představuje pro úspěch projektu rizika.

Systémový analytik nejprve oklasifikuje typ požadavků na změny. Požadavky rozdělí podle typů do několika kategorií jako např:

- nové požadavky
- požadavky na odstranění chyb software
- změny stávajících požadavků
- přidání nové funkcionality

Požadavky se poté standardním způsobem zdokumentují (metodami popsanými výše - SRS, číslování

kvůli trasovatelnosti, ...) a předají se ke zhodnocení revizorovi požadavků (pracovníci, kteří odpovídají za části systému, kterých se změna dotkne).

V případě, že jsou požadavky relevantní a z hlediska časového plánu a rozpočtu proveditelné, zajistí se jejich zapracování do projektu (včetně veškeré standardní dokumentace a se zajištěním trasovatelnosti projektu).

Stručně řečeno - je potřeba provést následující kroky:

1. identifikaci závislostí (jakých částí se změny požadavků dotknou)
2. úprava struktury požadavků
3. revize požadavků

Správa změn z hlediska agilních metodik

- častá komunikace s investory a postupné doplňování požadavků
- zmražení požadavků pro jednu iteraci, aby byla vývojářům poskytnuta stabilita prostředí
- funkční software je základním měřítkem postupu
- na základě prioritizace požadavků se zpracují nejprve požadavky s vysokou prioritou, díky čemuž se maximalizuje ROI (ekon. ukazatel return on investment - návratnost investic) investorů

Zdroje

Pozn: v následujícím seznamu je za odkazy uvedeno vždy několik klíčových slov nebo spojení, které odkaz blíže charakterizují:

- <http://www.feld.cvut.cz/education/bk/predmety/01/88/p18895.html> [<http://www.feld.cvut.cz/education/bk/predmety/01/88/p18895.html>] - syllabus předmětu Y36SIN
- EDUX [http://edux.felk.cvut.cz/modules/edux/course_detail.php?action=display&ID=34] - oficiální materiály k předmětu Y36SIN
- IEEE (cs wikipedia) [<http://cs.wikipedia.org/wiki/IEEE>], (en wikipedia) [<http://en.wikipedia.org/wiki/IEEE>] - standardy
- <http://www.fit.vutbr.cz/study/courses/MSP/public/pro-vytah.html> [<http://www.fit.vutbr.cz/study/courses/MSP/public/pro-vytah.html>] - SRS
- <http://www.techwr-l.com/techwhirl/magazine/writing/softwarerequirementspecs.html> [<http://www.techwr-l.com/techwhirl/magazine/writing/softwarerequirementspecs.html>] - SRS
- http://standards.ieee.org/reading/ieee/std_public/description/se/830-1998_desc.html [http://standards.ieee.org/reading/ieee/std_public/description/se/830-1998_desc.html] - SRS
- http://en.wikipedia.org/wiki/Software_Requirements_Specification [http://en.wikipedia.org/wiki/Software_Requirements_Specification] - SRS
- <http://www.komix.cz/upload/noviny2003b.pdf> [<http://www.komix.cz/upload/noviny2003b.pdf>] - trasovatelnost
- <http://www.rene-witte.net/traceability-cascon2007> [<http://www.rene-witte.net/traceability-cascon2007>] -

trasovatelnost

- http://en.wikipedia.org/wiki/Traceability_matrix [http://en.wikipedia.org/wiki/Traceability_matrix] - trasovatelnost
- <http://www.stickyminds.com/getfile.asp?ot=XML&id=6051&fn=XUS2242742file1%2Edoc> [http://www.stickyminds.com/getfile.asp?ot=XML&id=6051&fn=XUS2242742file1%2Edoc] - matice trasovatelnosti
- http://en.wikipedia.org/wiki/Requirements_Traceability [http://en.wikipedia.org/wiki/Requirements_Traceability] - trasovatelnost
- <http://blog.e-lm.com/?p=6> [http://blog.e-lm.com/?p=6] - trasovatelnost, správa změn požadavků
- http://en.wikipedia.org/wiki/Requirements_management [http://en.wikipedia.org/wiki/Requirements_management] - trasovatelnost, správa požadavků
- http://reliability.sandia.gov/Human_Factor_Engineering/human_factor_engineering.html [http://reliability.sandia.gov/Human_Factor_Engineering/human_factor_engineering.html] - lidský faktor
- http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1658569 [http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1658569] - lidský faktor
- <http://www.springerlink.com/content/112p27122068h444/fulltext.pdf?page=1> [http://www.springerlink.com/content/112p27122068h444/fulltext.pdf?page=1] - lidský faktor
- http://en.wikipedia.org/wiki/Agile_software_development [http://en.wikipedia.org/wiki/Agile_software_development] - agilní metodiky
- <http://www.agilemodeling.com/essays/agileRequirements.htm#Overview> [http://www.agilemodeling.com/essays/agileRequirements.htm#Overview] - požadavky v agilních metodikách
- <http://www.noop.nl/2008/07/the-definitive-list-of-software-development-methodologies.html> [http://www.noop.nl/2008/07/the-definitive-list-of-software-development-methodologies.html] - vývojové metodiky (přehled)
- http://en.wikipedia.org/wiki/Requirements_analysis [http://en.wikipedia.org/wiki/Requirements_analysis] - řízení požadavků a jejich změn
- <http://www.agilemodeling.com/essays/changeManagement.htm> [http://www.agilemodeling.com/essays/changeManagement.htm] - řízení požadavků a jejich změn
- <http://vondrak.cs.vsb.cz/download.html> [http://vondrak.cs.vsb.cz/download.html] - softwarové inženýrství obecně

¹⁾

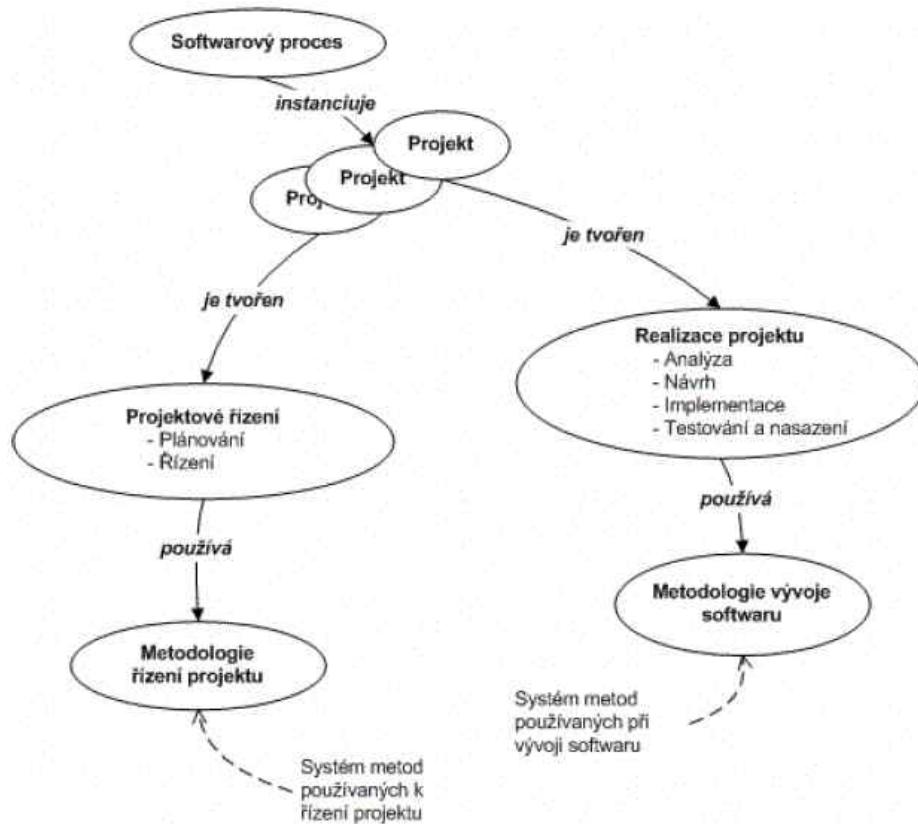
principy agilních metodik jsou shrnutý v dokumentu Agile Manifesto [http://agilemanifesto.org/] z roku 2001

Otázka 14 - Y36SI2

Zadání: Softwarový proces: standardy, nasazení, zajištění. Správa projektů, správa požadavků. Zjišťování a uspořádání požadavků, odhad ceny, plánování, monitorování a trasování projektů, správa rizik, řízení projektů a plánování změn

Softwarový proces

Softwarový proces je postup činností nutných k vytvoření softwarového produktu. Dle takto stanoveného předpisu jak vytvářet software se následně definují projekty (hovoří se o tzv.: instanciaci procesu) vztázené k jednotlivým zakázkám. Tyto základní pojmy ze softwarového inženýrství nejlíp popisuje následující obrázek:



Obr. 1: Sémantický graf vývoje softwarového produktu

Standardy

- IEEE 1042 definuje pojmy a procesy řízení konfigurací (změn)
- IEEE 1058 definuje vztahy mezi jednotlivými účastníky procesu.
- IEEE 828 je normou pro psaní Plánů řízení SW konfigurací (Software Configuration Management Plans – SCMP).

- *IEEE 1074* je normou pro tvorbu životního cyklu procesů. Definuje soubor aktivit a procesů povinných pro vývoj a údržbu softwaru. Definuje procesy a procesní skupiny.
 - Procesní skupiny jsou:
 - model životního cyklu
 - řízení projektu
 - „předvývoj“
 - vývoj
 - „povývoj“
 - integrace
 - Procesy jsou např. : specifikace požadavků, návrh, implementace.

Modely softwarového procesu

- Vodopád = Jednotlivé aktivity jsou zpracovány jako nezávislé procesy, které na sebe navazují.
- Prototypování = na prototypech modelovány vlastnosti systému, vhodné pro menší systémy
- RAD model - Rapid Application development = Lineární sekvenční model, krátký vývojový cyklus, použití znovupoužitelných komponent a rozdělení do modulů
- Přírůstkový model = evoluční - Tvorba po funkčních částech, první se nazývá jádro. Model vhodný pro malý tým a velký úkol.
- Spirálový model = evoluční - upřednostňuje tvorbu verzí s větší rizikovostí, vhodný pro velké systémy
- Model skládání komponent = evoluční - spirálový model pro objektové technologie
- Model souběžného vývoje = evoluční - jednotlivé komponenty vyvíjeny paralelně, vhodné např. pro klient-server aplikace
- Formální metody - spočívají ve formální specifikaci a verifikaci programů, časově náročné
- Techniky čtvrté generace - programování na vysoké úrovni abstrakce, rychlý vývoj, ale některé prostředky složité

Nasazení a zajištění

- Nasazení je proces instalace hotového softwarového projektu u zákazníka, může obnášet i instalaci hardwaru
- Zajištění je proces po nasazení, tzn. podpora, zpětná vazba od uživatelů, aktualizace softwaru atd.

Správa projektů

Správa projektů je způsob organizace projektu pomocí nástroje pro správu projektu. Známé systémy pro správu projektů:

- SourceForge
- Google Code
- BugZilla

- Trac
- ...

Systém pro správu projektů obvykle obsahuje:

- systém pro organizaci práce - milestones, roadmap atd.
- wiki pro psaní dokumentace
- systém pro správu požadavků
- repozitář pro správu kódu
- fórum nebo jiné způsoby podpory a komunikace

Systém pro správu projektů může obsahovat jenom některé komponenty nebo může být složen z několika systémů na specializované funkce (např. BugZilla na bug tracking a MediaWiki na dokumentaci)

Správa požadavků

Požadavek je požadavek na práci člena teamu. Např. nahlášený problém, požadavek na novou funkčnost, úkol, bug atd. Systém pro správu projektu by měl mít systém pro správu požadavků. Anglicky ticket, issue, task.

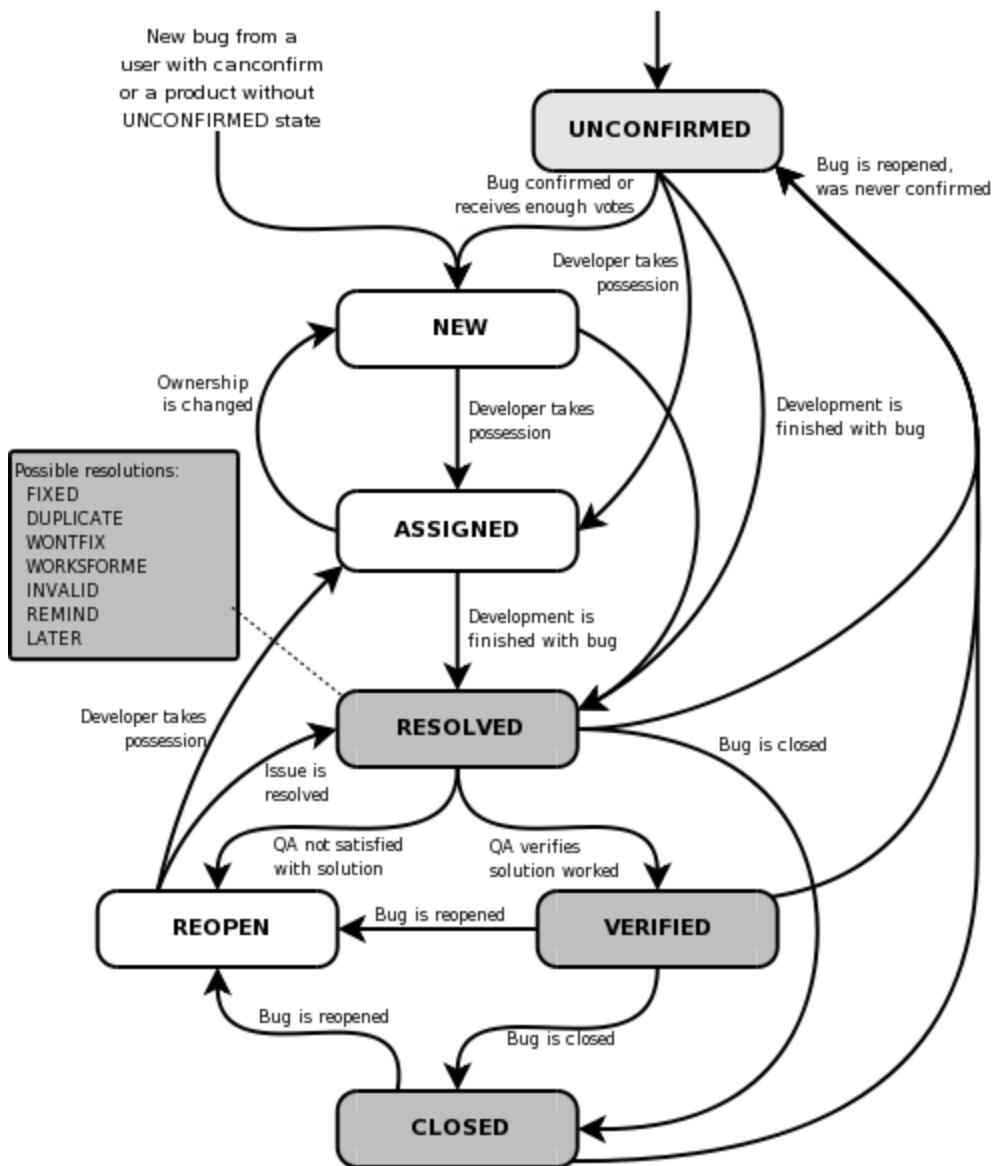
Zjišťování požadavků probíhá:

- od zákazníka
- při zadávání práce členům týmu
- při testování

Uspořádání požadavků:

- Požadavky mají různý typ, např.: chyba, vylepšení, úkol
- Požadavky mají různou prioritu, např.: trivial, minor, major, critical, blocker
- Požadavky mohou náležet k různým komponentám jako např.: database, user interface, organization, presentation, kernel, ...
- Požadavky mají různé stavy jako např.: new, reopened, assigned, invalid, duplicate, fixed
- Požadavky se také přiřazují jednotlivým lidem, k milestoneům nebo k verzím programu

Životní cyklus bugu v Bugzille:



Plánování

Plánování je popis toho, co chceme, aby se stalo (nikoli toho, co se stane).

Proces plánování

- stanovení cílů a definování strategie vedoucí k jejímu dosažení
- zpracování strukturované dekompozice činností projektu
- vytvoření projektové organizační struktury a sestavení projektových týmů
- zpracování implementačních plánů projektu, tj. časových plánů, plánů nákladů, alokace zdrojů
- specifikace nástrojů a technik pro řízení projektu
- identifikace možných omezení, rizikových oblastí a návrh způsobů eliminace těchto vlivů

Odhad ceny

Existuje několik technik, přesnější odhadů získáme porovnáním více technik. Vše závisí na dobrých historických datech!

Máme dvě hlavní kategorie odhadů:

Dekompozice

Dekompozice je rozdělení hlavních funkcí a odhad velikosti nebo pracnosti implementace každé funkce

Tyto techniky používají veličiny LOC – počet řádků kódu a FP – funkční bod. LOC a FP se používají:

- jako proměnné pro odhad různých veličin v projektu
- jako základní údaje o minulých projektech

K odhadu používáme tzv. *tříbodový odhad*, který je dán vzorečkem:

$$EV = \frac{(S_{opt} + 4S_m + S_{pes})}{6}$$

- S_{opt} je optimistický odhad
- S_m je střední odhadovaná hodnota
- S_{pes} je pesimistický odhad

Odhad pomocí LOC

Z historických údajů víme, že průměrná produktivita je např 620 LOC/mm (řádků kódu na jeden člověkoměsíc). Odhadnem počet řádků, odhadnem plat programátora a dopočítáme kolik bude stát produkt.

Odhad pomocí FP

Podle složitosti jednotlivých dílčích modulů těmto modulům určíme na základě tabulky počet funkčních modulů (tato tabulka určuje počet vstupů, výstupů, dotazů, rozhraní a vnitřních logických souborů), zároveň si spočítáme proměnnou, která odpovídá na otázky spojené s náročností celého systému. Tyto hodnoty poté dosadíme do kouzelného vzorce (který nemusíme znát) a vypadne nám počet fukčních bodů na celý SW. Z historických údajů víme, kolik nás stojí jeden bod a na základě toho spočítáme cenu projektu. Pokud tyto údaje nemáme, tak se podíváme do tabulek, kolik řádek kódu obvykle na FP připadá v daném programovacím jazyce a na základě znalosti ceny řádky kódu spočítáme cenu projektu.

Empirické modely

Empirické modely jsou odvozené formule pro pracnost a čas.

Pro odhad pracnosti existuje celá řada empirických modelů, ve kterých vystupují veličiny FP nebo LOC vycházející z řešeného projektu a empiricky odvozené konstanty z předchozích projektů A, B, C. Při jejich

použití je vhodné provést porovnání odhadu podle několika modelů.

Obecný tvar rovnice modelů je

$$\mathbf{E} = \mathbf{A} + \mathbf{B} (\mathbf{ev}) \mathbf{C}$$

kde E je pracnost

A, B, C jsou empiricky odvozené konstanty

ev je proměnná udávající hodnotu LOC nebo FP

$E = 5,2 (\text{KLOC})^{0,91}$	Walson-Felixův model
$E = 5,5 + 0,73(\text{KLOC})^{1,16}$	Bailey-Basiliho model
$E = 3,2 (\text{KLOC})^{1,05}$	Boehmův jednoduchý model
$E = 5,288 (\text{KLOC})^{1,047}$	Dotyův model pro KLOC > 9
$E = -13,39 + 0,0545 \text{ FP}$	Albrecht a Gaffneyův model
$E = 60,62 \times 7,728 \times 10^{-8} \text{ FP}^3$	Kemererův model
$E = 585,7 + 15,12 \text{ FP}$	Matson, Barett a Mellichampův model

Model COCOMO (COnstructive COst MOdel)

Za nejpracovanější a nejpoužívanější empirický model můžeme považovat model COCOMO.

Jsou definovány tři třídy projektů, které v modelech COCOMO rozdělujeme:

- Organický mód - malé týmy, přizpůsobivé požadavky
- Přechodný mód - středně velké týmy, mix mezi nepřizpůsobivými a přizpůsobivými požadavky
- Uzavřený mód - nepřizpůsobivé požadavky (vazba na konkrétní hardware, software, prostředí)

Model COCOMO je definován ve třech úrovních:

- Základní COCOMO model - pracnost a cena jako funkce velikosti programu v LOC.
- Střední COCOMO model - pracnost a cena jako funkce velikosti programu v LOC a množiny dalších faktorů (produkt, HW, lidé, projekt).
- Pokročilý COCOMO model – má navíc odhad faktorů každé etapy softwarového procesu.

Příklad

Projekt našeho typu potřebuje přibližně 33,2 KLOC (tisíců řádků kódu). Použijeme rovnici základního modelu. Tam se pracnost vyjádří $E = a \text{KLOC}^b$ a doba $D = c(E) d$. Koeficienty a, b, c, d použijeme z tabulky pro přechodný mód: 3,0; 1,05; 2,5 a 0,35.

Výpočet:

$$E = 3 \cdot (33,2)^{1,05} = 120 \text{ mm}$$

$$D = 2,5 \cdot (120)^{0,35} = 15,3 \text{ m}$$

Doporučený počet osob:

$$N = E/D = 120 / 15,3 = \text{asi } 8 \text{ lidí}$$

Monitorování

Během vývojového cyklu by se mělo monitorovat a zaznamenávat:

- plnění cílů
- chyby a nedostatky
- výkon aplikace a nároky na HW
- výsledky testů
- personální změny

Trasovatelnost

Trasovatelnost (Traceability) je vlastnost dobře zdokumentovaného a vedeného projektu, kdy jsme schopni přesně sledovat spojitosti od uživatelských požadavků až po finální testy prováděnými nad hotovým softwarem. Obecně se jedná o M:N vztahy mezi jednotlivými položkami (částmi dokumentace, odstavci, číslovanými požadavky, komponentami, funkcemi, ...). Jeden funkční požadavek se totiž může v fázi analýzy rozpadnout na několik aspektů nebo třeba ve fázi návrhu opět splynout v jednu komponentu.

