Projektovanie databázových systémov - úvod

Zoznámenie sa

Prednášky a cvičenia: Jaroslav Lach

Belastingdienst Centrum voor ICT, Apeldoorn

Service team Fysieke media verwerking

Senior software engineer

Kontakt:

Jaroslav.Lach@upce.cz

0031 55 528 1272

Skype: Jaroslav Lach

Zoznámenie sa II.

Cvičenia a konzultácie: Jan Fikejz

Organizačné záležitosti

- Týždenný kurz
- Prednášky a cvičenia
- Výuka pondelok až piatok
  - 8:00 až 9:00
  - 9:15 až 10:15
  - 10:30 až 11:30
  - 12:30 až 13:30
  - 13:45 až 14:45
  - 15:00 až 16:00
- Semestrálny projekt
- Konzultácie
- Zápočet a skúška

Študijná literatúra: povinná

- Jim Arlow, Ila Neustadt: *UML 2 a unifikovaný proces vývoje aplikací*. Computer Press 2007
- Meilir Page-Jones: Základy objektově orientovaného návrhu v UML. Grada 2001

Motivácia

- Nainštalovaním prvej verzie softwarového produktu u zákazníka sa všetko len začína....
  - Chyby a problémy (hlavne pri prvých verziách)
  - Zmeny a rozšírenia
- Ako písať software tak, aby tento aspekt bol zvládnuteľný
  - Aby sa dali analyzovať problémy
  - S minimálnym rizikom implementovať zmeny a rozšírenia

Študijná literatúra: doporučená

- Kraval: Objektové modelování pomocí UML v praxi, díl 1, PDF e-kniha, leden 2005
- Dokumentácia procesu UP (dostupná na www.eclipse.org/epf)
- http://www.tonymarston.net/php-mysql/database-design.html
- H. Darwen & C.J. Date: Databases, types and the relational model (The third Manifesto)
- C. J. Date: *An Introduction to Database Systems*. Addison Wesley; 8th edition 2003

Študijná literatúra: doporučená II.

- Matiaško,Karol: *Databázové systémy*. - 1. vyd. - Žilina : Žilinská univerzita, 2002
- Whitten, Bentley, Dittman: Systems Analysis & Design Methods, McGrawHill, 6th edition, 2004.
- Ivachiv Pavel; Kraval, Ilja: *Základy komponentní technologie COM.* Computer Press, 1998
- Head first: Design patterns
- Head first: Object-oriented analysis and design
- Head first: Software development

Potrebné predchádzajúce znalosti

- Objektovo orientované programovanie
- Dátové štruktúry a algoritmy
- Databázové systémy: relačný model, relačná algebra, entitno-relačné modelovanie
- Objektová analýza pomocou UP
- Jazyk UML
- Základné pojmy z oblasti projektovania informačných systémov

O čom bude tento predmet

- Ako postaviť aplikáciu, ktorá ukladá svoje dáta do databázy (relačnej či objektovej)
  - Architektúra a technológie
  - Proces modelovania a realizácie
  - Kritériá kvalitného dizajnu aplikácie a databázy
- Vychádzajúc z modelu požiadaviek a analytického modelu v UML

Prehľad súčasných trendov v oblasti IT

- Globalizácia svetovej ekonomiky
- Electronic Commerce
- Bezpečnosť a súkromie
- Collaboration & Partnership
- Knowledge Asset Management
- Continuous Improvement and Total Quality Management
- Business Process Redesign

Electronic Commerce and Business

E-Commerce – nákup a predaj tovaru a služieb cez Internet

E-Business – používanie Internetu na každodenný byznys

Typy e-commerce a e-business

- Marketing firemného image, výrobkov a služieb
- Business-to-consumer (B2C)
- Business-to-business (B2B)

Dôsledky pre vývoj IS

- ... *doplňte*

Knowledge Asset Management

Dáta – fakty o ľuďoch, miestach, udalostiach a veciach, ktoré majú význam pre danú organizáciu .

Informácie – dáta, ktoré sú spracované alebo zorganizované do zmysluplnej formy.

Vedomosti – dáta a informácie, ktoré sú ďalej zorganizované a upravené na základe faktov, právd, názorov, úsudkov, skúsenosti a expertízy prijímateľa

Knowledge Asset Management

- Rozpoznáva kritický význam dát, informácií a vedomostí pre danú organizáciu
- Pýta sa: “Ako má organizácia organizovať a zdieľať svoje vedomosti tak aby získala konkurenčnú výhodu?”
- Snaží sa integrovať dáta a informácie, z ktorých je možné vytvárať a uchovávať vedomosti

Continuous Improvement a Total Quality Management

Business procesy – aktivity, ktoré reagujú na udalosti spojené s činnosťou organizície (napr. objednávka). Business procesy sú činnosti, procedúry a pravidlá potrebné na vykonanie nejakej firemnej aktivity nezávisle od informačnej technológie, ktorá ich automatizuje alebo podporuje.

Continuous process improvement (CPI) – Sústavné monitorovanie business procesov za účelom ich zefektívnenia.

Total quality management (TQM) – komplexný prístup napomáhajúci zvyšovaniu kvality.

Projekty IS vzhľadom k Business procesom

- Business Process Automation
- Business Process Improvement
- Business Process Redesign