Při pohledu na vývoj software jako transformaci mezi jednotlivými modely (analytický model, návrhový model, ...) předsavuje trasovatelnost právě záznam zmíněné transformace (na co se převedou entity z předchozí fáze a opačně z čeho vznikly entity s nimiž se pracuje v následující fázi).

Trasování

Trasování (Tracing) je speciální použití logování běžícího programu. Logy používají primárně na debuggování vývojáři, ale také jako zdroj informací pro administrátory a zákaznickou podporu. Narození od *event loggingu* je tracing vytvářen vývojáři a je více low-level (event logging je spíš pro administrátory, v event logu bude že program selhal, v trace logu bude že program vyhodil tu a tu exception).

Příklady:

- debug výpisy z kódu do logu
- Windows software trace preprocessor (aka WPP)
- Linux system level and user level tracing s Kernel Markers a LTTng

Rizika

Riziko je nejistá událost, která má nějaký dopad na daný objekt nebo proces. Riziko se měří v pojmech pravděpodobnosti a dopadu.

Kategorie rizik

- *Projektová rizika*, která se týkají plánu projektu: problémy rozpočtu, harmonogramu, zdrojů, zákazníka, jeho požadavků a jejich dopadů na projekt.
- *Technická rizika*, která se týkají kvality produktu, potenciální rizika návrhu, implementace, verifikace, údržby produkovaného softwaru.
- *Obchodní rizika*, která mohou být například:
 - vybudování skvělého produktu, který nikdo nechce (marketingové riziko)
 - vybudování produktu, který už nezapadá do obchodní strategie firmy (strategické riziko)
 - vybudování produktu, kterému obchodní zástupci nerozumí a neví, jak ho prodat
 - ztráta podpory vedení, vlivem změny zaměření nebo změny osob (riziko managementu)
 - ztráta rozpočtu (rozpočtové riziko)
- *Rizika spojená s velikostí produktu* se projevují u velkých projektů, pokud bude systém používat enormní množství uživatelů atd.
- *Rizika obchodního dopadu* se projevují po dodání produktu. Řeší se rizika jako pozdní dodání, nedostatečné zaškolení uživatelů, omezení státními normami atd.
- *Rizika spojená se zákazníkem* vyplývají z neochoty zákazníka spolupracovat, nedostatečné komunikace nebo naopak z přílišného zájmu, kdy si zákazník myslí, „že ví přesně jak to udělat“
- *Procesní rizika* vyplývají ze špatně definovaného procesu a použití nedostatečných nebo špatných technik. Maximální používání dostupných standardů snižuje tato rizika.
- *Rizika vývojového prostředí* vyplývají ze zvoleného vývojového prostředí, využití jeho funkcí, zaškolení vývojářů atd.
- *Rizika spojená s velikostí týmu a jeho zkušeností*: Nárast produktivity není lineární s přibývajícím počtem lidí, naopak vzrůstají rizika způsobená špatnou komunikací a organizací práce.

Ohodnocení rizik

Rizika si vypíšeme do tabulky. Přiřadíme jím kategorie. Ohodnotíme pravděpodobnost rizik. Ohodnotíme dopad rizika čísly:

1. katastrofický
2. kritický
3. marginální
4. zanedbatelný.

Seřadíme tak, aby nejpravděpodobnější s největším dopadem byly nahoře.

Příklad:

<i>rizika</i>	<i>kategorie</i>	<i>pravděpodobnost</i>	<i>dopad</i>	<i>plánovaná opatření</i>
zákazník změní požadavky	zákazník	80%	2	
fluktuace členů týmu	projektový tým	70%	2	
nízký odhad velikosti	velikost produktu	60%	2	
nedostatečné školení práce ve vývojovém prostředí	vývojové prostředí	20%	3	
...				

Protiriziková opatření

Proti nejvýznamnějším rizikům bychom měli mít připraveny protiriziková opatření. Mezi ně patří:

- Havarijní plány
- Alternativní strategie
- Rezervy
- Obstarávání (provedení externí firmou)
- Pojištění

Plánování změn

Řízení softwarových konfigurací (změn) označujeme jako SCM (Software Configuration management) a má za cíl:

1. identifikovat změny
2. řídit změny
3. zajistit, aby změny byly rádně implementovány
4. informovat o změnách ty, kterých se to týká

SCM je aktivita, která začíná na začátku softwarového projektu a končí až, když je software vyřazený z provozu.

Pojmy z SCM:

- *SCI (Software Configuration Item)*: program/dokumentace/data, která jsou výstupem nějaké etapy nebo projektu
- *Baseline (Srovnávací základna)* – specifikace nebo produkt (SCI) – revidovaný a odsouhlasený

Řízení změn:

- Neformální řízení změn = předtím, než se SCI stane srovnávací základnou – vyvojář pouze zjistí,

- zda jsou změny v souladu s technickými požadavky
- Řízení změny na projektové úrovni = když SCI je projektová základna, je potřeba povolení a schválení od projekt. managera
- Formální řízení změn = po dodání produktu zákazníkovi – nutné rozhodnout, jak změna ovlivní chování systému a kvalitu

Plánování a řízení projektu

Při řízení projektu se nesmí zapomenout na:

- tvorbu plánu projektu
- stanovení a kontrolování milestones (milníků)
- plánování času (platí pravidlo 90-90: když to vypadá že je 90% práce hotovo, 90% ještě zbývá)
- správné rozložení pracnosti (mělo by platit pravidlo 40-20-40: 40% analýza, 20% implementace, 20% testování)
- správný stupeň *rigóznosti*. Rozeznáváme tyto stupně rigoróznosti:
 - neformální - minimální množina úkolů, redukovaná dokumentace
 - strukturovaný - rámcové aktivity a příslušné úkoly
 - přesný - úplný proces, potřeba vysoké kvality, robustní dokumentace
 - rychlá reakce - vzhledem k nouzové situaci jsou aplikovány jen ty aktivity, které zajišťují dobrou kvalitu, úplná dokumentace je dodálena dodatečně po dodání produktu

Sledování plánu

- Matice zodpovědností - matice lidí a činností. Jako hodnoty jsou: P-zodpovídá, O-provádí, K-konzultuje, S-schvaluje
- Síť úkolů, CPM - metoda kritické cesty, úkoly dáme podle závislostí do grafu a najdeme kritickou cestu - nejdelší cestu v gragu
- Časový diagram (Úsečkový graf, Ganttův diagram) - zobrazuje úkoly, jejich časovou náročnost a závislost v kalendáři
- Tabulka zdrojů (sloupcový graf - histogram) - určuje vytíženosť zdrojů v jednotlivých dnech v závislosti na úkolech

Zdroje

- Skripta Y36SI2 [http://www.stm-wiki.cz/index.php/Soubor:Y36SI2_skripta.pdf]
- Úvod do softwarového inženýrství, aneb co všechno neznáme - seriál na webu owebu.cz [<http://www.owebu.cz/filozofie/vypis.php?clanek=385>]
- Vypracované otázky k testu Y36SI2 - STM Wiki [http://www.stm-wiki.cz/index.php/Y36SI2_Vypracovan%C3%A9_ot%C3%A1zkы_k_testu]
- How do I use Bugzilla? [<http://www.bugzilla.org/docs/2.16/html/how.html>]
- Trasování (software) - Wikipedie [[http://cs.wikipedia.org/wiki/Trasov%C3%A1n%C3%AD_\(software\)](http://cs.wikipedia.org/wiki/Trasov%C3%A1n%C3%AD_(software))]
- COCOMO - Wikipedia [<http://en.wikipedia.org/wiki/COCOMO>]

- Ganttův diagram - Ing. Miroslav Lorenc [<http://lorenc.info/3MA381/ganttuv-diagram.htm>]

si/si14.txt · Poslední úprava: 2009/05/02 16:18 autor: Destil

Otázka 15 - Y36SI2

Zadání: Plánování projektů, odhad potřeb, rozpočet. Nástroje pro řízení projektů. Faktory ovlivňující produktivitu a úspěch. Metriky produktivity. Analýza rizik. Plánování a správa změn. Řešení výjimek. Správa verzí, správa konfigurací. Standardy pro proces vývoje a implementace software. Softwarové kontrakty, intelektuální vlastnictví. Přístupy k údržbě a dlouhodobému vývoji SW.

Slovníček pojmu

Projekt je časově ohraničené úsilí vynaložené s cílem vytvořit jedinečný výsledek (produkt).

Proces je ucelený sled činností, který má na výstupu měřitelný výsledný produkt.

Produkt může být hmotný (výrobek) nebo nehmotný (informace) nebo to může být služba.

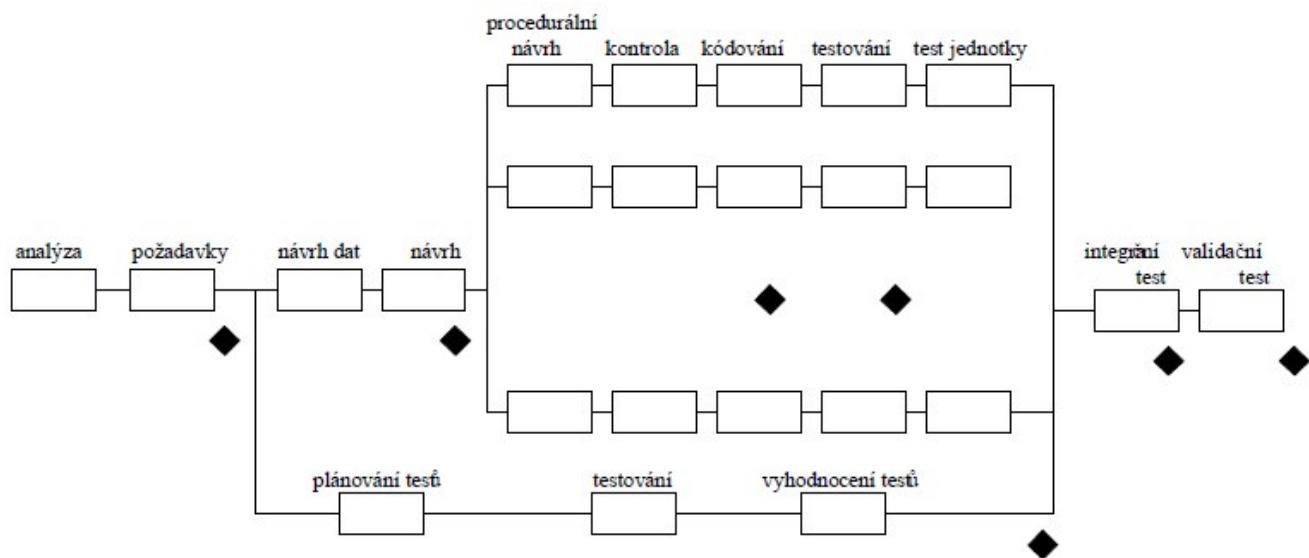
Plánování projektů

Se skládá ze dvou hlavních aktivit:

1. Zjistit, co vše bychom měli zákazníkovi předat: dokumentace, předvedení funkce, návrh subsystémů, důkaz přesnosti, prezentace efektivity, bezpečnosti a realizace, ...
2. Stanovit tzv. milníky (kontrolní dny), neboli rozčlenit projekt na fáze, kroky a aktivity.

Fáze, kroky a aktivity projektu

	fáze 1	krok 1	aktivita 1.1	
		krok 2	aktivita 1.2	
		
projekt	fáze 2	krok 1	aktivita 2.1	
		krok 2	aktivita 2.2	
		
	fáze 3	krok 1		
		krok 2		
		

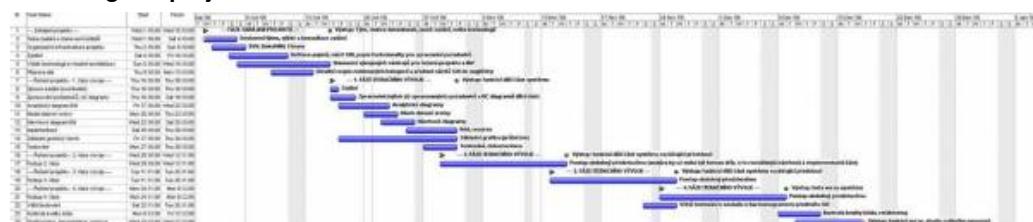


◆ milníky

Pro správné plánování je zapotřebí zejména:

- jasná koncepce produktu, detailní specifikace - pouze obecné zadání nestačí
- komunikace - jak se zákazníky, tak s realizačním týmem
- určit, jak bude zajištěna kvalita a jak budou řízeny změny
- definovat rizika a navrhnout techniky jejich řízení
- definovat cenu a rozvrh kontrol

Harmonogram projektu



Struktura plánu projektu

1. rozsah projektu a cíle
2. projektové odhady
3. strategie řízení rizik
4. časový harmonogram
5. zdroje projektu
6. organizační členění
7. způsob kontroly

Plánování času

- **Pravidlo 90-90:** Je-li odhadováno, že práce je hotová z 90%, pak ve skutečnosti zbyvá dokončit ještě 90% práce.
- **Brookův zákon:** Přidání programátorů do opožděného SW projektu zvýší jeho zpoždění.
- **Pravidlo 40-20-40:** 40 % času má trvat analýza, 20 % času programování, 40 % času testování.

- **Nespěchat s realizací** - čím dříve začneme psát program, tím později bude hotov. 50% - 70% práce na programu jsou až po předání zákazníkovi.

Nástroje pro řízení projektů

Následuje popis některých obecných metod, je na místě však také zmínit konkrétní nástroje typu MS Project nebo Google Code.

Matice zodpovědností

V prvním sloupci tabulky jsou uvedeny **činnosti**, pro které se přiřazují odpovědné subjekty.

V první řádce tabulky jsou uvedeny odpovědné **subjekty** (osoby, role, organizační jednotky).

Uvnitř tabulky jsou uvedeny písmena označující **typy zodpovědnosti**.

Obvyklé typy zodpovědnosti jsou:

P	subjekt je za činnost odpovědný
O	subjekt činnost provádí
K	subjekt má zodpovědnost konzultační
S	subjekt má odpovědnost schvalovací

Příklad:

Aktivita	Leela	Bender	Fry
Údržba lodí	P	O	O
Předání zásilky	P	O	O
Pilotování lodí	PO		K

Síťový graf, CPM

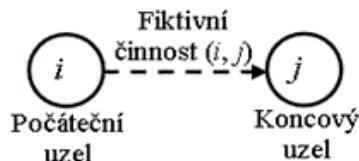
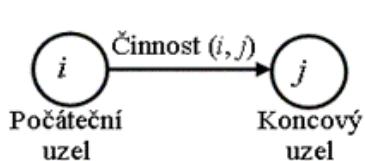
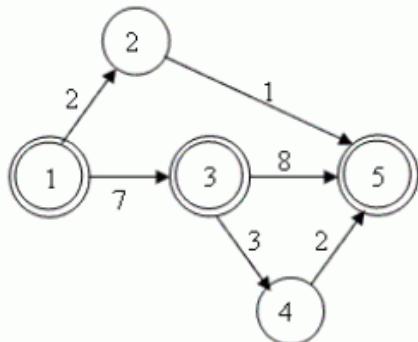
Síťový graf se skládá z uzlů a orientovaných hran. Hrany odpovídají jednotlivým dlíčím činnostem úkolu. Danou činnost jednoznačně určují počáteční a koncový uzel, kterými je každá činnost ohraničena.

více zde [<http://www.fsv.vsb.cz/books/SystAnal/texty/25.htm>]

CPM (Critical Path Method, Metoda kritické cesty)

- stanoví se doba trvání t úkolů (nejpravděpodobnější odhad času podle statistických modelů)
- potom se spočítají hraniční časy pro jednotlivé úkoly podle těchto kriterií:
 1. **B1** - nejdříve možné zahájení úkolu (když předchozí úkoly jsou zvládnuté v minimálním čase)
 2. **B2** - nejpozději přípustný začátek úkolu (anž by došlo ke skluzu projektu)
 3. **E1** - nejdříve možné ukončení úkolu ¹⁾
 4. **E2** - nejpozději přípustný konec úkolu ²⁾
 5. povolený **skluz**
 6. určí se **kritická cesta** (řetěz úkolů, určujících trvání projektu)
- časová rezerva:
 - celková rezerva $RC = E2 - B1 - t$
 - volná rezerva $RV = E1 - B1 - t$
 - nezávislá rezerva $RN = E1 - B2 - t$
- **Kritická cesta** je nejdelší cestou v grafu, ovlivňuje dobu trvání projektu.
 1. Průchodem od začátku do konce se určí nejdříve možné začátky činností (vstupuje-li do určitého uzlu více činností, je to maximální hodnota - max. součet nasbíraných dob trvání).
 2. Průchodem od konce se určí nejpozději přípustné začátky.

3. Pak se určí časové rezervy a kritická cesta. Činnosti kritické cesty mají nulovou časovou rezervu.



Dobu projektu můžeme zkrátit:

- změnou logiky vazeb
- přesunem vnitřních zdrojů (z nekritických do kritických činností)
- nasazením dodatečných zdrojů

Tabulka projektu (project table)

Obsahuje: úkol, jeho zahájení a skončení (plánované i skutečné), přidělené osoby, pracnost, poznámky.

Sloupcový graf zdrojů

Znázorňuje ve sloupcích vytížení zdrojů. Výhodnější je zapojit zdroje, co nejpozději (Metoda *Just in Time*) - projekt tak váže nejmenší objem oběžných prostředků.

Odhad potřeb, rozpočet

Odhady pracnosti, času, zdrojů a nákladů jsou dány:

- složitostí a velikostí projektu
- metrikami minulých projektů
- variabilitou v softwarových požadavcích

Rozsah projektu se stanoví na základě interview se zákazníkem.

Systém schůzek

- **Předběžná řízená schůzka se zákazníkem** zahrnuje 3 doporučené typy otázek:

1. Bezkontextové otázky (context free questions)
 - Např.: Kdo tu práci požaduje? Kdo ji bude užívat? Jaký bude ekonomický užitek při úspěšném ukončení? Je ještě jiná možnost, jak to vyřešit?
2. Otázky vedoucí k hlubšímu pochopení
 - Např.: Jak byste charakterizoval „dobrý“ výstup? Na jaké problémy je toto řešení zaměřeno? Ukažte mi (popište) prostředí, kde to bude systém pracovat? Jsou nějaké speciální požadavky na chování

systému a na jeho omezení?

3. „Meta otázky“

- Např.: Jste ta správná osoba, která mi může na tyto otázky odpovědět? Jsou vaše odpovědi oficiální? Jsou mé otázky relevantní k danému problému? Nedávám vám moc otázek? Je tu ještě někdo další, kdo by mohl poskytnout doplňující informace? Je ještě něco, na co bych se měl zeptat?

■ Další schůzky se více týkají řešení problému, vyjednávání a specifikací zadání. Využívá se techniky FAST (Facilitated Application Specification Techniques), pro kterou platí:

- Jsou přítomni realizátoři i zákazníci, je však řízena neutrální stranou (facilitátorem, tj. odborník na moderování diskuze)
- Určují se pravidla pro přípravu a průběh realizace projektu.
- Je specifikována agenda, která bude vedena.
- Používá se tabule, flip chart apod.
- Cílem schůzky je identifikovat problém a specifikovat požadavky.

Plánování zdrojů

Zdroje jsou lidské zdroje, využitelné SW komponenty, HW a SW nástroje. Je nutné provést identifikaci každého zdroje, jeho popis, zjistit jeho dostupnost, čas, kdy bude požadován a na jak dlouho.

Stále roste význam komponent.

Druhy komponent

1. hotové komponenty (off-the-shelf components)
2. komponenty se zkušeností (full-experience components)
3. komponenty s částečnou zkušeností (partial-experience components) - jejich použití v projektu se musí být detailně analyzováno
4. nové komponenty

Odhad ceny a pracnosti

Odhad je možné provést jako:

- Odhad se zpožděním.
- Počáteční odhad podle minulého podobného projektu.
- Odhad s použitím dekompozičních technik.
- Odhad s použitím empirických modelů.

Přesnost odhadu projektu je ovlivněna:

- přesností odhadu velikosti produktu
- schopností převést odhad velikosti na odhad pracnosti, času, finančních nákladů (závisí na dostupnosti spolehlivých metrik z minulých projektů)
- schopnostmi projektového týmu
- stabilitou požadavků na projekt a vývojovým prostředím

Metody odhadů používají veličiny LOC (počet řádků kódu) a FP (funkční bod). LOC a FP se používají jako proměnné pro odhad různých veličin v projektu a jako základní údaje o minulých projektech.

Máme dvě kategorie odhadů:

- dekompozice - rozdelení hlavních funkcí a odhad velikosti nebo pracnosti implementace každé funkce
- empirické modely - odvozené formule pro pracnost a čas

Přesnější odhady získáme porovnáním více technik. Vše závisí na dobrých historických datech.

Tříbodový odhad

$EV = (s_{opt} + 4 s_m + s_{pes}) / 6$, kde s_{opt} je optimistický odhad, s_m je střední odhadovaná hodnota a s_{pes} je pesimistický odhad.

Např. má-li program $s_{opt} = 1800$ LOC, $s_m = 2400$ LOC a $s_{pes} = 2650$ LOC, pak je podle vzorce $EV = 2341,67$. Pro výpočet nákladů využijeme průměrnou hodnotu LOC/mm (počet řádků kódu na jeden člověkoměsíc), kterou zjistíme z historických údajů.

DISKUSE

Pro **průměrný odhad ceny produktu** se podle příkladu ve skriptech SI2 s výsledkem na straně 47 využívá *odhad LOC, odhad FP a procesní odhad* (průměr z těchto odhadů). V tomto příkladu mi však příje nesprávné průměrovat procesní odhad s odhadem LOC, protože odhad LOC je v podstatě podmnožinou procesního odhadu. Taky vychází podle odhadu LOC: 1 614 000 Kč, a podle procesního odhadu v části kódování (což by měl být ekvivalent) 221000 Kč. Myslíte, že by byl tento postup výpočtu ceny akceptovatelný (i kdyby čísla odhadů jinak odpovídala)?

Mannová: Samozrejme odhad je odhad a jakýkoliv výsledek je jen orientacni. Pokud si ruzne odhady odpovidaji, je odhad lepsi.

Empirické modely odhadu

Existuje celá řada empirických modelů, obecný tvar rovnice modelů je:

$E = A + B(ev)C$, kde E je pracnost; A, B, C jsou empiricky odvozené konstanty; ev je proměnná udávající hodnotu LOC nebo FP

Např.: $E = 585,7 + 15,12 FP$ (*Matson, Barett a Mellichampův model*)

Model COCOMO (COnstructive COst MOdel)

Jsou definovány tři třídy projektů, které v modelech COCOMO rozdělují:

1. Organický mód (malý tým, známé prostředí, známá problematika)
2. Přechodný mód (složitější problémy, komunikace, vyšší rozsah)
3. Uzavřený mód (krátké termíny, omezení, ladění HW, změny požadavků)

Model COCOMO je definován ve třech úrovních:

1. **Základní COCOMO model** - pracnost a cena jako funkce velikosti programu v LOC.
2. **Střední COCOMO model** - pracnost a cena jako funkce velikosti programu v LOC a množiny dalších faktorů (produkt, HW, lidé, projekt).
3. **Pokročilý COCOMO model** – má navíc odhad faktorů každé etapy softwarového procesu.

Rovnice základního modelu

$$E = a (KLOC)^b$$

$$D = c (E)d$$

kde E je pracnost v člověkoměsících, D je doba vývoje v měsících, koeficienty a, b, c, d se odvozují z třídy projektu a jsou uvedeny v tabulce (není potřeba se je učit nazpaměť).

Rovnice středního modelu

$$E = a (KLOC)^b * EAF$$

kde EAF je faktor pracnosti a nabývá hodnot mezi 0.9 a 1.4, jeho stanovení zahrnuje atributy produktu, hw, týmu, projektu.

Odhad pomocí softwarové rovnice

$$E = (LOC \times B^{0,333} / P)^3 * (1 / t^4)$$

kde E je pracnost v mm (člověkoměsíce)

t je trvání projektu v měsících nebo letech

B je faktor zkušenosti (pro KLOC =5..15, je B= 0,16, pro KLOC>70 je B=0,39)

P je parametr produktivity, který odráží:

- celkovou zralost procesu a praktik vedení projektu
- použité praktiky analýzy a návrhu
- úroveň programovacího jazyka
- stav softwarového prostředí
- zkušenosť softwarového týmu
- složitost aplikace
- P = 2000 - systémy reálného času
- P = 10000 - telekomunikační a systémový software,
- P = 28000 - obchodní aplikace

Faktory ovlivňující produktivitu a úspěch

Produktivitu software ovlivňují objektivní i subjektivní faktory, některé lze ovlivnit. Hlavní faktory, které ovlivňují tvorbu softwaru jsou:

- **Lidský faktor** (velikost a zkušenosť firmy)
- Faktor **problému** (složitost problému, počet změn v návrhu omezení a požadavků)
- Faktor **procesu** (techniky analýzy a návrhu, jazyk a CASE, techniky kontroly)
- Faktor **produkту** (chování a spolehlivost systému)
- Faktor **zdrojů** (dostupnost CASE nástrojů, HW a SW zdrojů)

Metriky produktivity

Správnost (correctness) se měří nejčastěji počtem chyb na tisíce řádku kódu (KLOC). Chyby by měly být odhaleny během testování

Udržovatelnost (maintainability) je snadnost, s níž může být program opraven, když se vyskytne chyba, jak rychle může být upraven, změní-li se prostředí, jak může být zdokonalován, rozhodne-li se zákazník upravit požadavky.

Používáme metriku **MTTC** (mean-time-to-change) – střední doba potřebná pro realizaci změn tj. doba potřebná pro analýzu požadovaných změn, návrh, implementaci, otestování a distribuci uživatelům.

Rozeznáváme:

- **Spoilage** - cena za opravu defektů objevených po distribuci.
- **Integrita** - odolnost vůči náhodným nebo záměrným útokům z vnějšího prostředí (viry, hackers).
 - integrita = suma ($1 - \text{hrozba} \times (1 - \text{bezpečnost})$)
 - hrozba je pravděpodobnost útoku speciálního typu
 - bezpečnost je pravděpodobnost odolání útoku speciálního typu
 - suma je součet přes všechny typy útoků
- **použitelnost** (usability) - je snaha změřit „uživatelskou přívětivost“.

dobrý software z pohledu zákazníka:

1. nízká intelektuální dovednost nutná pro zvládnutí systému
2. krátký čas potřebný na to, aby uživatel byl mírně zdatný v ovládání systému
3. vzrůst produktivity vzhledem k systému, který produkt nahrazuje, jestliže je uživatel mírně zdatný
4. subjektivní ocenění uživatelů (dotazníkem)
5. software není konfliktní s dalším software

Měření účinnosti odstraňování chyb software

DRE (Defect Removal Efficiency) = $E / (E + D)$

E je počet chyb nalezených před předáním uživateli

D je počet defektů po předání

Kvalita software

- Metriky vytvářejí možnost srovnání s předchozími projekty.
- Bez jejich měření nelze poznat, zda se kvalita zlepšuje či zhoršuje.

Analýza rizik

Rizika zahrnují nejistoty a pokud nastanou, znamenají ztrátu, většinou vychází z předchzí zkušenosti.

Ukázková tabulka rizik

#	Název	Kategorie	P	D	R
1	Fluktuace pracovníků	Team	83	3	249
2	Chyby v analýze	Analýza	35	4	140
3	Zákazník začne vznášet nové požadavky	Zákazník	60	2	120
4	Neznalost vývojového prostředí	Vývoj	30	2	60
5	Chyby HW a SW	Technické	3	4	12

Typy rizik

- **známá rizika** – odhalí se vyhodnocením plánu, obchodního a technického prostředí
- **předvídatelná rizika** – lze je předpovědět na základě zkušení z předchozích projektů
- **nepředvídatelná rizika** – těžko předpovídatelná rizika s nízkou pravděpodobností

Identifikace rizik

- **obecná rizika**
 - **velikost produktu**
 - špatný odhad způsobí prodloužení vývoje, zohledňuje množství uživatelů a znovupoužitelného software, velikost databáze
 - **obchodní dopad**
 - Lze splnit termín, kolik to bude stát, pokud se nesplní
 - Cena za defektní produkt
 - Bude z této zakázky nějaký zisk?
 - Množství a kvalita dokumentace
 - **charakteristiky zákazníka**
 - Jedná se o nového zákazníka?
 - Ví zákazník, co vlastně chce?
 - Souhlasí zákazník s tím, že bude muset věnovat nějaký čas pro stanovení požadavků a rozsahu projektu?
 - Chce se zúčastnit sledování projektu?
 - Je zákazník technicky vzdálený v oblasti produktu?
 - **definice procesu**

- Podporuje vedení společnosti standardizaci procesu softwarového vývoje?
- Existuje použitelná metodika, byla někdy použita v jiném projektu, budou ji lidé používat?
- Poskytuje organizace kurzy pro manažery a vývojáře?
- Dochází k revizi specifikace požadavků, testovacích procedur a dokumentace?
- Jsou řízeny změny v požadavcích, návrhu, kódu a testovacích případech?
- Jsou používány specifické metody pro softwarovou analýzu, návrh struktury dat a architektury?
- Je definována konvence pro psaní kódu?
- Jsou použity specifické metody pro návrh testovacích dat?
- Používají se softwarové nástroje pro podporu plánování a řízení, pro řízení konfigurací a změn, pro podporu analýzy software a pro testování?
- Existuje standard pro tvorbu dokumentace?
- Sbírají se kvalitativní metriky a metriky produktivity?
- **vývojové prostředí**
 - Jsou k dispozici vhodné nástroje pro řízení, analýzu a vývoj?
 - Jsou s těmito nástroji všichni seznámeni?
 - Jsou k dispozici experti, schopní konzultovat používání nástroje?
 - Existuje vhodná dokumentace k nástrojům?
- **vytvářená technologie**
 - Je technologie v organizaci nová?
 - Požaduje zákazník tvorbu software založeném na neznámé technologii?
 - Bude mít produkt propojení s novým neověřeným hardware nebo softwarovým produktem?
 - Je vyžadováno speciální uživatelské rozhraní?
 - Je požadována tvorba programových komponent, které jsou odlišné od dříve vytvořených v dané organizaci?
 - Je vyžadováno použití nových metodu analýzy, návrhu nebo testování?
 - Je požadováno použití nekonvenčních metod vývoje sw?
- **velikost týmu a jeho zkušenosti**
 - Mají všichni členové týmu odpovídající zkušenosti?
 - Je jich dostatečný počet?
 - Budou na projektu pracovat na plný úvazek?
 - Rozumí všichni projektu?
 - Může v případě úbytku pracovní síly projekt pokračovat?
 - Jsou všichni pracovníci nahraditelní?
- **specifická rizika** (vázaná na vyvíjený produkt)

Nejčastější rizika a jejich důsledky

- Špatné zadání - software neplní svoji funkci
- Úbytek pracovní síly - prodloužení projektu
- Inflační množství požadavků
- Chybný časový plán
- Nízká produktivita práce

Rizikové komponenty

- **Rizika provedení** - stupeň nejistoty, že produkt bude odpovídat požadavkům a bude vyhovovat zamýšlenému použití
- **Rizika ceny** - stupeň nejistoty, že bude dodržen rozpočet
- **Rizika podpory** - stupeň nejistoty, že software půjde snadno opravovat, upravovat a zlepšovat
- **Rizika času** - stupeň nejistoty, že časový harmonogram bude dodržen a že produkt bude dodán včas

Kategorie dopadu

1. zanedbatelný
2. marginální
3. kritický
4. katastrofický

Tvorba tabulky rizik

1. Identifikace rizik: výběr rizik z výše uvedeného seznamu doplněný o kategorii (obchodní, velikost software ...)
2. Pro každé riziko se určí pravděpodobnost v procentech. Odhad je subjektivní, takže je lepší udělat ve více lidech a zprůměrovat.
3. Pro každé riziko se určí dopad (1-4)
4. Výsledné riziko je součin pravděpodobnosti a dopadu
5. Rizika se seskupí podle závažnosti a vedoucí projektu stanoví dělící čáru pro významná rizika
6. Pro významná rizika se sestaví plán **RMMRM**(Risk Mittigation, Monitoring and Management) - plán zmírnění, monitorování a řízení rizik

Plán RMMRM

Cílem plánu je odhadnout rizika, zajistit jejich monitorování, určit postupy pro zmírnění a sbírat informace pro příští analýzu.

obsahuje tyto části:

- Úvod
 - 1. Obsah a účel dokumentu
 - 2. Přehled hlavních rizik
 - 3. Odpovědnost za monitorování a řízení rizik
- Tabulka rizik projektu
 - 1. Popis všech rizik nad dělící čarou
 - 2. Faktory ovlivňující pravděpodobnost a dopad
- Zmírnění, monitorování a řízení rizik
 - zmírnění - obecná strategie a specifické kroky
 - monitorování - co všechno musí být monitováno a jak
- Management
 - Plán pro případ, že riziko nastane
- Harmonogram iterací plánu
- závěr

Opatření

Opatření, která obsahuje plán RMMRM lze rozdělit na tyto kategorie:

- **Předcházení** - opatření eliminuje riziko
- **Zmírnění** - opatření sníží dopad nebo pravděpodobnost rizika
- **Akceptace** - opatření se provede až v době, kdy riziko nastane

Konkrétně lze ke zmírnění rizik podniknout tyto kroky:

- **Havarijní plány** - co dělat, když riziko nastane
- **Alternativní strategie** - návrh řešení s nižším rizikem (např. namísto nového frameworku se vrátíme zpět k starému odzkoušenému)

- **Rezervy** - při plánování je nutné vytvořit rezervu času a peněz na nepředvídatelné události. Časová rezerva by měla být polovina celkového času.
- **Obstarávání** - některé činnosti mohou být provedeny externí firmou, která má větší zkušenosti
- **Pojištění** - Pojistné smlouvy mohou zmírnit některá rizika

Řešení výjimek

*Pane kolego - kolegové, Fakt je, že řešení vyjimek „kde se vzalo tu se vzalo“. Opravdu spise zapada do řešení vyjimek v Jave atd. Uznávám, že to je odpověd nic moc. Mannova
(Ona se na to ptát nebude a nikdo jiný SI2 prý nemá zkoušet, takže odpověď je tady z formálního důvodu a je sepsána doktorkou Mannovou)*

I při plánování SW procesu můžeme mít výjimky (tedy speciální situace či chyby).

Řešení výjimek při plánování procesu jsou situace, požadavky a jevy, které je nutno ošetřit speciálním způsobem.

Návrh, analýza a řešení výjimek pomůže instituci zorganizovat a směrovat plánovací procesy, významně zlepšit odhady a soustředit se na hlavní vlivy, které nejvíce působí na nedokonalý výsledek (chyby v produktu). Existují na to i CASE nástroje.

Řešení výjimek při hledání chyb je ovlivňováno pravidlem 80/20. Hledáme zdroje chyb, které se obvykle soustředí ve 20 % dodaných produktů, přičemž 80 % má chyb minimum. Postupujeme i tak, že vyhodnocujeme posloupnost byznys procesů a hledáme v nich kumulované příčiny chyb.

Plánování a správa změn, správa verzí, správa konfigurací

Softwarová konfigurace je množina vzájemně se ovlivňujících objektů (SCI - *Software Configuration Item*, může to být program, dokumentace nebo data), které jsou produktem některé SI aktivity.

Když je konfigurační objekt vytvořen a prošel oponenturou, stává se tzv. srovnávací základnou.

Řízení změn je procedurální aktivita, zajišťující kvalitu a konsistenci, při provádění změn na objektu. Konfigurační audit je SQA (*Software Quality Assurance*) aktivita, která pomáhá zajistit kvalitu provedené změny.

SCM (Software Configuration management)

Proces řízení konfigurací, má za cíl:

1. identifikovat změny
2. řídit změny
3. zajistit, aby změny byly řádně implementovány
4. informovat o změnách ty, kterých se to týká

SCM je aktivita, která začíná na začátku softwarového projektu a končí až, když je software vyřazený z provozu.

Musí odpovědět na otázky:

- Jak organizace identifikuje a řídí velké množství existujících verzí programů (včetně dokumentace) tak, aby bylo možné účinně provádět změny?
- Jak řídí organizace změny před a po dodání softwaru zákazníkovi?
- Kdo zodpovídá za schvalování a určování priorit změn?
- Jak je zabezpečeno, aby změny proběhly správně?
- Jak jsou ošetřeny vedlejší změny, které mohou nastat?

Aby proces SCM mohl na uvedené otázky odpovědět, musí provést některé základní úkoly, kterými jsou:

- identifikace SCI

- řízení verzí
- řízení změn
- audit konfigurací
- evidence a zprávy

Plánování a správa změn

Změny v SW projektu jsou často iniciovány změnami požadavků, se kterými ve velkém projektu musíme počítat vždy. Pro úspěšné řešení změn je potřeba:

- už ve fázi konzultace najít podrobné a co nejvíce vyhovující požadavky
- zamezit nepochopení požadavků realizačním týmem, nejúčinnější metoda: realizační tým je z požadavků důkladně překoušen
- zvolit iterativní metodiku a v každé její fázi konzultovat výstupy se zákazníky
- naplánovat opatření, stanovit odhadované ceny eventuálních změn a zákazníka o nich informovat

Proces změny můžeme popsat následujícími body:

- zjištěná potřeba změny
- požadavek na změnu od uživatele
- vyhodnocení vývojářem
- generování změnové zprávy
- odpovědná osoba nebo skupina - CCA (change control authority) rozhodne o změně (v případě neakceptace je informován uživatel a proces změny končí)
- požadavek je zařazen do pořadí k vyřízení, stanoví se kritéria pro revizi či audit - ECO (engineering change order)
- určí se pracovníci
- příslušný konfigurační objekt se vybere z databáze
- provedou se změny
- provede se revize (audit)
- konfigurační objekt se zařadí do databáze
- stanoví se srovnávací základna pro testování
- provedou se aktivity QA a testování
- změny se zahrnou do dalšího vydání (release)
- přepracuje se příslušná verze softwaru
- revize (audit) změn ve všech SCI
- zahrnout změny do nové verze
- distribuovat novou verzi

Přístup k databázi musí mít **synchronizační kontrolu**, aby dvě osoby nemohly provádět současně změny jednoho objektu.

Neformální řízení změny - předtím, než se SCI stane srovnávací základnou. Vývojář se dotáže, zda změny jsou v souladu s technickými a projektovými požadavky (pokud se nedotýkají širších systémových požadavků)

Řízení změny na projektové úrovni - když je SCI srovnávací základna. Vývojář musí dostat schválení k provedení změny od projektového manažera (je-li změna místní, nebo CCA, zasáhne-li další SCI).

Formální řízení změny (viz výše).- jakmile se produkt dodá zákazníkovi. CCA musí rozhodnout: jak změna ovlivní chování systému, jak se projeví z pohledu zákazníka, jak to ovlivní kvalitu a spolehlivost.

Správa verzí

Forma reprezentace verzí je evoluční graf, který zobrazuje hierarchii změn: drobné úpravy 1.1 → 1.1.1, větší změny 1.1 → 1.2, ještě větší 1.1 → 2.0

Každý uzel je agregovaný objekt, kompletní verze softwaru - sada SCI.

Verze - varianta - komponenta Každá verze může být složena z různých variant. Varianty se mohou lišit např. výběrem různých komponent (např. pro monochromní nebo barevný displej)

Každá varianta je složena s odlišného souboru objektů jedné úrovně revize a tudíž koexistuje paralelně s jinými variantami.

Komponenta je složena z objektů stejné úrovně revize.

Správa verzí pomocí Subversion

Subversion [<http://cs.wikipedia.org/wiki/Subversion>] patří do kategorie version control nástrojů a uspokojuje základní potřeby při správě verzí.

Předpokládejme, že je již v repository importovaný projekt, jedná se o zjednodušený proces, kdy požadavky jsou vyvíjeny za sebou, nikoliv paralelně.

1. Vyzvednutí projektu (tzv. checkout) z repository do lokálního adresáře. Tím se vytváří pracovní kopie, která funguje jako pracovní prostor.
2. Editace požadovaných souborů (přidání, mazání).
3. Odeslání změn do repository (tzv. commit). Změny jsou viditelné pro všechny uživatele repository. Spolu se změnami se zapisuje čas jejich poslání do repository, autor a textový komentář. Další vývojář může pokračovat v práci.
4. Další vývojář (pokud již má pracovní prostor) provede stažení aktuální verze z repository (tzv. update) a pokračuje ve vývoji. Vytvořené změny opět odešle do repository (tzv. commit).

Audit konfigurací

1. Formální revize je zaměřena na technickou správnost modifikovaného konfiguračního objektu, prověří konzistentnost k jiným SCI, úplnost, případné vedlejší efekty.
2. SC audit kontroluje, zda byly změny specifikované v ECO provedeny, zda nebyly provedeny ještě další změny, zda byly změny správně označeny v SCI, zda bylo zaznamenáno datum a autor změny. Zda byla změna evidována a oznámena. Zda byly opraveny všechny související SCI.

Evidence stavu (status reporting)

1. co se stalo
2. kdo to udělal
3. kdy se to stalo
4. co dalšího to ovlivnil

Standardy pro proces vývoje a implementace software

Lineární sekvenční model (model vodopád)

Je to klasický životní cyklus. V 80. letech byl podroben kritice, která dospěla k formulování základních nedostatků, kterými jsou:

- reálné projekty zřídka kdy sledují jednotlivé kroky v předepsaném pořadí
- pro zákazníka je obtížné vyjádřit předem všechny požadavky
- provozuschopná verze bude k disposici až po delší době (může být už zastaralá)
- často dochází k prodlevám

Přesto však i v současné době je pro řešení řady i velkých projektů model vodopádu používán.

Prototypování

Slouží k pochopení požadavků na systémy, které nejdou dobře specifikovat předem. Prototypy, na kterých mají být modelovány nějaké vlastnosti systému, mohou být dělány s vědomím, že budou zahrozeny. Tento model lze použít pro menší systémy.

RAD (Rapid Application Development) model

Je to lineární sekvenční model spočívající v extrémně krátkém vývojovém cyklu - do 3 měsíců. Model je určen pro dobře srozumitelné a dobře vymezené problémy, s malými riziky. Problém je rozdělen na samostatné moduly, které se rozdělí týmům. Při vývoji jsou využívány hotové softwarové komponenty.

Přírůstkový model (evoluční model)

Kombinuje lineární model s iterativní filosofií. Produkt se vytváří po částech (přírůstcích), které jsou funkční (na rozdíl od prototypu). První přírůstek je označován jako jádro. Model je vhodný pro malý tým a velký úkol, který se dá zvládnout v předem po částech, v domluvených termínech.

Sem se dá zařadit metodika UP (*Unified Process*).

Spirálový model (evoluční model)

Model kombinuje prototypování se systematickým sekvenčním přístupem a opakováním na výším stupni zvládnutí problematiky. Je založen na prioritě tvorby verze s vyšší rizikovostí. Části s větší mírou rizika jsou realizovány dříve. Nevýhodu modelu je, že nelze stanovit přesné termíny (při práci na zakázku), závisí na správnosti rizikové analýzy, náročné na zkušenosť pracovníků (je nutné více kontrolních bodů, po každé analýze rizik). Je vhodný pro velké systémy.

Model skládání komponent (evoluční model)

Spirálový model pro objektové technologie, který využívá skládání hotových komponent z knihovna tříd objektů.

Model souběžného vývoje (evoluční model)

Jednotlivé komponenty jsou vyvíjeny paralelně (vývoj je modelován jako paralelní procesy). Je vhodný např. pro klient/server aplikace.

Formální metody

Spočívají na formálních specifikacích, podporují formální verifikaci programů. Nevýhody, které zatím brání praktickému nasazení je, že je drahý a časově náročný, je náročný na kvalifikaci vývojářů, je v něm obtížná komunikace s běžným uživatelem.

Techniky čtvrté generace

Jsou založeny na programování na vysoké úrovni abstrakce. Výhodou je rychlý vývoj a zvýšení produktivity. Nevýhodou je, že některé prostředky jsou stejně složité jako klasické programovací jazyky, výsledný kód nemusí být dostatečně efektivní a údržba velkého systému může být problematická.

Sem se dá zařadit pojem MDA (Model Driven Architecture) - „nakresli a vygeneruj“.

Za perspektivní metodu vývoje softwaru se považuje spojení CASE nástroje s modelem skládání komponent.

Softwarové kontrakty, intelektuální vlastnictví

Softwarové kontrakty

Smlouva o dílo by měla mj. obsahovat:

- požadavky
- akceptační test
- specifikaci plnění, termíny a cenu

Intelektuální vlastnictví

Program nemůže být vynálezem, proto nemůže být chráněn patentem. Je chráněn **autorským právem (copyright)** před nespravedlivým využíváním. Autorské právo nechrání samotné myšlenky či ideje, pouze konkrétní díla, konkrétní vyjádření takových myšlenek, dílo v objektivně vnímatelné podobě.

Z důvodu ulehčení spolupráce a distribuce vznikly různé **licence**. Přehled známých softwarových licencí je zde [http://cs.wikipedia.org/wiki/Softwarov%C3%A9_licence]. V české legislativě platí od dubna 2009 **Creative Commons** licence, která umožnuje nastavit pomocí kombinací vlastností: *Attribution* (uvedení autora), *Noncommercial*, *No Derivative Works* a *Share Alike* různé varianty licence. Vlastnost *Share Alike* umožňuje ostatním rozšiřovat odvozená díla pouze za podmínek identické licence (viz též copyleft.)

Při vytvoření odvozeného díla z díla, jež je dostupné jen pod **copyleft** licencí, musí být toto odvozené dílo nabízeno pod stejnou (copyleft) licencí jako dílo původní. Princip copyleftu vytvořil Richard Stallman a uplatnil jej v copyleft licencích **GPL** a **GFDL GNU** projektu. Copyleft je pokládán za základ úspěchu hnutí svobodného software.

Jména programu a jeho logo ošetruje **ochranná známka**. Registrace ochranné známky je důležité strategické rozhodnutí (platí minimálně 10 let). Samotná ochranná známka úspěšného produktu nebo služby může mít velkou finanční hodnotu.

Přístupy k údržbě a dlouhodobému vývoji SW

- autor software je nejlepší správce běžícího projektu
- ovšem může zdržovat od vývoje celý team
- vždy používat **konfigurační soubory**, změna v konfiguráku je levnější, než změna v kódu (oddělení politiky od mechanismu)
- pokud je potřeba měnit kód, vždy je dobré provést **regresní testování**

Údržba software

- zavádí se jen minimálně nutné věci
- pro zadávání, dokumentaci a sledování chyb je vhodné použít Bugzillu nebo jiný bugtracker.
- při rozšíření aplikace se dodržuje existující architektura
- každou opravu je potřeba před vypuštěním do světa otestovat

¹⁾ 1)+doba trvání

²⁾ 2)+ doba trvání

Otázka 16 - Y36SI3

Zadání: Modelování a návrh flexibilních systémů na architektonické úrovni. Základy modelem řízené architektury. Architektonické styly a vzory. Komunikační prostředky a komponenty (middleware), aplikační prostředí (application frameworks). Konfigurace a správa konfigurací. Produkční linky.

slovnicek pojmu

- MDA - model driven architecture - modelem řízená architektura
- OMG - Object management group - skupina stojící za MDA
- RAD - Rapid Application Development - zjednodušeně řečeno metodiky vedoucí ke zrychlení vývoje aplikací zahrnující iterativní vývoj a konstrukce prototypů ⇒ nižší náklady ⇒ vyšší zisky
- EMS - Enterprise messaging system - takový hrozně chytrý systém nad XML, SOAP nebo Web services, který umožňuje posílat sémanticky přesné zprávy mezi počítačovými systémy

Modelování a návrh flexibilních systémů na architektonické úrovni

V tomto bode sa po nás chce vymenovať nejaké výhody modulárnych/vrstvených aplikácií a používania architektonických štýlov a vzorov.

Cielom softvérových architektúr je poskytnúť nástroje, ktoré nám umožnia písanie znovupoužitelný kód v ktorom sa jednoducho robia zmeny.

Výhody softvérovej architektúry

- pomoc pri plánovaní projektov - Ak má spoločnosť hotové riešenia niektorých služieb, umožňuje to lepší odhad harmonogramu, náročnosti prác a rýchlejší vývoj.
- produkty tretích strán - Komunikačné rozhrania jednotlivých komponent môžu poslúžiť pri integrácii s produktmi iných dodávateľov.
- znovupoužitelnosť - Softvérový architekt bude vedieť aké komponenty sa použili v posledných projektoch a ktoré z nich by išlo použiť v tom aktuálnom (šetrenie času a peňazí).
- ulahčenie testovania - Možnosť testovať jednotlivé komponenty systému nezávisle na sebe nám umožní rýchlejšie nájdenie chýb v systéme.
- rýchlejší vývoj - Ak sú dané rozhrania medzi komponentami, môže na projekte súčasne pracovať viac týmov.
- jednoduchá úprava kódu - Možnosť vykonať zmeny v systéme bez velkých zásahov do kódu (stačí zmeniť komponentu a netreba sa hrabáť v celom systéme).
- komunikácia so zákazníkom - Vrstvené a modulárne architektúry umožňujú kresliť pekné farebné obrázky a to je predsa vec ktorú chcú manažéri u zákazníka vidieť. :) Zároveň zákazník lepšie pochopí náš návrh (jednoduchší pohľad aj na zložité systémy) a to nám umožní predať mu nás produkt.

Základy modelem řízené architektury

Model driven architecture (dále jen MDA) je metodika návrhu aplikací vzniknulá v letech 2001, která si bere za cíl za pomocí několika jasně a přesně definovaných, **vzájemně transformovatelných** modelů přistupovat k návrhu aplikace.

Abych ale případné zájemce utišil ve volání „Sláva!“, je třeba podotknout, že navzdory existenci nástrojů, které jsou opravdu schopny vytvořit takový model, provést transformace a následně dokonce vygenerovat spustitelný kód, je to za cenu naprostých zbytečností a mnohé práce v modelu a výsledek bývá přinejmenším velmi diskutabilní po stránce kvalitativní. Je prostě mnohem jednodušší a v neposlední řadě i levnější použít několik mozků a tým biologických automatů pro tvorbu kódu, než jiný (a dražší) tým analytiků s praktickou znalostí MDA.

Proč si to tedy někdo vymyslel?

Dejme tomu, že nastane chaotické schvalování daňových zákonů na poslední chvíli a následná potřeba promítnout příslušné změny do softwarových aplikací v šíbeničních termínech (politici k „10.12. Y“ schválí novelu platnou od „1.1. Y+1“). To vede k potřebě něčeho, čemu se obecně říká RAD (Rapid Application Development). Takové metodiky mají ale tu nevýhodu, že vesměs snižují flexibilitu produkovaných aplikací (vemte si např. projekty Joomla nebo Magento, které sice ohneme, ale díky jejich návrhu jsou některé věci dosti krkolomné).

Snaha o vyšší flexibilitu byla a je také jedním z tlaků, které nutí vývojové týmy více využívat objektově orientované přístupy. A právě objektové přístupy vedou i k většimu využití UML (Unified Modeling Language) jako jazyka popisu systému.

Návrháři využívají UML celkem přirozeně – přesně totiž koresponduje se stylem, jakým je vytvářen objektový kód aplikace. Analytici zase v UML uvítali bohatší notaci pro přesnější popis koncepcí řešení. Zdálo by se tedy, že všichni jsou spokojeni a nikde není žádný problém. Bohužel tomu tak není.

Na problémy se obvykle narazí ve chvíli, kdy si analytici a návrháři (případně programátoři) mají své informace předat. Jejich modely mají sice stejnou notaci a syntaxi, občas se jim i podaří se na jednotlivém konkrétním elementu shodnout, ale celkově se jejich modely značně liší.

Konsorcium Object Management Group si je vědomo úskalí použití UML různými rolemi ve vývojových týmech a nabízí pohled na tvorbu softwarových systémů trochu z jiného úhlu, a to ve formě specifikace Model Driven Architecture.

Úrovně / Modely

MDA přistupuje k aplikaci v následujících rovinách:

- CIM – **Computation Independent Model** (model nezávislý na počítačovém zpracování)
- PIM – **Platform Independent Model** (platformově nezávislý model řešení)
- PSM – **Platform Specific Model** (platformově specifický model řešení)
- Code – **Code** (kód aplikace, tj. výsledná realizace řešení)

CIM – Computation Independent Model

Model nezávislý na počítačovém zpracování je de facto modelem vlastního „businessu“. Jedná se především o **model procesů**. Jeho notace však není v UML standardizována (ani v UML 2.0), nicméně se dá pro tyto účely „přehnout“ **diagram aktivit**. Dále je to **slovník pojmu** problémové oblasti, který se dá u složitějších oblastí vyjádřit pomocí **koncepcionálního modelu tříd**. Tento model vytvářejí buď sami uživateli nebo business analytici.

PIM – Platform Independent Model

Platformově nezávislý model reprezentuje koncepci řešení dané problémové oblasti na základě konkrétních požadavků. Jeho hodnota je především ve vyřešení koncepčních otázek, jako jsou třeba algoritmy zaskladňování a vyskladňování v případě skladů, nebo způsob párování saldokontních položek v účetnictví. Tento model však **neobsahuje informace spojené s konkrétní technologií realizace** a vytvářejí ho IT analytici.

PSM – Platform Specific Model

Model řešení na dané platformě (např. J2EE nebo C# a ASP.NET) je podkladem pro vlastní implementaci. Důležité je to, že **PSM má stejnou strukturu jako kód aplikace**. Tento model vytvářejí návrháři.

Code

Z hlediska MDA je zdrojový kód chápán také jako **model konkrétní realizace na dané platformě**. Konec konců určitě každý zná ze svého okolí aplikace, kde jedině tento model skutečně odpovídá realitě toho, co aplikace dělá.

Co je v MDA objevného?

Z dosud uvedeného by se mohlo zdát, že OMG „vymyslelo kolo“, neboť to, že je potřeba vytvářet analytický model (PIM) a návrhový model (PSM), víme již dávno. Nicméně síla MDA není ani tak v členění modelů, jako v tom, že **definuje způsob a postup transformace modelů**. Opět se nejedná o žádnou magii, ale o aplikaci osvědčených praktik, především zkušeností z použití návrhových vzorů (Design Patterns). Nejzajímavější je otázka transformace platformově nezávislého modelu (PIM) do platformově specifického modelu (PSM).

Postup transformace dle MDA

1. PIM model je doplněn mapovacími značkami, které definují, jaká obecná transformační pravidla budou použita.
2. Pro PIM model (resp. jeho části) je zvolena implementační platforma.
3. Na základě mapovacích značek jsou provedeny odpovídající transformace již s ohledem na zvolenou platformu.

MDA mapovací značkou je obecně definováno, jakým způsobem bude realizována asociace, kterou vytvořil analytik. Jedná se v podstatě o přiřazení obecnějšího návrhového vzoru příslušnému modelovému

elementu. Konkrétní způsob realizace pak může být pro různé platformy jiný, takže z jednoho „značkovaného“ modelu PIM je možné odvodit různé PSM modely.

Při tvorbě PSM modelu pro zvolenou platformu je pak rozvinut aplikovaný návrhový vzor, čímž dojde k doplnění příslušných implementačních atributů, operací nebo i celých tříd a tím vznikne základ platformově specifického modelu. Důležité je to, že přidané implementační prvky jsou označeny jako modelově závislé (MDA doporučuje používat značku řecké pí) a ve výsledném PSM modelu je tak dobře vidět, co jsou prvky převzaté z analýzy a které prvky byly přidány až v návrhu.

Přínosy nasazení

Přínos koncepce Model Driven Architecture je tedy především v tom, že jasně definuje, co je analytický model, co je návrhový model a jak provádět jejich transformaci. Cílem je, aby aplikace byla popsána na různých úrovních abstrakce a tím pádem je značně usnadněno konzistentní promítání změn v aplikaci. Další pozitivní důsledek je standardizace návrhu, která umožňuje zvýšit kvalitu aplikací. A co je důsledkem toho všeho? Především snížení nákladů na údržbu aplikací, tedy peníze, o které jde jako obvykle až v první řadě.

Architektonické styly a vzory

Architektúra softvéru sa zaobráva popisom komponent na tvorbu systémov, interakciou medzi nimi, architektonickými vzormi a obmedzeniami týchto vzorov. Komponenta môže byť napríklad vrstva, filter; spojenie medzi komponentami je napríklad volanie funkcie. Spoločným rysom architektúr je snaha znižovať zložitosť pomocou abstrakcie a rozdelenia systému na logické časti.

Najpoužívanejšie štýly/vzory

Layers - Komponenty systému tvoria vrstvy. Každá vrstva poskytuje funkčnosť nadadeným vrstvám a zároveň využíva funkcie hierarchicky nižších vrstiev. Komunikácia medzi vrstvami môže mať rôzny stupeň volnosti - v striktnej verzii môže vrstva komunikovať len s priamo pod- a nadadenou vrstvou (napr. OSI model). Nižšie vrstvy riešia úlohy blízke hardvéru a vyššie vrstvy zaobstarávajú interakciu s užívateľom. Výhodou vzoru je rozdelenie zložitého problému na menšie časti, nevýhodou je nemožnosť rozdelenia každého problému do strikných vrstiev.

Model-View-Controller - Architektonický vzor, ktorý rozdeluje dátový model, uživatelské rozhranie a riadiacu logiku do troch komponent.

- Model je špecifická reprezentácia informácií s ktorými aplikácia pracuje-
- View prezentuje dátá ktoré poskytuje model uživatelovi.
- Controller reaguje na udalosti (typicky zo strany užívateľa) a zaistuje zmeny v modeli a pohlade.

Ako to funguje: Užívateľ vykoná akciu na ktorú zareaguje Controller tak, že pristúpi k Modelu a aktualizuje ho na základe vykonanej akcie (napr. aktualizácia obsahu nákupného košíku). Model spracuje zmeny (napr. prepočíta celkovú cenu nákupu) a prípadne ich uloží do databáze. View použije aktualizovaný model pre zobrazenie nových dát uživateli (napr. výpis obsahu košíku). Je dôležité si všimnúť, že v komunikácii medzi Modelom a View nefiguruje Controller. (Controller ale môže zaktivovať View aby svoj obsah aktualizoval podľa nového Modelu).

<http://cs.wikipedia.org/wiki/Model-view-controller> [<http://cs.wikipedia.org/wiki/Model-view-controller>]

Klient-server - Popisuje vzťah medzi dvoma programami - klientom ktorý posielá požiadavky a serverom, ktorý tieto požiadavky spracováva. Túto architektúru typicky využívajú aplikačné protokoly na sieti (e-mail, DHCP, HTTP, SSH alebo aj databáze). Medzi výhody patrí určitá nezávislosť klientskej a serverovej časti, uchovávanie dát na serveri (možnosť riadenia prístupu, správa dát na jednom centrálnom mieste). Ako nevýhodu možno spomenúť možnosť záhltenia serveru či nízku robustnosť - pád serveru (ak je len jeden) znamená nefunkčnosť aplikácie.

<http://en.wikipedia.org/wiki/Client-server> [<http://en.wikipedia.org/wiki/Client-server>]

Three-tier model - Jedná sa o typ klient-server aplikácie, v ktorej sú logicky rozdelené prezentačná, aplikačná a dátová vrstva. Do tejto kategórie spadajú aplikácie používajúce medzi uživatelia a databázou middleware na spracovanie požiadavkov na dátu.

- Prezentačná vrstva zobrazuje dátu uživatelovi (zoznam zboží, obsah nákupného košíku).
- Aplikačná vrstva má na starosť samotnú funkcionality aplikácie, spracovanie dát.
- Dátovú vrstvu tvorí databáza ktorá uchováva všetky informácie. Zaistuje nezávislosť dát na aplikačnej vrstve a obchodnej logike (business logic).

Trojvrstvový model sa môže na prvý pohľad zdať ako iné pomenovanie návrhového vzoru Model-View-Controller, no nie je to tak. Trojvrstvový model je striktne lineárny - prezentačná vrstva nikdy nekomunikuje priamo s dátovou vrstvou - všetka komunikácia ide cez middleware. MVC je trojuholníkový model - View posielá nové dátá Controlleru, Controller updatuje Model a View je updatovaný priamo z Modelu.

http://en.wikipedia.org/wiki/Three-tier_%28computing%29

[http://en.wikipedia.org/wiki/Three-tier_%28computing%29]

– tohle je blbost, to si fakt nepamatujte...zasadni rozdiel ej v to, ze MVC jsou 3 logické vrstvy (proste 3 package trid, kde tridy delaj obdobnou cinnosť na stejny urovni abstrakce), 3-tier jsou 3 fyzicky vrstvy - tenkej klient (browser), aplikacni server (tomcat, apache, glassfish...) a databazovej server (postgre, mysql, oracle...) – mickap1 (nejak to uniklo pri kontrole)

Front-end a back-end - Front-end je časť aplikácie ktorá zaistuje interakciu s užívateľom. Back-end spracováva vstup z front-endu a posielá svoj výstup späť na front-end. Týmto je zaistená abstrakcia oddielujúca rozdielne časti systému. Túto architektúru využívajú aj grafické front-endy postavené nad terminálovými aplikáciami.

http://en.wikipedia.org/wiki/Front-end_and_back-end [http://en.wikipedia.org/wiki/Front-end_and_back-end]

Pipes and filters - Pri predstavení si tohto vzoru nám pomôžu znalosti z predmetu Y36UOS :) Filtre (jednotlivé komponenty) sú spojené rúrami, výstup jedného filtru sa posielá cez rúru na vstup druhého filtra. Jednotlivé filtre sú na sebe nezávislé entity, ktoré nemajú žiadnu informáciu o tom, kto im posielá vstup a komu posielajú svoj výstup. Výhodou tohto riešenia je, že filter môže byť kedykolvek nahradený vylepšenou verziou ktorá má rovnaké chovanie (napr. vstup a výstup zostane rovnaký, ale zrýchli sa spracovanie dát). Existujú aj objektové filtre, ktoré si cez rúry posielajú objekty namiesto plain-textu.

http://en.wikipedia.org/wiki/Pipes_and_filters [http://en.wikipedia.org/wiki/Pipes_and_filters]

Inversion of Control - Tradičný tok riadenia (control flow) je, keď uživatelské funkcie volajú funkcie knihovní, funkcie knihoviek však nikdy nevolajú uživatelské funkcie. Inverzia riadenia nastane, ak funkcia z knihovny zavolá uživatelské funkcie. Lepšie si to vysvetlíme na príklade:

Predstavme si program, ktorý načíta uživatelovo meno a pozdraví ho. Program z konzole by mohol

vypadať nasledovne:

```
Zadaj svoje meno: Bob
Vitaj, Bob!
```

Pseudokód:

```
funkcia pozdrav(meno){
    vypis("Vitaj, " + meno + "!");
}

meno = citaj_vstup();
pozdrav(meno);
koniec();
```

Obdobný program v GUI by mal textové pole na zadanie mena a tlačítko. Keď užívateľ zadá meno a stlačí tlačítko, program mu zobrazí pozdrav.

Pseudokód:

```
funkcia pozdrav(meno){
    zobraz_okno("Vitaj, " + meno + "!");
}

tlacitko_akcia(pozdrav, argument=vyber_obsah_textoveho_pola());
```

Rozdiel v programoch je, ako sa vykonajú:

- V prvom prípade určuje programátor, kedy sa majú funkcie volať.
- V druhom prípade programátor predáva riadenie GUI. GUI rozhoduje, kedy sa majú funkcie zavolať, programátor len určí čo sa má vykonať.

<http://martinfowler.com/bliki/InversionOfControl.html> [<http://martinfowler.com/bliki/InversionOfControl.html>]

Dependency injection - Objekt A závisí na objekte B, inými slovami objekt A obsahuje odkaz na objekt B. Pri použití Dependency Injection budú pri štarte kontajneru (ktorý tieto objekty spravuje), oba objekty vytvorené tak, že v objekte A bude inicializovaný odkaz na objekt B. Existujú tri základné spôsoby ako toto dosiahnuť:

- Setter Injection

```
public class Apple {

    private Banana banana;

    public void setBanana(Banana b) {
        this.banana = b;
    }
}
```

```
<bean id="apple" class="Apple">
    <property name="banana" ref="Banana"/>
</bean>

<bean id="banana" class="Banana"/>
```

- Constructor Injection

```
public class Apple {
    public Banana banana;
    public Apple(Banana b) {
        this.banana = b;
    }
}

<bean id="apple" class="Apple">
    <constructor-arg ref="banana"/>
</bean>

<bean id="banana" class="Banana"/>
```

- Interface Injection

```
public interface InjectFinder {
    void injectFinder(MovieFinder finder);
}

class MovieLister implements InjectFinder {
    public void injectFinder(MovieFinder finder) {
        this.finder = finder;
    }
}
```

<http://blog.springsource.com/2007/07/11/setter-injection-versus-constructor-injection-and-the-use-of-required/> [<http://blog.springsource.com/2007/07/11/setter-injection-versus-constructor-injection-and-the-use-of-required/>]
<http://www.martinfowler.com/articles/injection.html> [<http://www.martinfowler.com/articles/injection.html>]

Event-driven architektúra - Vzor založený na tvorbe, detekcii a reakcií na udalosti. Takýto systém pozostáva z tvorcov udalostí (agent) a ich konzumentov (sink). Konzumenti majú na starosť reagovať na udalosť hneď po jej detekcii (zavolaním nejakej funkcie alebo preposlaním udalosti). Event-driven architektúra pozostáva zo štyroch logických vrstiev:

- generátor udalosti - Zistí fakt z ktorého vytvorí udalosť. Fakt môže byť čokolvek čo možno nejakým spôsobom detektovať (zvýšenie teploty na senzore, prijatie mailu...).
- kanál na prenos udalosti - Mechanizmus pomocou ktorého sa dostane udalosť od tvorca ku konzumentovi (TCP/IP spojenie, vstupný súbor...).
- engine na spracovanie udalosti - Po identifikovaní udalosti vyberie vhodnú reakciu a vykoná ju. Táto činnosť môže vyvolať ďalšie udalosti.
- downstream aktivita - Zobrazenie výsledku udalosti (hláškou na obrazovke, poslaním mailu...). Táto činnosť nie je povinná.

http://en.wikipedia.org/wiki/Event_Driven_Architecture

[http://en.wikipedia.org/wiki/Event_Driven_Architecture]

Implicitné volanie - Komponenta namiesto toho, aby priamo zavolala nejakú funkciu ohlási udalosť. Ostatné komponenty môžu prejaviť záujem o udalosť spojením tejto udalosti s funkciou. Vysielajúca komponenta teda nevie a nemôže ovplyvniť, ktoré ďalšie komponenty budú reagovať na vyvolanú udalosť.

Systém na udalosť reaguje tak, že zavolá funkcie, ktoré majú túto udalosť zaregistrovanú.
http://en.wikipedia.org/wiki/Implicit_invocation [http://en.wikipedia.org/wiki/Implicit_invocation]

Monolitická aplikácia - Všetká funkčnosť aplikácie (uživatelské rozhranie, spracovanie a prístup k dátam) je zakomponovaná do jedného programu.

http://en.wikipedia.org/wiki/Monolithic_application [http://en.wikipedia.org/wiki/Monolithic_application]

Plugin - Hostitelská aplikácia ponúka cez svoje API obmedzenú sadu funkcií a služieb ktoré sú dostupné cez interface. Toto umožňuje vývojárom tretích strán pridať do aplikácie funkčnosť ktorá rozšíri jej možnosti.

<http://en.wikipedia.org/wiki/Plugin> [<http://en.wikipedia.org/wiki/Plugin>]

Service-oriented architektúra - Aplikácia sa skladá z nezávislých blokov (služieb) ktoré sú previazané volnou väzbou (loose coupling). Služba je komponenta, ktorá má presne definované rozhranie a toto rozhranie určuje funkcionality, ktorú poskytuje. Rozhranie služieb je popísané pomocou štandardizovaného rozhrania WSDL a komunikácia medzi nimi prebieha pomocou komunikačného protokolu SOAP. WSDL a SOAP sú základnými špecifikáciami webových služieb - Web Services. Proces komunikácie zahrňuje posielanie jednoduchých dát medzi komponentami a umožňuje tým vznik akejsi komplexnejšej aplikácie.

http://en.wikipedia.org/wiki/Service-oriented_architecture

[http://en.wikipedia.org/wiki/Service-oriented_architecture]

<http://www.osu.cz/katedry/kip/aktuality/sbornik99/honzik.html> [<http://www.osu.cz/katedry/kip/aktuality/sbornik99/honzik.html>]
http://en.wikipedia.org/wiki/Software_architecture [http://en.wikipedia.org/wiki/Software_architecture]
prednášky predmetu Y36ASS

Middleware

Řečeno velmi prostě je middleware software, který zprostředkovává komunikaci (tj. výměnu dat) mezi dvěma a více aplikacemi napříč operačními systémy (a třeba i po síti).

Implementace bývá v dnešní době postavená především na technologiích XML, SOAP a Web services, nicméně nikdo neříká, že middleware musí nutně být nějaká separátní aplikace, jak si jí asi představíme. On to totiž je třeba i TCP/IP stack jako middleware pro síťovou komunikaci a tohle dnes umí snad každý operační systém.

Typy

Hurwitzové systém [<http://www.dbmsmag.com/9801d04.html>] rozděluje middleware na základě škálovatelnosti, stability a „zotavitelnosti“ na:

- **Remote Procedure Call** — Klient volá procedury běžící na vzdáleném stroji. Může být synchronní či asynchronní. Jen podotýkám, že implementace tohoto principu v Javě se jmenuje RMI (Remote Method Invocation, materiály [<http://nb.vse.cz/~zelenyj/it380/eseje/xnovm44/rmi.htm>]).
- **Message Oriented Middleware** — Zprávy zaslané aplikací #1 jsou zaslány na „kanál“ a uloženy

do doby, než na ně zareaguje aplikace #2. Aplikace samozřejmě pokračují ve své obvyklé činnosti dál a vzájemně na sebe nečekají.

- **Object Request Broker** — Tento typ dovoluje aplikacím posílat objekty a požadovat služby v objektově orientovaném systému.
- **SQL-oriented Data Access** — middleware mezi aplikacemi a databázovými servery.
- **Embedded Middleware** — komunikační služby a interface software/firmware, který pracuje mezi embedded aplikací a operačním systémem.

Dále se pak počítá ještě s těmito typy:

- **Transaction processing monitors** — Poskytuje nástroje a prostředí pro vývoj a nasazení distribuovaných aplikací.
- **Application servers** — software umožňující poskytování (běh) jiných aplikací (viz. JEE aplikační servery).
- **Enterprise Service Bus** — Abstraktní vrstva nad Enterprise Messaging System (EMS).

K Messagingu, jakožto asi nejrozšířenějšího middleware:

- Hlavní myšlenky: asynchronní přístup, „Store and forward“ a „Send and forget“, kanál = sdílený prostor mezi aplikacemi
 - **Store and forward** - - - - - Messaging systém
uloží zprávu (na počátku vysílače) a snaží se ji vyslat příjemci.
 - **Send and forget** - nečeká se na okamžík potvrzení
- Životní cyklus zprávy
 1. Create (vytvořit) - vytvořit zprávu a naplnit ji daty
 2. Send (poslat) – přidat zprávu do kanálu
 3. Deliver (doručit) - přesun zprávy ze zdroje do cíle – to zajistí MsS
 4. Receive (přijmout) - příjemce přečte zprávu z kanálu
 5. Process (zpracovat) – příjemce rozbalí data ze zprávy

Pochopitelně jsou s tím i problémy jako např. jak zajistit, aby zpráva fakt dorazila na kanál popř. jak to udělat, aby nevadilo, kdy tam dorazí (že to nebude zrovna včas apod.).

Application frameworks

Abychom mohli vůbec začít hovořit o **aplikačních frameworcích**, měli bychom vědět něco málo o **softwarových frameworcích**.

Software frameworks

Zkráceně vzato pojmem **sw. framework** rozumíme sadu knihoven, která si klade za cíl:

- IoC - v tomto případě application flow přebírá framework (odkazují na otázky zabývající se IoC).
Proč tomu tak je? Protože nechceme do projektu tahat každého přístup k pojednání aplikacní logiky.
- Poskytovat knihovny - ...ale nebýt jen jejich množinou. Framework mnohdy knihovny poskytuje či

dovoluje integrovat, nicméně není to jeho primární účel

- Dosazovat štábní kulturu - protože chceme v projektu jednotnou kulturu názvů a logiku členění kódů do celků

Příklady

Softwarové frameworky obsahují obvykle značné množství relativně obecně použitelných komponent, nicméně bývá zvykem, že se se specializují na některý segment aplikací:

- Umělecké kresby, komponování hudby, CADy
- Aplikace pro finanční modelování
- Decision support systémy
- Media playback a nahrávání
- Webové aplikace
- Middleware

Co to teda je?

Pojmem „Application framework“ rozumíme „Software framework“ používaný k implementaci standarních komponent/struktur aplikace pro specifický operační systém (resp. aplikační prostředí, kterým může být např. aplikační server).

Běžné použití takových frameworků najdeme v návhu GUI, které dalo a dává aplikacím jednotnou strukturu. Příkladem budiž Cocoa, Qt, GTK+, MFC nebo Swing. Nejen, že vývoj takové aplikace je mnohem rychlejší, neboť není třeba si každou GUI komponentu tvořit od počátku (zní to šíleně něco takového dělat, že?), ale pomůže to i ve vnitřní struktuře programu (jeden příklad za všechny: .NET naklikávátko GUI).

Abychom ale nebyli uvedeni v omyl, že application frameworky použijeme jen a výhradně při tvorbě GUI, pak připomínám, že se na app. framework díváme z pohledu prostředí, v němž běží. Tím pádem máme i webové aplikační frameworky (Spring, Zend Framework a asi 2000 dalších).

Stoklasa: *Myšlenkou aplikačního frameworku je skládání aplikace z hotových komponent a usnadnění tvorby opakování použitelných komponent. Kromě aplikačních frameworků pro konkrétní oblast (tvorba GUI, webové frameworky pro PHP apod.) existují i ambicioznější frameworky které se snaží být nějakým způsobem univerzální, jako například **Spring**. Možná by se pod tuto definici vešel i soubor knihoven okolo Microsoft .NET, je docela rozsáhlý a snaží se taky být univerzální. (Microsoft název .NET Framework používá pro všechno včetně běhového prostředí CLR - obdobu JVM - ale v rámci naší definice je Framework jen ta sada objektových knihoven.*

Konfigurace a správa konfigurací

Čo je konfigurácia?

Slovo konfigurácia obecne znamená zoskupenie vecí ktoré tvoria jeden celok. V počítačovej terminológii má slovo dva významy:

- **zoznam HW a SW** ktorý tvorí počítač (napr. niekto môže povedať: „*Konfigurácia počítača je Pentium4 3 GHz, 1 GB RAM, 160 GB pevný disk, Windows XP...*“);
- proces **výberu nastavení** pri inštalácii SW alebo HW, respektívne samotné **nastavenie** SW či HW.

Správa konfigurácií

Správa konfigurácií je systematické vyhodnocovanie, schvalovanie, zamietanie, a implementácia schválených zmien v konfigurácii systému. Táto činnosť zahrňuje aj zaznamenávanie a aktualizáciu informácií ktoré popisujú počítačový systém (vrátane všetkých HW a SW komponent) počas jeho životného cyklu. Tieto informácie obsahujú verzie, updaty a nastavenia nainštalovaného SW, verzie nainštalovaného firmwaru na zariadeniach, konfiguráciu sieťových prvkov, zoznam HW ktorý je v počítači/serveri a pod. Dôvodov pre ukladanie týchto informácií je viacero:

- dokumentácia konfigurácie systému počas jeho životného cyklu, kontrola modifikácií,
- kontrola definovaných požiadavkov (splňuje konfigurácia systému to, čo zákazník požadoval?),
- kontrola komponent systému,
- možnosť učiniť rýchlejšie rozhodnutie o zmene v systéme na základe aktuálnej konfigurácie tak, aby sa neovplyvnila funkčnosť systému.

Konfiguračná položka - Lubovolná časť konfigurácie ktorá musí byť samostatne uchovaná, testovaná, dodávaná a udržovaná (napr. kus HW, knihovna, súbor...). U každej položky sa uchováva názov, verzia, model, autor/výrobca, uživatelia, umiestnenie.

Medzi najčastejšie príčiny ktoré môžu viesť k zmenám v konfigurácii patria:

- nové podmienky (obchodné, tržné, zmena v zákonoch) na trhu ktoré vyvolajú zmenu v požiadavkoch na systém,
- zmena v potrebách zákazníka,
- obmedzenie rozpočtu alebo harmonogramu.

Baseline (porovnávacia základňa) je produkt, ktorý bol revidovaný tak, že ďalej slúži ako základ pre ďalší vývoj. Jedná sa teda o akýsi milestone vo vývoji.

Činnosti ktoré správa konfigurácií zahrňuje

- zistenie potreby zmeny,
- výber konfiguračných položiek podla špecifikácie,
- vyhodnotenie navrhnutej konfigurácie, jej schválenie či zamietnutie,
- realizácia zmeny,
- vykonanie auditu,
- testovanie novej konfigurácie,
- zahnutie zmien do ďalšieho vydania.

Audit konfigurácie je proces, počas ktorého sa overí, že vykonané zmeny odpovedajú špecifikovaným požiadavkom, že dokumentácia popisuje nové konfifuračné položky a že všetky požiadavky na zmenu boli úspešne vyriešené. Jedná sa teda o kontrolu kvality produktu po zmene v konfigurácii.

http://stm-wiki.cz/index.php/Soubor:Y36SI2_skripta.pdf [http://stm-wiki.cz/index.php/Soubor:Y36SI2_skripta.pdf]

(Kapitola 10. Řízení změn)

Diplomka Systém podpory řízení konfigurace [<http://www.rydval.cz/phprs/view.php?cisloclanku=2005061303>]
http://en.wikipedia.org/wiki/Configuration_management [http://en.wikipedia.org/wiki/Configuration_management]
<http://www.informit.com/articles/article.aspx?p=31451> [<http://www.informit.com/articles/article.aspx?p=31451>]

Správa zdrojového kódu (SCM, Source Code Management)

Správa zdrojového kódu (verzovanie, Revision Control) je spôsob uchovávania história všetkých zmien obecne u akéhokoľvek súboru. Najčastejšie sa verzuje zdrojový kód, konfiguračné súbory a dokumentácia. Jednotlivé verzie súboru sa označujú číslom zvaným „verzia“ alebo „revízia“. Pri každej verzii sa uchováva dátum zmeny a kto túto zmenu vykonal. Verzie môžu byť porovnávané, obnovené (restore) alebo zlúčené (merge). Dôvodom vzniku Version Control Systémov (VCS) bola potreba možnosti vrátiť kód do stavu keď bol ešte funkčný. Velká väčšina VCS neuchováva celé zmenené súbory, ale len rozdiely medzi verziami.

Tradičné (centralizované) VCS používajú centralizovaný model, v ktorom sa verzovanie súborov vykonáva na centrálnom serveri, ktorý je zdielaný. Preto je potreba zaistiť, aby dvaja ľudia editujúci ten istý súbor v rovnakom čase neprepísali svoje zmeny. Existujú dva prístupy ako to urobiť:

- uzamknutie súboru - Po tom, čo vývojár urobí „check out“ nad súborom, súbor sa uzamkne (ostatným bude prístupný len na čítanie) až kým vývojár „necommitne“ novú verziu.
- zlúčenie (merge) verzí - Umožňuje pracovať viacerým ľuďom nad rovnakým súborom. Prvý commit bude vždy úspešný. Ďalšie commity môžu byť konfliktné, preto si vývojár musí dať pozor, aby jeho commit nezaniesol do kódu chybu.

Príkladom centralizovaných VCS je CVS a Subversion (<http://coding-time.blogspot.com/2008/04/subversion-visually-explained-in-30sec.html>) [<http://coding-time.blogspot.com/2008/04/subversion-visually-explained-in-30sec.html>]).

Distribuované VSC nemajú centrálny server ktorý by uchovával referenčnú verziu kódu, existujú iba pracovné kópie (každý vývojár má vlastný repozitár). Synchronizácia je zariadená výmenou patchov medzi pracovnými kópiami. Príkladom distribuovaných VCS je git a Mercurial.

<http://rg03.wordpress.com/2007/06/10/distributed-versus-centralized-version-control-systems/> [<http://rg03.wordpress.com/2007/06/10/distributed-versus-centralized-version-control-systems/>]

Dôležité pojmy

- **baseline** - Schválená verzia súboru ktorá slúži ako základ pre ďalší vývoj.
- **branch** - Vývojová vetva, kópia súborov v repozitári ktoré je potreba z nejakého dôvodu (vyskúšanie iného postupu, testovanie novej funkcionality...) vyvíjať nezávislo na hlavnej vetve.
- **change, diff** - Zmena v súbore oproti predchádzajúcej verzii.
- **change list** - Zoznam zmien vykonaných vrámci jedného commitu.
- **check out** - Vytvorenie lokálnej pracovnej kópie z repozitára.
- **commit** - Zápis zmien vykonaných v pracovnej kópii do spoločného repozitára.
- **conflict** - Konflikt nastane, ak dvaja ľudia urobia zmeny v tom istom súbore a VCS tieto zmeny nedokáže zlúčiť. Konflikt musí vyriešiť uživatel buď kombináciou oboch zmien alebo uprednostnením jednej zmeny pred druhou.

- **import** - Zkopírovanie lokálnych adresárov ktoré nie sú súčasťou pracovnej kópie do repozitára.
- **merge** - Operácia počas ktorej sú dve zmeny aplikované na jeden súbor. Táto operácia nastane ak:
 - dvaja ľudia ukladajú v rovnakom čase do repozitára novú verziu toho istého súboru,
 - je potreba zlúčiť branch s iným branchom alebo s hlavnou vývojovou vetvou.
- **repository** - Miesto kde sú uchovávané súčasné aj staršie verzie súborov.
- **resolve** - Činnosť počas ktorej rieši uživateľ konflikt.
- **revision, version** - Stav súboru v určitom čase v minulosti.
- **update** - Stiahnutie nových verzí z repozitára do pracovnej kópie.
- **working copy** - Pracovná kópia je lokálna kópia súborov z repozitára. Všetky zmeny sa robia v pracovnej kópii a potom sa commitnú do repozitára.

http://en.wikipedia.org/wiki/Revision_control [http://en.wikipedia.org/wiki/Revision_control]

Produkční linky

Termín (softvérová) produkčná linka označuje postupy, nástroje a techniky softvérového inžinierstva umožňujúce tvorbu softvérových produktov, ktoré zdieľajú podobnú funkčnosť a možno ich vytvoriť rovnakými prostriedkami. Nápad tvorby softvéru z už existujúcich častí kódu pochádza z odvetví priemyslu (napr. výrobné linky na autá). Znovupoužitelné časti kódu nie sú vytvárané náhodne, ale sú písané ako časti dobre premyslenej rady softvérových produktov.

Výhody produkčných liniek

- rádovo rýchlejšie dodanie produktu na trh,
- zvýšenie produktivity aj kvality,
- zníženie nákladov,
- jednoduchšia údržba kódu,
- hromadné a rýchle prispôsobenie potrebám zákazníkov, škálovateľnosť produktu.

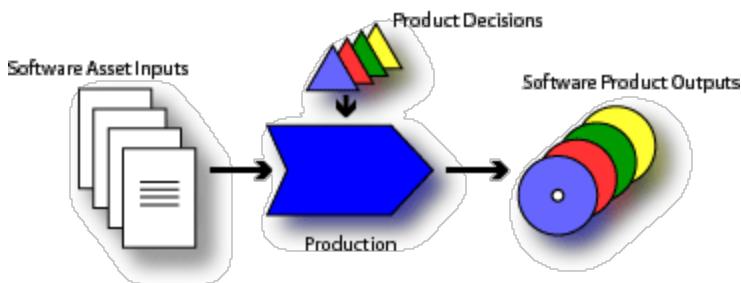
Charakteristika produkčných liniek

Software asset inputs - Množina kladov a prínosov softvéru, ktoré môžu byť rozdielne zostavené tak, aby vytvorili produktovú radu softvéru. Aby boli produkty vrámcí rady dostatočne odlišné, kladné vlastnosti sa nepridávajú do každého produktu alebo umožňujú rozdielnú konfiguráciu.

Rozhodovací model - Popisuje voliteľné a odlišné vlastnosti produktov vrámcí rady (každý produkt ich má jednoznačne určené).

Produkčný mechanizmus - Prostriedky umožňujúce zostavenie a konfiguráciu softvéru zo *software asset inputs* na základe rozhodovacieho modelu.

Výstupné softvérové produkty - Kolekcia softvéru jednej produktovej rady (produkčnej linky). Rozsah rady je určený rozhodovacím modelom.



Obrázok prevzatý z <http://www.softwareproductlines.com/introduction/concepts.html> [<http://www.softwareproductlines.com/introduction/concepts.html>]

Software production line vs Software factory

„V Software product lines i Software factories jede o to sekat software „jak Baťa cvičky“, ale Software factories na to jdou skládáním z hotových komponent a Software product lines na to jdou využením jenom těch komponent o kterých se ví že se opakovaně použijí v dalších projektech. Takže jemný rozdíl tam je...“ (Stoklasa)

<http://www.sei.cmu.edu/productlines/> [<http://www.sei.cmu.edu/productlines/>]

<http://www.softwareproductlines.com/introduction/introduction.html> [<http://www.softwareproductlines.com/introduction/introduction.html>]

http://en.wikipedia.org/wiki/Software_product_lines [http://en.wikipedia.org/wiki/Software_product_lines]

Otázka 17 - Y36SI3

Zadání: Návrhové vzory, knihovny vzorů a architektury. Přehled současných komponentových komunikačních prostředí. Návrh distribuovaných systémů. Komponentový návrh.

slovnicek pojmu

* **Distribuovaný systém**

- systém, který je nasazen na více počítačích (či jiném HW)

* **Socket**

- prostřednictvím socketů probíhá síťová komunikace
- socket obsadí určitý port na některé z adres počítače

* **RMI**

- Remote Method Invocation
- umožňuje objektu z jednoho Javoveho Virtualního Stroje (JVM) vyvolávat metody na jiném objektu, který se může nacházet v jiném JVM

* **Komponenta**

- Modulární část systému, která se chová jako blackbox. (ukrývá svůj obsah). Má definované rozhraní, pomocí kterého poskytuje služby v sobe zapouzdření.

* **SOA**

- Service Oriented Architecture - architektura orientovaná na služby
- Architektura orientovaná na služby je tvořena množinou komponent, které mohou být volány a jejichž popis rozhraní může být přístupný přes nějaké veřejné rozhraní ostatním aplikacím. (W3C)
- SOA není vázaná na konkrétní technologii a může být implementována s použitím velkého množství různých standardních technologií.
- V dnešní době je SOA úzce spjata s webovými službami. Neznamená to však, že sebez nich SOA neobejde.
- Jádrem SOA v prostředí internetu jsou tři části – poskytovatel (tj. entita poskytující své služby), odběratel (tj. klient, zákazník, který chce využít služeb poskytovatele) a registr

* **SOAP**

- Simple Object Access Protocol
- je protokolem pro výměnu zpráv založených na XML přes síť
- HTTP i SMTP se dají použít jako aplikační vrstva pro protokol SOAP
- Využívá se např. ke komunikaci s webovovými službami

* Webová služba

- je podle definice W3C řešení, jak spolu aplikace mohou komunikovat a vyměňovat si informace po síti/Internetu.

Návrhové vzory, knihovny vzorů a architektur

Návrhové vzory

- Metodická katalogizace úspěšných návrhů je nejlepší způsob, jak dosahovat vysoké kvality SW a jak zvyšovat produktivitu. (Larman)
- Návrhové vzory = pojmenovaná řešení problémů, která kodifikují příkladový návrh, resp. vhodné principy
- Základní prvky návrhového vzoru
 - **Název** - co nejvíce vystihující podstatu, usnadnění komunikace - společný slovník
 - **Problém** - obecná situace kterou má NV řešit
 - **Podmínky** - popis okolností ovlivňujících použití NV a kontextu vhodném pro použití některé okolnosti mohou být využity při řešení, jiné naopak jsou v konfliktu
 - **Řešení** - soubor pravidel a vztahů popisujících jak dosáhnout řešení problému nejen statická struktura, ale i dynamika chování
 - **Souvislosti a důsledky** - detailní vysvětlení použití, implementace a principu fungování, způsob práce s NV v praxi
 - **Příklady** - definice konkrétního problému, vstupní podmínky, popis implementace a výsledek
 - **Související vzory** - použití jednoho NV nepředstavuje typicky ucelené řešení - řetězec NV okolnosti pro rozhodování mezi různými NV

knihovny vzorů a architektur

- Základní typy vzorů
 - **Creational Patterns (vytvářející)**
 - Creational Patterns řeší problémy související s vytvářením objektů v systému. Snahou těchto návrhových vzorů je popsát postup výběru třídy nového objektu a zajistění správného počtu těchto objektů. Většinou se jedná o dynamická rozhodnutí učiněná za běhu programu.
 - **Structural Patterns (strukturální)**
 - Structural Patterns představují skupinu návrhových vzorů zaměřujících se na možnosti uspořádání jednotlivých tříd nebo komponent v systému. Snahou je zpřehlednit systém a využít možností strukturalizace kódu.
 - **Behavioral Patterns (chování)**
 - Behavioral Patterns se zajímají o chování systému. Mohou být založeny na třídách nebo objektech. U tříd využívají při návrhu řešení především principu dědičnosti. V druhém přístupu je řešena spolupráce mezi objekty a skupinami objektů, která zajišťuje dosažení požadovaného výsledku.

- Knihovny vzorů

- **GoF** - Gang of Four Design Patterns

- využívání ověřených postupů a popsaných úspěšných řešení neustále se opakujících problémů
- 23 vzorů

- **GRASP**

- pojmenované techniky, které podporují Responsibility-Driven Design (RDDes - návrh řízený odpovědnostmi)
- Jedná se o postupy (zásady) správného návrhu
- Používáme jich ve svých hlavách při programování i při kreslení – modelování
- Návrh řízený odpovědnostmi spočívá v uvážení přiřazení odpovědností spolupracujících objektů.

- **POSA**

- Pattern Oriented Software Architecture
- Konkrétnější (i nízkoúrovňové implementační) vzory
- Velké množství vzorů

Přehled současných komponentových komunikačních prostředí

- **CORBA**

- CORBA je zkratkou pro Common Object Request Broker Architecture.
- Jedná se o jazykově nezávislý objektový model a specifikaci pro vývojové rozhraní distribuovaných aplikací.
- vytvářena s cílem podporovat různé sítě, operační systémy a jazyky

- **COM**

- Component Object Model je způsob, kterým mohou komunikovat softwarové komponenty.
- **COM komponenta** je složena ze dvou hlavních částí - Interface (rozhraní) a vlastního COM objektu
- Rozhraní je jedinou možností jak se může program domluvit s a vlastním COM objektem
- **DCOM** je rozšířením COM pro účely distribuovaných systémů. Obsahuje části pro komunikaci mezi COM komponentami umístěných na různých počítačích.
- Výhody a nevýhody COM
 - Výhodou COM je, že je na rozdíl od CORBA distribuováno úplně zadarmo.
 - Nevýhodou je, že rozhraní není obecně podporováno například v UNIXu. Jedná se tedy především o MS technologii.
- **COM+** je rozšíření rozhraní COM. Přidává nové vlastnosti a reviduje zastarávající COM model

- **EJB**

- Enterprise JavaBeans
- standardní komponentní architektura, sloužící pro realizaci aplikační vrstvy informačního systému
- EJB komponenty jsou objekty implementované vývojářem, které zajišťují vlastní aplikační (Business) logiku systému

■ Java RMI + .NET Remoting

- RMI - Remote method invocation
 - umožňuje objektu z jednoho Javoveho Virtualního Stroje (JVM) vyvolávat metody na jiném objektu, který se může nacházet v jiném JVM
 - dokáže lokalizovat vzdálený objekt
 - komunikuje se vzdálenými objekty - komunikace na nižší úrovni je před programátorem skryta
 - v případě dodržení podmínek dokáže přenést mezi danými objekty jiný objekt ve tvaru bajtového kódu
- .NET Remoting
 - Velmi podobné RMI, k instanci objektu v jedné aplikační doméně (procesu) lze přistupovat z jiné aplikační domény.
 - Volání probíhá po tzv. kanálech - TCP nebo HTTP.
 - Přednášení objektů (Marshaling) může být referenční nebo hodnotové.
 - Objekty předávané hodnotou jsou serializovány a kanálem přeneseny
 - Pro objekty předávané odkazem se vytvoří zástupný objekt (tzv. proxy objekt), který zajišťuje komunikaci se vzdáleným objektem.

Návrh distribuovaných systémů

* Cíle návrhu

- Transparentnost
 - přístupová: proces má stejný přístup k lokálním i vzdáleným prostředkům
 - lokační: uživatel (proces) nemůže říct, kde jsou prostředky umístěny
 - migrační: procesy mohou běžet na libovolném uzlu / přemisťovat se
 - perzistentní: uživatel se nemusí starat o stav objektů, ke kterým přistupuje
- Přizpůsobivost:
 - autonimie: každý uzel je schopný samostatné funkčnosti
 - decentralizované rozhodování: každý uzel vykonává rozhodnutí nezávisle na ostatních
 - otevřenosť: lze zapojit různé komponenty vyhovující rozhraní
- Spolehlivost
 - teorie
 - nespolehlivost 1 serveru = 1 %
 - nespolehlivost 4 serverů $0,01^4 = 0.00000001$
- Výkonnost
 - teorie: více uzlů = vyšší výkon
 - praxe: výrazně nižší než lineární nárůst výkonu
 - příčiny: komunikace, synchronizace, komplikovaný software
- Rozšiřitelnost
 - snaha vyhnout se všemu centralizovanému (serverům, službám, tabulkám,...)

* Distribuované systémy a jazyk JAVA

- Sockety
 - dostačující pouze pro běžnou komunikaci
 - nevýhodou je nutnost definovat komunikační protokol a nutnost realizovat pro každou komunikaci nové spojení
- RMI
 - RMI aplikace obsahuje dva oddělené programy: klient a server. Server vytvoří určitý počet vzdálených objektů, zpřístupní reference na tyto vzdálené objekty a čeká na klienty, kteří tyto objekty využijí vzdáleným voláním jejich metod. Klient přistupuje k objektu implementovaném na serveru pomocí svého lokálního zástupce stejným způsobem, jako přistupuje ke svým lokálním objektům.
 - Při návrhu takovéto distribuované aplikace je pak možno použít návrhových metod stejných jako při tvorbě nedistribuovaných aplikací. Toto je umožněno tím, že veškerá komunikace mezi jednotlivými částmi distribuovaného systému se řeší na úrovni JVM (Java virtual machine – virtuálního stroje Javy).

Komponentový návrh

* druhy komponent

- hotové komponenty
- komponenty se zkušeností
- komponenty s částečnou zkušeností
- nové komponenty

* doporučení v návrhu

- Pokud již existuje hotová komponeta, která odpovídá projektovým specifikacím, je vhodné ji získat. Cena za nákup a integraci je skoro vždy menší než cena za vývoj.
- Pokud jsou dostupné komponenty se zkušeností, riziko spojené s jejich modifikací a integrací je obecně přijatelné
- Pokud jsou dostupné komponenty s částečnou zkušeností, jejich použití pro projekt musí být analyzováno. Pokud je potřeba komponentu upravit, musíme být opatrní, neboť cena za úpravu může být větší než cena za vývoj nových.

* výhody

- Rozdělení systému do komponent je architektonicky velmi výhodné. Systém se rozčlení do menších logických celků, které se pak mohou skládat z dalších komponent (subsystémů).
- Snadno se oddělí část programu, která může být použita i při vývoji jiného systému. (znovupoužitelnost)
- Zvýšení výkonu programů
 - kód opakovatelně využitelné komponenty se může optimalizovat a každá optimalizace se současně projeví ve všech programech, ve kterých je komponenta použita
- Zvýšení kvality programů
 - opakovatelně použitá komponenta je testována častěji a zjištěné chyby jsou odstraněny ve

všech programech, kde je komponenta použita

* nevýhody

- Někdy mohou být vysoké náklady na vývoj komponent, protože je nutné vyvíjet obecnější kód.
- V době kdy se vybírá komponenta obvykle existuje o vyvíjeném programu pouze hrubá představa. Proto je někdy problém vybrat vhodnou komponentu.
- Problém s údržbou programu – dodavatel komponent může mít jiné priority než dodavatel programu, který komponentu používá a nemusí mít motivaci komponentu upravovat.

si/si17.txt · Poslední úprava: 2009/05/06 22:20 autor: Funic

Otázka 18 - Y36SI3

Zadání: Metriky a jejich využití při návrhu. Zajištění výkonu, bezpečnosti, spolehlivosti a znuvupoužitelnosti při návrhu. Měření a vyhodnocení návrhu.

slovnicek pojmu

- DSM - Design Structure Matrix je matici pro identifikaci samostatných a nezávislých modulů
- LI - Level of Independence je metrika pro určení stupně nezávislosti modulu
- LOC - Line of Code, řádek zdrojového kódu
- OOD - Objektově orientovaný návrh
- AOP - Aspektově orientované programování

Úvod do problematiky

Zhruba jde o to že softwaroví inženýři by rádi formální posouzení kvality designu - nejen na základě zkušenosti (což je podle mého názoru nejrozumnější způsob), ale i na základě sady pravidel a postupů - o jejich definici se pokouší práve ATAM. Ideálním cílem takové snažení je nástroj který zanalyzuje UML model a hledá vzájemné závislosti modulů (loose/tight coupling, separation of concerns - rozdělení software do vrstev které spolu komunikují přes definovaná rozhraní) a jeho výsledkem je nějaké číslo na stupnici „špatně“ až „výborně“. Je to poměrně nová oblast a zatím se spíš vymýšlejí ty metriky než že by se to rutinně používalo, ale už existují softwary které tvrdí že to umí - <http://www.sdmetrics.com/> [<http://www.sdmetrics.com/>].

Podobné automatické nástroje existují i pro zdrojový kód, tato oblast je o něco dál - například Rational má nástroje pro statickou analýzu kódu, která posuzuje, jak moc se v kódu dodržují konvence programovacího jazyka a používají Design Patterns atd...

Softwarové metriky

Metriky produktu, procesu a projektu jsou kvalitativní charakteristiky programů, procesu jejich tvorby a projektu.

Důvody pro měření metrik:

- plánování projektu (odhad nákladů, pracnosti, času)
- kontrola kvality produktu
- odhad produktivity
- zdokonalení práce (růst výkonnosti organizace)

Používané metriky vychází převážně z „historických zkušeností“:

- Jaké byla produktivita vývoje minulých projektů?
- Jaká byla kvalita vytvořeného softwaru?

- Jak mohou být tato data, týkající se kvality a produktivity, extrapolována na současný projekt?
- Jak nám pomůže minulost při plánování a odhadech současného projektu?

Základní klasifikace metrik

Ukazatel (indikátor) je metrika nebo kombinace metrik, které poskytují náhled na softwarový projekt, proces nebo produkt. Slouží k jejich hodnocení, aby bylo možné případně provést nápravu.

Ukazatelé procesu umožní náhled na efektivitu existujícího procesu. Metriky procesů jsou sbírány dlouhodobě v průběhu řešení různých projektů. Mají indikovat zlepšení softwarového procesu. (strategie)

Ukazatelé projektu umožní odhadnout stav projektu, potenciální rizika, odkryt oblasti problému, dřív než budou kritické, přizpůsobit směr práce a úkolů, vyhodnotit schopnosti projektového týmu ředit kvalitu SW. (taktika)

Určujícími prvky kvality softwaru a efektivity organizace jsou:

- Zkušenosť a motivace lidí
- Složitosť produktu
- Technologie (metody softwarového inženýrství)
- obchodní podmínky (obchodní pravidla, termíny..)
- charakteristiky zákazníka (snadnosť komunikace..)
- vývojové prostředí (CASE)

Metriky softwarového procesu mohou hrát významnou roli v růstu výkonnosti podniku, ale mohou být i zneužity a přinášet víc problémů, než užitku.

Aby nebyly metriky zneužívány, měly bychom je používat v souladu s **následujícími doporučeními**:

- Při interpretaci dat používej zdravý rozum a organizační cit.
- Poskytuj zpětnou vazbu týmům a jednotlivcům, kteří sbírali data a prováděli daná měření.
- Nepoužívej metriky k hodnocení jednotlivců.
- Ved' jednotlivce a týmy k tomu, aby si stanovili jasné cíle a metriky, s jejichž pomocí by těch cílů dosáhli.
- Nikdy nepoužívej metriky k vyjednávání s jednotlivci ani s týmy.
- Data, která indikují nějaký problém, nemají být považována za negativní - jsou to pouze data, která poslouží k zlepšení procesu.
- Nezaměř se jen na jednu metriku, nezapomeň na ostatní důležité metriky.

Metoda SSPI (Statistical Software Process Improvement)

Tato metoda definuje následující postup:

- chyby a defekty jsou kategorizovány podle původu (chyba ve specifikaci, v logice..)
- je stanovena cena za opravu chyby
- sečte se počet chyb podle kategorie
- je stanovena celková cena chyb podle kategorie

- analyzují se kategorie s nejdražšími chybami
- snaha modifikovat proces, aby se eliminovala frekvence výskytu chyb v této kategorii

Projektové metriky

Projektové metriky slouží k taktickým účelům (odhady času, nákladů, k monitorování projektu a k jeho vyhodnocování).

Model projektové metriky:

- **vstupy**: měří se zdroje (lidé, zařízení..) potřebné pro práci
- **výstupy**: měří se předané výstupy nebo produkty vytvořené během projektu
- **výsledky**: měří se efektivita (užitečnost) výstupů

Tato metrika se může použít i k měření procesu a projektu. U projektu ji lze použít postupně pro každou vývojovou fázi.

Měření softwaru

Při měření softwaru používáme dva odlišné přístupy:

- **přímá měření** - počet řádek kódu (LOC), rychlosť výpočtu, velikost paměti, počet chyb za určitou dobu ...
- **nepřímá měření** - funkčnost, kvalita, složitost, pracnost, spolehlivost, schopnost údržby,...

Metriky orientované na velikost (size-oriented)

Tato metrika je odvozená z velikosti produktu a normalizovaná faktorem kvality či produktivity. Vychází se statistiky minulých projektů.

Vstupní hodnoty odvozené z předcházejících projektů zahrnují:

- počet řádek kódu (LOC)
- pracnost (m/m)
- cena
- počet stran dokumentace
- počet chyb
- počet defektů
- počet realizátorů

Z těchto údajů můžeme odvodit:

- počet chyb na KLOC
- počet defektů na KLOC
- cena LOC
- počet stran dokumentace na KLOC
- počet chyb za člověkoměsíc

- počet LOC za člověkoměsíc
- cena stránky dokumentace

Funkčně orientované metriky

Základní veličinou, se kterou pracujeme je **FP** (function point, funkční bod).

Tato veličina je dána empirickým vztahem mezi spočitelným měřením informační domény a odhadem složitosti softwaru.

Rozhodující pro její určení jsou údaje:

A	počet logicky různých vstupů
B	počet výstupů
C	počet dotazů
D	počet vnitřních logických souborů
E	počet souborů na rozhraních

Příklad výpočtu FP

měrný faktor			váhový faktor			
Počet	jednoduchý		průměrný		složitý	
A:	(3 a ₁	+	4 a ₂	+	6 a ₃)	=
B:	(4 b ₁	+	5 b ₂	+	7 b ₃)	=
C:	(3 c ₁	+	4 c ₂	+	6 c ₃)	=
D:	(7 d ₁	+	10 d ₂	+	15 d ₃)	=
E:	(5 e ₁	+	7 e ₂	+	10 e ₃)	=
		+		+	TOTAL	=

$$FP = TOTAL \times (0,65 + 0,01 \times \text{suma } (F_i))$$

F_i pro $i = 1..14$ (i je složitost zpracování).

Mají hodnotu 0 – 5

hodnota F_i	vliv
0	nemá vliv
1	nahodilý
2	mírný
3	průměrný
4	významný
5	podstatný vliv
hodnota F_i	význam
F_1	Požaduje systém zálohování a obnovu?

F ₂	Jsou požadovány datové přenosy?
F ₃	Obsahuje funkce distribuovaného zpracování?
F ₄	Je požadován kritický výkon?
F ₅	Bude systém pracovat za silného provozu?
F ₆	Požaduje systém přímý vstup dat?
F ₇	Požadují vstupní transakce přímé vstupy dat prostřednictvím více obrazovek nebo operací?
F ₈	Jsou hlavní soubory aktualizovány přímo?
F ₉	Jsou vstupy, výstupy, soubory a dotazy složité?
F ₁₀	Je složité vnitřní zpracování?
F ₁₁	Je kód navržen pro opakování použití?
F ₁₂	Jsou v návrhu zahrnuty konverze a instalace?
F ₁₃	Je systém navržen pro vícenásobné instalace na různých místech?
F ₁₄	Je aplikace navržena tak, aby usnadnila změny a snadné uživatelské ovládání?

Rozšířené funkční metriky

Třírozměrná funkční metrika - *3D Function Point* pracuje ve třech rozměrech.

K hodnotám A – E, ze kterých jsme odvozovali hodnotu FP, přichází další dva rozměry s hodnotami F a G:

1. **datový** (jako FP)
2. **funkční** (F - počet vnitřních operací potřebných k transformaci vstupních dat na výstupní)
3. **řídicí** (G - počet přechodů mezi stavý)

měrný faktor			váhový faktor			
Počet	jednoduchý		průměrný		složitý	
A:	(3 a ₁	+ 4 a ₂	+ 6 a ₃)	=		
B:	(4 b ₁	+ 5 b ₂	+ 7 b ₃)	=		
C:	(3 c ₁	+ 4 c ₂	+ 6 c ₃)	=		
D:	(7 d ₁	+ 10 d ₂	+ 15 d ₃)	=		
E:	(5 e ₁	+ 7 e ₂	+ 10 e ₃)	=		
F:	(7 f ₁	+ 10 f ₂	+ 15 f ₃)	=		
G:	(1 g ₁	+ 1 g ₂	+ 1 g ₃)	=		
			3DFP	=		

Vztahy mezi metrikami

Hrubý odhad počtu LOC na 1 FP pro různé programovací jazyky:

Assembler	320
-----------	-----

C	128
Cobol	105
Fortran	105
Pascal	90
Ada	70
objektově orientované jazyky	30
4GLs	20
tabulkové kalkulátory	6
grafické jazyky (ikony)	4

Faktory ovlivňující produktivitu tvorby softwaru

Produktivitu softwaru ovlivňuje řada faktorů, některé jsou objektivní jiné subjektivní. Některé se ovlivňují snadno, jiné se prakticky nedají ovlivnit.

Hlavní faktory, které ovlivňují tvorbu softwaru jsou:

- Lidský faktor (velikost a zkušenosti firmy)
- Faktor problému (složitost problému, počet změn v návrhu omezení a požadavků)
- Faktor procesu (techniky analýzy a návrhu, jazyk a CASE, techniky kontroly)
- Faktor produktu (chování a spolehlivost systému)
- Faktor zdrojů (dostupnost CASE nástrojů, HW a SW zdrojů)

Metriky kvality softwaru

Jak se dá měřit kvalita softwaru? Umíme posoudit složitost použitých algoritmů, ale ověřit správnost a spolehlivost softwaru je složitější.

Správnost (correctness) se měří nejčastěji počtem chyb na tisíce řádku kódu (KLOC).

Udržovatelnost (maintainability) je snadnost, s níž může být program opraven, když se vyskytne chyba, jak rychle může být upraven, změní-li se prostředí, jak může být zdokonalován, rozhodne-li se zákazník upravit požadavky.

Používáme zde **přímé metriky**.

Používáme metriku MTTC (*mean-time-to-change*) – střední doba potřebná pro realizaci změn tj. doba potřebná pro analýzu požadovaných změn, návrh, implementaci, otestování a distribuci uživatelům.

Rozeznáváme:

- **Spoilage** - cena za opravu defektů objevených po distribuci.
- **Integrita** - odolnost vůči náhodným nebo záměrným útokům z vnějšího prostředí (viry, hackers).
 - **integrita** = suma $(1 - \text{hrozba} \times (1 - \text{bezpečnost}))$; hrozba je pravděpodobnost útoku speciálního typu; bezpečnost je pravděpodobnost odolání útoku speciálního typu; suma je součet přes všechny typy útoků
- **užitečnost** (usability) - je snaha změřit „uživatelskou přívětivost“.

Čtyři charakteristiky dobrého softwaru z pohledu zákazníka:

1. fyzická a intelektuální dovednost nutná pro zvládnutí systému
2. potřebný na to, aby uživatel byl mírně zdatný v ovládání systému
3. vzrůst produktivity vzhledem k systému, který produkt nahrazuje, jestliže je uživatel mírně zdatný
4. subjektivní ocenění uživatelů (dotazníkem)

Měření účinnosti odstraňování chyb softwaru

DRE (*Defect Removal Efficiency*)

DRE = E / (E + D), kde E je počet chyb nalezených před předáním uživateli D je počet defektů po předání.

Hodnotu DRE definujeme vzhledem k jednotlivým vývojovým fázím:

DRE_i= E_i / (E_i + E_{i+1}), kde E_i je počet chyb nalezených během SI etapy i. E_{i+1} je počet chyb nalezených během SI etapy i+1, které vznikly vlivem neobjevených chyb v etapě i.

Závěr

- Sbírám-li data o projektu, procesu a vytvářeném produktu, provádím-li měření a počítám-li metriky, mám možnost srovnání a poučení se vlastních chyb.
- Bez měření nepoznám, zda se zlepšuji.

ATAM

ATAM (Architecture Tradeoff Analysis Method) je strukturovaná technika sloužící k pochopení kompenzací vyplývajících z architektury software-intensive systémů. Tato metoda byla vyvinuta k poskytnutí principu, jak hodnotit způsobilost softwarové architektury s ohledem na atributy kvality, které jsou: přizpůsobitelnost, bezpečnost, výkon, dostupnost, a další. Tyto atributy se vzájemně ovlivňují, a zlepšení často přichází za cenu zhoršení jednoho nebo více z nich. Tato metoda pomáhá s důvody architektonických rozhodnutí, která ovlivňují kvalitu interakcí atributů. ATAM využívá spirálového modelu, známý z OOD, který s analýzou a změnou rizik vede k rafinované architektuře. Tato technika se využívá v ranném stádiu procesu vývoje aplikace, tak aby se zjistilo, že daná architektura je dobrá, to na základě kompromisů (tradeoff). Najde rizikové a jinak význačné body na architektuře. Je vyvinutá tak, aby byla provedna rychle a levně.

Architecture Tradeoff Analysis

Kvalita atributů velkých softwarových systémů se v zásadě určuje podle softwarové architektury systému. To znamená, že u velkých systémů, dosažení kvalit, jako je výkon, dostupnost a přizpůsobitelnost, závisí spíše na celkové architektuře software než na codelevel postupech, jako je volba jazyka, podrobný návrh, algoritmy, datové struktury, testování, a tak dále. Tím nechci říct, že výběr algoritmu nebo datové struktury není důležitý, ale spíše, že tyto volby nejsou tak zásadní, aby systém byl úspěšný než jeho celková software struktura, architektura.

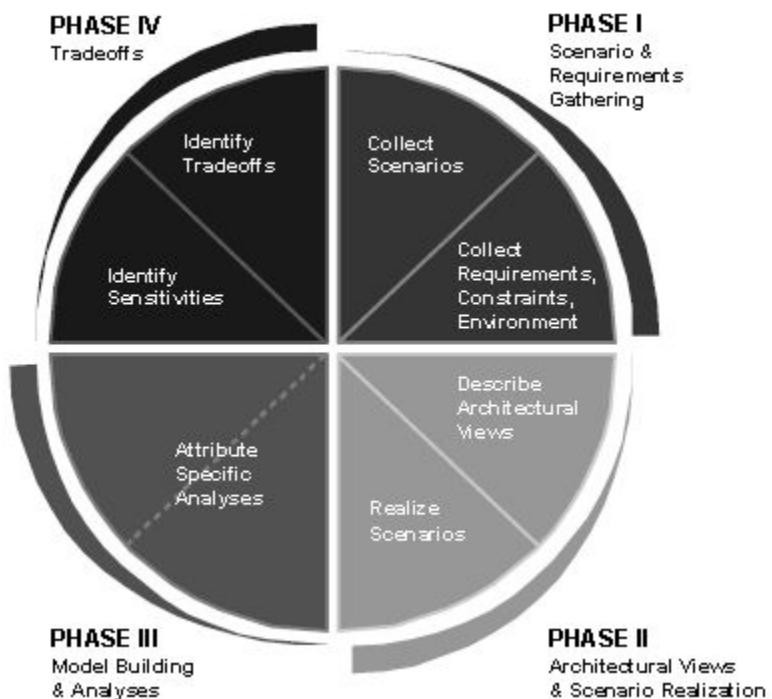
ATAM navazuje na **SAAM** (Software Architecture Analysis Method). Ta sloužila k vyhodnocení: pro přizpůsobitelnost, výkonnostní analýza, analýza dostupnosti a bezpečnostní analýzy. SAAM se již

úspěšně používá k analýze architektury z široké škály oblastí: softwarových nástrojů, řízení letového provozu, finanční řízení, telefonie, multimédia, vestavěný kontroly vozidel, a tak dále

Architecture Tradeoff Analysis Method (ATAM)

V ATAM, stejně jako běžném spirálovém modelu, se každá iterace se zabývá jiným chápaním systému a sníží jeho riziko a rozrušení designu (návrhu). Narozdíl od standardní spirály se zde neimplementuje, že každá iterace je motivována výsledky analýzy a v nové iteraci jsou výsledky více rozvinuté a infomované od návrhu.

Analýza architektury zahrnuje srovnání, kontrolu, měření několika sad architektonických prvků, faktorů prostředí a architektonických omezení. Primárním úkolem architekta je navrhnut architekturu, která povede k chování systému, které je co nejbliže požadavkům s co nejmenšími náklady. Například požadavky na výkon jsou uvedeny v podobě zpoždění a/nebo propustnosti. Nicméně, tyto atributy jsou závislé na architektonických prvcích, týkajících se přidělování zdrojů: politika pro přidělování procesů procesorům, plánování souběžných procesů na jeden procesor, nebo řízení přístupu ke sdíleným datovým uložištěm. Architekt musí porozumět vlivu těchto architektonických prvků na schopnost systému ke splnění jeho požadavků a manipulovat s těmito prvky vhodně.



Nicméně, tento úkol je většinou postižen nedostatkem nástrojů. Nejlepší architekti používají své tušení, zkušenosti s ostatními systémy, a prototypování, které je navádí. Příležitostně explicitní modelovací krok zahrnuje navrhovací (designovou) aktivitu, nebo explicitní formální analýzu jediného atributu kvality.

Kroky metody

Tato metoda je rozdělena do čtyř hlavních oblastí: shromažďování scénářů a shromažďování požadavků, architektonické názory a realizaci scénáře, stavění modelu a analýzy, a kompenzací (tradeoffs). Obecně platí, že metoda funguje takto: jakmile je do systému zavedena inciliazace množinou scénářů a

požadavků a iniciázace architektury či malé množiny architektonických návrhů o (v závislosti na prostředí a dalších úvah), každý atribut kvality je podle pořadí v izolaci vyhodnocen s ohledem na všechny navrhovaný design. Po této hodnocení přichází kritický krok. Během tohoto kroku, tradeoff body-prvky, které mají vliv na více atributů jsou nalezeny. Po tomto kroku můžeme bud: zpřesnit modely a opětovně vyhodnotit; zpřesnit architekturu, změnit, aby odrážela tyto upřesněné modely a opětovně ocenit; nebo změnit některé požadavky. Následuje detailní popis každého kroku.

Krok 1 — Shromažďování scénářů - Collect Scenarios

První a druhý krok v metodě - zjištění systémových scénářů a sběru požadavků, omezení, a informací o prostředí od zástupce skupiny investorů (zákazníka) - jsou zaměnitelné. Občas požadavky existují před zahajením architektonické analýzy. Jindy se scénáře řídí požadavky. Může také začít analýza procesů, je to výhodné. Vyvolává scénáře slouží stejným účelům jako v SAAM: ke zpracování funkčních a kvalitativních požadavků, aby usnadnil komunikaci mezi zúčastněnými stranami a to vytvořit společnou vizi z důležitých činností, které by systém měl podporovat.

Krok 2 — Shromažďování požadavků - Collect Requirements/Constraints/Environment

Kromě scénáře, atributy založené na: požadavcích, omezeních a prostředí, musí být v systému označeny. Požadavek může mít určitou hodnotu nebo může být obecnější, jak vyplývá ze scénáře hypotetických situací. Prostředí musí být charakterizováno, protože následné analýzy (např. výkon nebo zabezpečení) omezení prostoru designu (návrhu), jak se vyvíjejí, jsou zaznamenány, protože ty také ovlivňují atributové analýzy. Tento krok se klade silný důraz na přehodnocení scénáře z předchozího kroku, aby se zajistily důležité atributy kvality.

Krok 3 — Describe Architectural Views

Požadavky, scénáře a inženýrské designové principy dohromady vytvářejí kandidáta na architekturu a vymezují prostor designovým možnostem. Navíc design skoro nikdy nezačíná z čistého konta: stávající systémy, interoperabilita, a úspěchy / neúspěchy předchozích projektů, to vše omezuje prostor architektury. Kandidát na architekturu musí být popsán v termínech architektur, elementů a vlastností, které jsou důležité pro každého z důležitých atributů kvality. Například replikace a hlasovací systémy jsou důležitým prvkem pro spolehlivost; souběžný rozklad, proces priority, propustnost odhadů a řazení systémy(schemes) jsou důležité pro výkon; firewally a intruder modely jsou důležité pro bezpečnost; abstrakce a zapouzdření je důležité pro přizpůsobitelnost. Kromě toho, údaje požadované pro analýzu každé z těchto vlastností jsou obvykle zachyceny v různých architektonických pohledech (možnostech). Definování několika takových zobrazení pro použití v architektonické analýze: a module view (pro úvahu okolo práce s úkoly a schovávání infomrací), process view (ohledně výkonu), dataflow view (ohledně funkčních požadavků), class view (ohledně sdílení objektových definicí) a další.

Existuje další důležitý bod, který dotváří architektonické zastoupení: v tomto podání ATAM se domníváme, že jsou více srovnávány konkurenční architektury. Nicméně, designy obvykle pracují pouze na jediné architektuře najednou. Z našeho pohledu architektura je sada funkcí přidělená množině konstrukčních prvků, které popisují soubor pohledů či názorů. Téměř každá změna bude mutovat jeden z těchto názorů (pohledů), což vede k nové architektuře. Zatímco tento bod by mohl vypadat jako dělení chloupek, je důležité chápout chloupy v této souvislosti z tohoto důvodu: ATAM vyžaduje budování a udržování atribut modelů (kvantitativních i kvalitativních modelů), které budou odrážet a pomáhat architektuře. Chcete-li změnit některý aspekt architektury-functionality, strukturní prvky, koordinace modelů bude mít vliv na jeden nebo více modelů. Jakmile je změna navržena, nová a stará architektura jsou „konkurenční“ a musí být

porovnány: proto je nutné udělat nové modely, kterí reflektují tyto změny. Pomocí ATAM je neustálý výběr mezi konkurenčními architektury „jasný“, i když mohou vypadat „skoro stejné“ pro příležitostné pozorovatele.

Krok 4 — Attribute-Specific Analyses

Jakmile systém vyvolá inicializační množinu požadavků a scénářů je navrhнута počáteční architektura (nebo její část) a každý atribut kvality musí být analyzován v izolaci s ohledem na každou architekturu. Tyto analýzy mohou být provedeny v libovolném pořadí, protože se neděje v tomto okamžiku porovnání mezi jednotlivými atributy vůči požadavkům a interakci mezi atributy. Povolení oddělených (souběžných) analýz je důležité pro **separation of concerns**, protože umožňuje individuální atributům přinést jejich odborné znalosti na systém.

Výsledek analýzy vede k prohlášení o chování systému s ohledem na hodnoty konkrétních atributů: „žádá, aby se reagovalo do 60 ms. průměr“, „průměrná doba do selhání je 2,3 dny“, „je systém odolný proti útoku známých skriptů“; „hardware stojí 80000 dolarů pro platformu“, „software bude vyžadovat 4 lidí ročně k udržbě“, a tak podobně.

Krok 5 — Identify Sensitivities

Tento krok určuje citlivost jednotlivých atributů analýzy pro konkrétní architektonické prvky; tzn jedne nebo více atributů je měněno. Různorodé modely zachycují tyto změny designu a výsledky jsou vyhodnoceny. Jakákoli z modelovaných hodnot, která je výrazně ovlivněna změnou v architektuře se považuje za citlivý bod.

Krok 6 — Identify Tradeoffs

Dalším krokem této metody je vyhodnocení modelů postavených v kroku 4 a najít architektonický tradeoff (obchod) bodů. Ačkoli je běžnou praxí kritika vzorů, další významný vliv lze získat pro tuto kritiku tím, že se zaměří na interakci atributo-specifické analýzy, zejména umístění obchod bodů. Takto je to hodnoceno.

Jakmile architektonicky citlivé body byly určeny, hledání obchod bodů je pouze identifikací architektonických prvků, které mají několik atributů, které jsou citlivé. Například, když jde o klient-server architekturu mohla být velmi citlivá na počet serverů (výkon se zvyšuje, v některých rozsahu zvýšením počtu serverů). Dostupnost této architektury také zavисí na přímé změny počtu serverů. Nicméně, bezpečnost systému se může lišit inverzně s počtem serverů (protože systém obsahuje více možných bodů útoku). Počet serverů, je pak tradeoff (obchod) bod v souvislosti s touto architekturou. Ten je pak jedne z mnoha potencionálních bodů, který vede, vědomě či nevědomě, ke změně architektury.

Iterace ATAM

Poté, co jsme dokončili výše uvedené kroky, můžeme porovnat výsledky analýz s požadavky. Když analýzy ukazují, že systém je dostatečně blízko chování k jeho požadavkům, mohou návrháři přikročit k podrobnějšímu úrovni designu nebo k realizaci. V praxi je však užitečné, aby i nadále byly sledovány architektury s analytické modely na podporu vývoje, nasazení a údržbu (development, deployment, and maintenance). Design nikdy nevypadá ze životního cyklu systému, stejně tak i analýza. V případě, že analýza ukazuje na nějaký problém, jsme nyní schopni vypracovat akční plán pro změnu architektury, modelu, či požadavků. Akční plán bude vycházet z attributespecific analýzy a identifikace obchod (tradeoff) bodů. To pak vede k další iteraci metody. Meli bychom si ujasnit, že neočekáváme, že tyto kroky

následují lineárně. Mohou se ovlivnit s ostatními komplexními způsoby: analýza může vést k přehodnocení požadavku; přebudování modelu může poukázat na místa, kde architektura nebyla dostatečně promyšlená a zdokumentována. To je důvod, proč se zachycují kroky jako kousky v kruhu: ve středu kruhu se každý krok dotkne (a výměnu informací s) ostatními dalšímy kroky.

Výhody ATAM

- Podporuje získávání přesných požadavků na jakost
- Vytváří dřívější start na dokumentaci architektury
- Vytváří základ pro architektonické rozhodnutí
- Podporuje identifikaci nebo rizik v počátku životního cyklu
- Podporuje zvýšenou komunikaci mezi zúčastněnými stranami

Measuring Software Design Modularity

Tento článek pojednává o metrikách, které hodnotí návrh z hlediska rozdělování na samostatné a nezávislé části/moduly. Hlavní myšlenka tkví v tom, že čím více nezávislých částí/modulů, tím více lidí respektive týmu na daném projektu může pracovat najednou a tak může být projekt realizován za kratší dobu. Metrika, která je zde popisována je tzn. LI - Level of Independence, který je definován jako podíl velikosti nezávislého modulu a celkové velikosti. Čím více proměnných v nezávislých modulech, tím větší část systému lze vyvíjet nezávisle, čím vyšší hodnota LI tím lépe.

Design Structure Matrix (DSM) zobrazuje závislosti jednotlivých částí projektu mezi sebou (křížky), na diagonále jsou pak vyznačovány nezávislé moduly, jak ukazuje obrázek níže. Více o DSM je možné najít na [Design Structure Matrix](http://129.187.108.94/dsmweb/en/understand-dsm/tutorials-overview/descripton-design-structre.html) [<http://129.187.108.94/dsmweb/en/understand-dsm/tutorials-overview/descripton-design-structre.html>]

Článek:

Softwarový vývojáři se odkazují na funkce, třídy, nebo modulární komponenty a velké množství prací, které jsou dělány s mírou propojení (coupling) a soudržnosti (cohesion) mezi jednotlivými moduly. Moderní programovací vzory jako jsou OOD a AOP rozšiřují definici modulů jako novou strukturu programování, Odborníci vedli spoustu empirických studií oceňující dopad jiných programových vzorů na softwarovou modulárnost.

Všímáme si dvou převládajících definicí modulu a modulárních metod. První definice - Parnasova je široce uznávanou definicí modulu, který je dle této definice nezávislý na přidělení úkolu (independent task assignment) (moduly lze vyvíjet paralelně, bez toho, aniž by spolu jednotlivé teamy musely příliš komunikovat). V důsledku nezávislosti, funkce, třídy, nebo aspekty nezachycují jak lze návrh (design) rozdělit do samostatného a nezávislého přidělování úkolů, tak aby mohly být plněny současně. Další převládající metriky, jako jsou propojení (coupling), soudržnost (cohesion), rozdělení úloh (separation of concerns), které jsou obvykle použity k posouzení kvality zdrojového kódu, ale ne damotného návrhu (design), který má přednost před kódováním. Otázkou zůstává jak může návrh (design) podporovat nezávislé a modulární implementace, vývoj nelze měřit účinně pomocí běžně používaných metrik. Například převládající rozdělení úloh (separation of concerns) je hodnoceno pomocí míry rozdělení/přidělení úloh v kódu, ale ne pomocí rozšíření návrhu (design), které může zajistit paralelní vývoj.

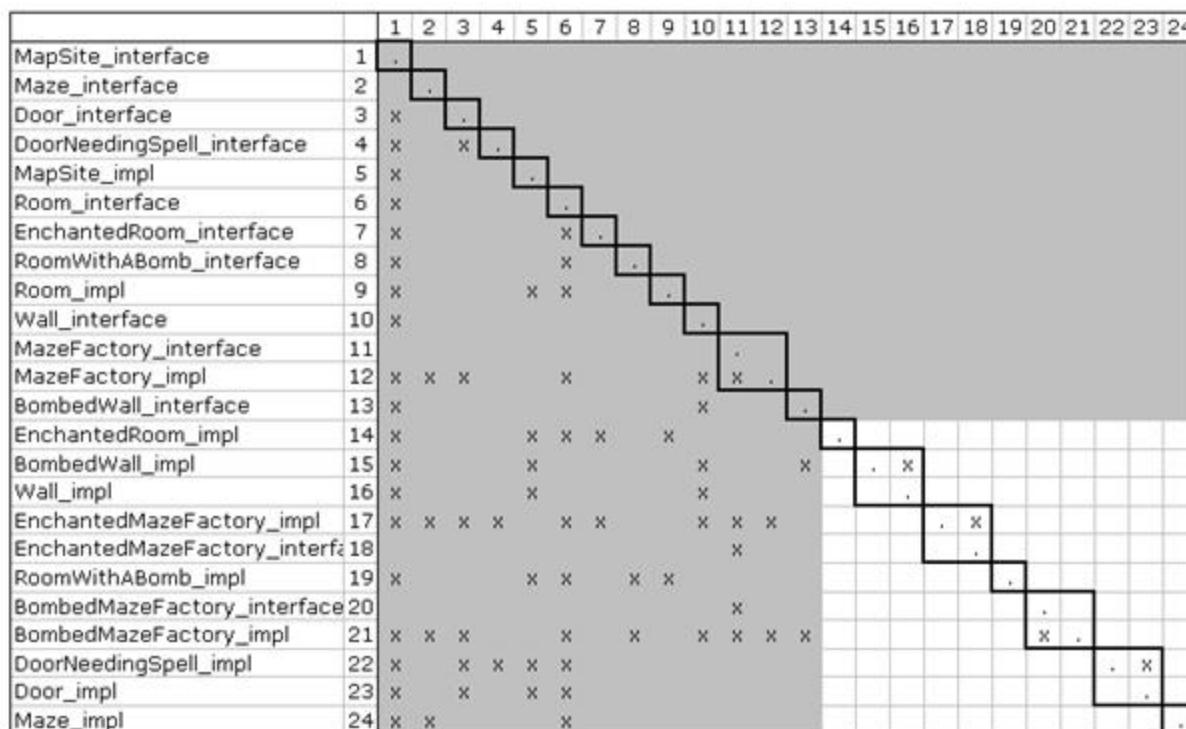
Navrhujeme nové metriky založené na návrhové (design) strukturální matice (DSM), které pomohou určit

míru, do kterých je software schopný podporovat nezávislé paralelní implementace. Používáme Parnasova definici modulu, nezávislé přiřazování úkoly (independent task assignments) a ukázku jak může být modul zachycen v DSM. Dále navrhujeme rozvíjení metody automatického clusteru DSM, který zajistí možnost automatického propojení modulů. V další části zavedeme definici DSM, modulů a nové modulární metriky.

Parnasova definice modulů může být obecně platná bez ohledu na použité programovací paradigmaty a může být použita při návrhu (design) a implementaci. Podobně může být obecně použita modulární metrika v různých tradičních i soudobých programovacích paradigmach. DSM modelování rozděluje pomezí návrhu (design) a implementace, protože obě tyto části mohou být modelovány současně ve stejném modelu DSM. Důsledkem toho je, že pomocí odvozené modulární metriky můžeme posoudit jak moc dobře návrh (design) podporuje nezávislé přiřazování úkoly (independent task assignments), způsob rozdělování, řešení úloh atd. Ukažme si menší příklad pro ilustraci definice a myšlenek.

Příklad

Tato část ukazuje, jak nezávislé moduly mohou být zachyceny v DSM a v nových metrikách modularity. Obrázek ukazuje DSM modelování návrhu hry bludiště s použitím návrhového vzoru Abstract Factory. DSM je čtvercová matici, ve které jsou sloupce a řádky označeny pomocí návrhových proměnných, buňky obsahují modely, které návrh (design) popisuje.



Předchozí práce ukázala, že důležité při návrhu (design) software je pojem „design rules“. Použití „design rules“ je pevné rozhodnutí, které slouží jako rozhraní. V DSM mohou být „design rules“ modelovány jako předchozí proměnné.

Figure 1: Maze Game DSM

Šedá oblast je dopadu „design rules“, tzn. proměnné 1-13 jsou rozhraní, které jsou viditelné na více než jednom podřízeném návrhovém rozhodnutí.

Definice DRs definuje modul jako nezávislý blok podél úhlopříčky v DSM. Nezávislým blokem je myšlen soubor návrhových proměnných, které jsou závislé pouze na „design rules“, ale na jiných nezávislých blocích. Na obrázku jsou zobrazeny moduly v řádcích/sloupcích 14-24. Tmavá pole indikují moduly. Například proměnná 17 a 18 jsou agregovány do jednoho modulu, což znamená, že při návrhu a implementace tohoto modulu, může být provedena nezávisle, jsou-li stanoveny „design rules“.

Výše uvedené definice nám umožňují definovat novou modulární metriku: stupeň nezávislosti (LI) tj. Velikost nezávislého modulu/celková velikost. Důvodem je skutečnost, že čím více proměnných v nezávislých modulech, tím větší část systému lze vyvijet nezávisle. LI hry bludiště za pomocí návrhu z obrázku je 0,46. LI pro information hiding design je 60% a LI pro sequential design je 44% což ukazuje, že information hiding design je více modularizovatelný. *Vypočty jsou převzанé z jiné práce.*

Závěry

Tento poster představuje novou metriku modularity, stupeň nezávislosti (LI), které jsou založeny na Parnasově definici modulu a matici DSM. Předběžně jsem zhodnotili několik kanonických a „design rules“, které prokazují, že tato metrika má potenciál k porovnat architekturu aplikace.

Metrika potřebuje další rozvoj pro různé velikosti modulů. Například je možné, že velký počet proměnných se agreguje do jednoho velkého nezávislého bloku, který bude potřeba dále rozdělit. Na druhou stranu, pokud takový velký blok bude existovat jako nezávislý modul, pak může uživatel okamžitě vizualizovat z DSM matice

Zdroje

- Mannová - Skripta 4. kapitola. http://stm-wiki.cz/index.php/Soubor:Y36SI2_skripta.pdf [http://stm-wiki.cz/index.php/Soubor:Y36SI2_skripta.pdf]
- The Architecture Tradeoff Analysis Method, Rick Kazman, Mark Klein, Mario Barbacci, Tom Longstaff, Howard Lipson, Jeremy Carriere. <http://www.sei.cmu.edu/pub/documents/98.reports/pdf/98tr008.pdf> [<http://www.sei.cmu.edu/pub/documents/98.reports/pdf/98tr008.pdf>]
- Measuring Software Design Modularity - Yuanfang Cai, Sunny Huynh. http://www.comp.lancs.ac.uk/computing/ACoM.08/papers/cai_huynh.pdf [http://www.comp.lancs.ac.uk/computing/ACoM.08/papers/cai_huynh.pdf]
- ATAM: Method for Architecture Evaluation - Rick Kazman, Mark Klein, Paul Clement. <http://www.sei.cmu.edu/pub/documents/00.reports/pdf/00tr004.pdf> [<http://www.sei.cmu.edu/pub/documents/00.reports/pdf/00tr004.pdf>]

Otázka 19 - Y36SI3

Zadání: Detailní návrh a vývoj. Použití návrhových vzorů a refaktORIZACE. Kritéria kvality návrhu a jejich analýza. Zlepšování výkonu a udržovatelnosti. Zpětné inženýrství. Disciplinované přístupy ke změnám software.

V materiálech pro předmět SI3 (rok 2008/2009) moc zmínek o těchto tématech nejsou, proto bude níže obecnější popis těchto pojmu a konkrétní otázky si budete muset dát do souvislostí sami.

Návrhové vzory

Velmi často se programátoři setkávají při řešení svých úkolů s opakujícími se problémy, ať se již s nimi setkali oni nebo někdo jiný. Ale dá se říci, že velká část řešeného problému již byla řešená někým jiným. Proto je snaha o **nalezení určitých univerzálních postupů, metodik**, jak otázky určitého druhu řešit. Tyto praktiky pomáhají nejen při tvorbě návrhu a implementace, ale také při následné údržbě, kdy nově příchozí programátor snáze nahlédne do implementačního řešení, když v něm rozpozná nějaký obecný vzor.

Vezmeme-li to konkrétně, tak velká část aplikací potřebuje kupříkladu jednotně řešit přístup k databázi, či k logovací části. Píše-li programátor celou aplikaci sám, bez využití tzv.frameworků, často využívá návrhových vzorů typu Singleton či Factory k řešení těchto úkolů.

Co je tedy takový **návrhový vzor (Design Pattern)**? Je to určitý obecný návrh, jak psát kód řešící danou část problematiky (**pojmenované a popsané řešení typického problému**). Ve většině případů se návrhové vzory týkají **OOP** (Objektově Orientovaného Programování). Návrhový vzor není žádnou hotovou knihovnou nebo implementací v nějakém konkrétním programovacím jazyce. Návrhový vzor je jen popis v obecném algoritmickém zápisu (i pomocí UML diagramů), který lze poměrně snadno převést do konkrétního programovacího jazyka.

Existuje jakýsi katalog základních návrhových vzorů (tzv. **GoF**) - celkem 23 rozdělených do 3 kategorií (dnes existuje další množství vzorů, mnoho z nich spočívá v kombinaci či pozměnění těch 23 základních):

1. *Creational Patterns* (vytvářející)

- Řeší otázky vytváření a předávání referencí objektů v systému. Umožňují ovlivnit způsob vytváření objektů (upřednostňují flexibilní objektovou kompozici namísto „pevné“ dědičnosti.)
- příklady: Singleton, Abstract Factory, Builder, Prototype

1. *Structural Patterns* (strukturální)

- Popisují jak jsou třídy a objekty složeny do větších struktur.
- příklady: Adapter, Decorator, Proxy, Facade, Composite

1. *Behavioral Patterns* (chování)

- Věnují se rozdělení funkčnosti a zodpovědnosti mezi objekty, komunikaci mezi objekty.
- Observer, Command, Iterator, Strategy, Template Method

Příklad návrhové vzoru Singleton

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {
    }
    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
    public void operation() {
        System.out.println("Singleton.operation() executing" );
    }
}

// Pouziti:
object = Singleton.getInstance();
object.operation();
```

Refaktorizace

Refaktorování je termín, který zavedl Martin Fowler v knize Refactoring: Improving the Design of Existing Code. Jde o takové **změny ve zdrojovém kódu programu, které nemění jeho funkčnost**, ale jeho vnitřní strukturu tak, **aby byl lépe čitelný a udržovatelný** pro programátory.

Refaktorování zahrnuje například

- Přejmenování tříd, metod (funkcí), proměnných pro větší srozumitelnost
- Přesouvání metod (funkcí)
- Extrahování části zdrojového kódu jako samostatné metody (funkce)
- Extrahování výrazu jako lokální proměnné
- Zjednodušení podmíněných výrazů
- Přesun společného kódu více tříd do společné nad třídy

Zpětné inženýrství (Reverse Engineering)

Je to označení procesu, ve kterém se programátor snaží **zpětně rozkryt** algoritmus a princip **fungování daného programu** (části programu). Cílem je pochopit danou problematiku a způsob řešení (třeba např. na základě kódu nakreslit class diagramy a pochopit fungování programu).

Často je potřeba provádět zpětné inženýrství na začátku každé nové (či na konci předešlé) iterace, **aby dokumentace odpovídala implementovanému systému** – během implementace jsme například vytvořili nové třídy (např. kvůli zlepšení koheze), které nejsou v návrhu obsaženy – některé nástroje nám např. na základě kódu vygenerují UML diagramy. Nebo u projektů, kde nebyla vytvořena dostatečná dokumentace a je nutné zásahu do programu.

Wiki: Reverzní inženýrství je v informatice definováno jako proces analýzy předmětného systému s cílem identifikovat komponenty systému a jejich vzájemné vazby a/nebo vytvořit reprezentaci systému v jiné

formě nebo na vyšší úrovni abstrakce.

Kritéria kvality návrhu a jejich analýza

Kvalitní návrh by měl využívat tzv. **GRASP vzorů** (soubor obecných principů, které by se měli dodržovat při OO návrhu – zkušený návrhář je používají automaticky a možná ani výraz GRASP neznají) a následně GoF návrhových vzorů (viz. výše - osvědčené praktiky popisující seskupení a chování tříd řešících určité problém) a vzorů architekturních.

GRASP vzory **přiřazují odpovědnosti** SW objektům. GRASP vzorů (principů) se využívá jak v době návrhu, tak v době kódování (viz. zlepšování udržovatelnosti).

2 typy odpovědností:

- *knowing* – objekt zapouzdruje data a na jejich základě poskytuje informace (odvozuje, počítá)
- *doing* – iniciuje, řídí a koordinuje akce jiných objektů (např. Factory – vytváří instance jiných objektů)

Základní GRASP vzory (zvýrazněné jsou podle mně nejlépe srozumitelné/zapamatovatelné):

- **Creator**
 - Řeší, kdo má vytvářet objekt třídy A (jaká třída) (třída B vlastní objekt třídy A, třída B používá objekt třídy A, třída B obsahuje nezbytná inicializační data pro objekt třídy A – čím více je splňeno, tím spíše by měl B vytvářet A)
- **Information Expert**
 - Odpovědnost je přiřazena třídě, která má nejvíce informací.
- **Low coupling**
 - Existuje-li provázanost mezi objekty, pak změna jednoho může ovlivnit všechny svázané atd.
 - Čím menší provázanost, tím lépe (stupeň provázanosti určuje počet vazeb a síla vazby – např. dědičnost je silná vazba)
 - Správné přiřazování odpovědnosti (vzor Information Expert), vede k minimalizaci propojení mezi třídami.
- **High cohesion**
 - Koheze je měřítkem, jak moc spolu souvisí operace třídy – jak moc je třída je soudržná (má správně přiřazenu odpovědnost).
 - Čím větší koheze, tím menší provázanost, tím lepší.
- **Controller**
 - Odpovídá na otázku jak propojit UI vrstvu s aplikační logikou – objekt (či celá vrstva) vykonává systémové operace a zároveň koordinuje spolupráci těchto dvou vrstev.
- **Pokročilé GRASP vzory** (např. Polymorphism – využití polymorfismu a další)

ATAM

Tím, že se vývojáři snaží využívat již ověřených principů (vzorů), mají zajištěnu kvalitu. Je tu však i snaha aby se dalo formálně posoudit kvalitu designu na základě sady pravidel, postupů a metrik - o jejich definici se pokouší práve ATAM. Ideálním cílem takového snažení je nástroj, který zanalyzuje UML model a hledá vzájemné závislosti modulů (loose/tight coupling, separation of concerns - rozdelení software do vrstev které spolu komunikují přes definovaná rozhraní) a jeho výsledkem je nějaké číslo na stupnici „špatně“ až „výborně“. Je to poměrně nová oblast a zatím se spíš vymýšlejí ty metriky než že by se to rutinně používalo, ale už existují softwary které tvrdí, že to umí.

Poměrně nová metoda, která ohodnocuje kvalitu návrhu (na úrovni architektury) podle nejrůznějších kvalitativních kritérií. Příklady kritérií:

- performance (výkon)
- modifiability (přizpůsobitelnost)
- reliability (spolehlivost)
- security (bezpečnost)

Ortogonalita

Moduly projektu musí být maximálně nezávislé. V matematickém světě ortogonální znamená na sebe kolmý - v řeči vektorů znamená dokonce přeneseně nezávislý. Převedeno zpět do řeči softwarového inženýra, ortogonalita představuje druh nezávislosti, vzájemného oddělení. Dvě části jsou na sobě nezávislé, pokud změny v jedné části nás nenutí ke změnám v části jiné. Pokud máme databázi a uživatelské rozhraní, tak můžeme provádět změnu v uživatelském rozhraní, aniž bychom museli měnit databázi a zároveň můžeme přejít na jiný databázový stroj bez změny uživatelského rozhraní v ideálním ortogonálním systému. Ne vždy to samozřejmě funguje, ale myšlenka by měla být jasná: ortogonalita ~~ nezávislost.

Základním testem ortogonality by mohla být změna jednoho požadavku → kolik modulů se musí kvůli tomu upravit? ideálně jeden.

„Separate policy from mechanism“ – abstrakce kódu – detaily, konkrétní hodnoty pro aplikaci v metadatech a konfiguračních souborech (možnost volby)

Zlepšování výkonu a udržovatelnosti

– *Zlepšování udržovatelnosti*

U velkých projektů je dobrá udržovatelnost jednou z klíčových vlastností. Na projektu pracuje mnoho různých lidí a proto porozumění kódu a dokumentace je velice důležitá (určitě se bude do projektu zasahovat).

Zamezení duplikace (stejná data na dvou místech, stejný kód na dvou místech) - **DRY princip** (Don't Repeat Yourself) - pokud se podobný algoritmus využívá na více místech, pak radši předělat do oddělené části (např. metoda, třída). Používat konstanty místo statických hodnot v kódu (mohou se opakovat).

Strukturovat kód

- zlepšení koheze (zaměřenosti) jednotlivých tříd, metod – třída řeší problémy, které spolu sovisí a kterým odpovídá název třídy/metody
- metoda by neměla být delší, než aby se nevešla na obrazovku, třídy by neměly mít obrovské

množství metod

- odstranění nepotřebných vazeb – podle GRASP vzoru low coupling (čím méně vazeb mezi třídami/moduly, tím lépe)
- vhodná pojmenování proměnných, metod, tříd, modulů – lepší pochopení kódu jinými lidmi
- vytváření lokálních proměnných, či metod namísto složitých logických či jiných výrazů
 - `boolean isMarried = theMan.getWifes().size() > 0;`
 - `if (isMarried && hasChildren){ ... }`
- odstraňování hluboko vnořených if–then–else apod.
- *vsuvka*: snažím se, aby můj kód pochopil i ten největší blbec
- **Demeterův zákon** = Metoda objektu smí volat:
 - Jiné metody objektu
 - Metody objektů které vlastní
 - Metody objektů které vytvořila
 - Metody objektů které jsou parametrem metody

Používání návrhových vzorů – lepší rozpoznání a orientace v řešení, které je postaveno na známém návrhovém vzoru (jeho využití je zdokumentované)

Zlepšování dokumentace

- čím kvalitnější dokumentace, tím lépe se projekt udržuje
- využití zpětného inženýrství

Analýza kvality návrhu a implementace - **Code Review** - prověrka kódu jiným programátorem (technickým vedoucím).

– *Zlepšování výkonu*

Statické analyzátory kódu - mohou např. odhalit větve, které se nikdy nevykonají (umí často i samotné IDE).

Statistika Code Coverage - pokrytí kódu - pokud je kvalitně zpracovaná dokumentace a jsou pokryty všechny use case pro daný software, může se použít nástroj, který analyzuje, které řádky kódu byly vykonány při průchodu všemi scénáři. Teoreticky by řádky, které vůbec neproběhly, neměly být přítomny. Code Coverage statistika se hlavně využívá při spouštění unit testů.

Pravidelné spouštění unit testů pomocí nějakého integračního nástroje.

Profilování aplikací - je možné mít spuštěnou aplikaci a měřit, které části kódu potřebují ke svému běhu nejvíce paměti, které spotřebují nejvíce procesorového času. Podle toho se pak mohou dané části kódu optimalizovat.

Disciplinované přístupy ke změnám software

Na tuto otázku může být více pohledů, co je tím vůbec myšleno. Autoři tohoto textu si myslí, že se jedná zejména o software pomocí kterého spravujeme verze software, spravujeme a komunikujeme požadavky na změny software. Jedná se tedy o tzv. **SCM** (Source Control Management) systémy a systémy pro týmovou komunikaci.

SCM = Source Control Management software, základní typologie rozdělení je podle počtu a umístění základního úložiště kódu(=repository) na:

- **Centralizované** - příklad: SVN(Subversion), CVS
- **Distribuované** - příklad: Git, Mercurial

Centralizované můžeme charakterizovat tak, že máme jednu ústřední repository, do které vývojáři nahrávají změny kódu a udržují verze vyvíjeného software. Zatímco distribuované nelze tak snadno charakterizovat - repository existuje v rámci týmu vývojářů vícero a výsledný produkt vzniká často různým spojováním verzí z různých repository nebo je vytvořen strom repository a změny se do vyšších větvích promítají postupně. Domníváme se však, že specifické otázky k tomuto tématu nebudou, neboť ani na přednáškách nebyly rozebírány.

K tématu správa zdrojového kódu ještě nutno uvést názvosloví:

- *checkout* - získání celé aktuální verze z repository
- *update* - získání změn z repository, změny jsou jen mezi lokální kopíí a aktuální v repository
- *commit* - nahrání změn do repository
- *branch* - zkopirování vývojové či jiné větve do nové větve
- *tag* - označení aktuálního stavu větve nějakým unikátním jménem
- *merge* - spojení dvou větví (nahrání změn v jedné větvi do druhé větve)
- *blame(annotate)* - výpis řádků daného souboru, kdy na začátku každého řádku je vidět, kdo ho naposledy měnil

Issuetracking a bugtracking software = nástroje pro záznam změn a chyb v software

Jedná se často o webové aplikace napojené na SCM systém. Umožňují vedení a sledování změn v systému, často i prostřednictvím nich probíhá komunikace se zákazníky, kdy zákazník sám zadá chybu do bugtracking systému a je mu prostřednictvím něho odpovězeno - ve velkých firmách ovšem tato praxe nebývá, tam komunikace probíhá nejčastěji přes helpdesk oddělení.

Jak vlastně ideálně zaznamenat chybu do takového systému? (Podle Joel Spolskyho) Danou chybu je potřeba popsat a hlavně způsob jakým lze chybu reprodukovat(pokud vůbec), počet kroků k reprodukci chyby by měl být co nejmenší.

Známým příkladem issuetracking systému s možností napojení na Subversion repository, vlastní Wiki je volně dostupný TRAC.

Druhý pohled na disciplinované přístupy ke změnám software (podle předmětu ASS)

Změny jak okolí systému, tak uživatelských požadavků jsou nevyhnutelné. Software modeluje část reality a realita se mění, zároveň zákazník sám přesně neví (co vlastně chce, co je technicky možné, co je finančně schůdné). Software se tedy musí vyvíjet a **musí počítat se změnami**. Tvorba SW je činnost s extrémně vysokým stupněm změny:

- typický projekt mění své požadavky z 25%
 - celá 1/4 všech požadavků bude tedy jiná na začátku a na konci
- průměrná změna v zadání je mezi 35–50%
 - Jinými slovy klient si průměrně polovinu toho, co chtěl, rozmyslí, zruší nebo chce jinak.

U většiny projektů je tedy nemožné přesně a úplně zjistit a stabilizovat specifikace zadání a požadavky na projekt před samotným započetím práce na něm. (tzv. vodopádový model)

Nesnažíme se vědět přesně, CO se má udělat, nýbrž, **co se změní a jak** – identifikujeme tzv. variační (místo změny v současném systému) a evoluční (v budoucnu zde pravděpodobně nastane změna) body. Při návrhu (i samotné implementaci) **oddělujeme jednotlivé části a podčásti systému na základě evolučních a variačních bodů**. K oddělení využíváme rozhraní, návrhových vzorů (např. GoF vzor Adapter).

Možné řešení minimalizovat dopad změn je **využít SW iterativně**, (tzn. v rámci jednotlivých malých iterací rozšiřovat SW, každý takovýto release je dostatečně otestovaný a konzultovaný s koncovým zákazníkem - jednotlivé **změny jsou pak snáze zahrnuty do systému** a jsou tudíž levnější).

Odkazy:

- Přednášky Y36SI3(2008/2009) <http://service.felk.cvut.cz/courses/Y36SI3/> [<http://service.felk.cvut.cz/courses/Y36SI3/>]
- Návrhové vzory <http://objekty.vse.cz/Objekty/Vzory> [<http://objekty.vse.cz/Objekty/Vzory>] , http://cs.wikipedia.org/wiki/N%C3%A1vrhov%C3%BD_vzor [http://cs.wikipedia.org/wiki/N%C3%A1vrhov%C3%BD_vzor]
- Refaktoring <http://cs.wikipedia.org/wiki/Refaktorizace> [<http://cs.wikipedia.org/wiki/Refaktorizace>], <http://www.refactoring.com> [<http://www.refactoring.com>]
- Zpětné inženýrství http://cs.wikipedia.org/wiki/Zp%C4%9Btn%C3%A9_in%C5%BEen%C3%BDrstv%C3%AD [http://cs.wikipedia.org/wiki/Zp%C4%9Btn%C3%A9_in%C5%BEen%C3%BDrstv%C3%AD]

Slovníček pojmu

GoF = (Gang Of Four) – skupina čtyř autorů, která napsala významné publikace v oblasti programování a použití návrhových vzorů (Design Patterns: Elements of Reusable Object-Oriented Software)

GRASP = General Responsibility Assignment SW Patterns – Postupy (zásady) správného návrhu.

ATAM = Architecture Tradeoff Analysis Method – Metoda, která se snaží vyhodnotit kvalitu návrhu (architektury) podle nejrůznějších kritérií.

DRY = Don't Repeat Yourself

No Silver Bullet = neexistuje žádná technologie, která by najednou rapidně uspíšila projekt

Dogfooding = vývojář používají aplikaci, kterou vyvíjejí, tím vychytají spoustu chyb a vytvoří lépe uživatelské rozhraní

SCM = Source Control Management - správa zdrojového kódu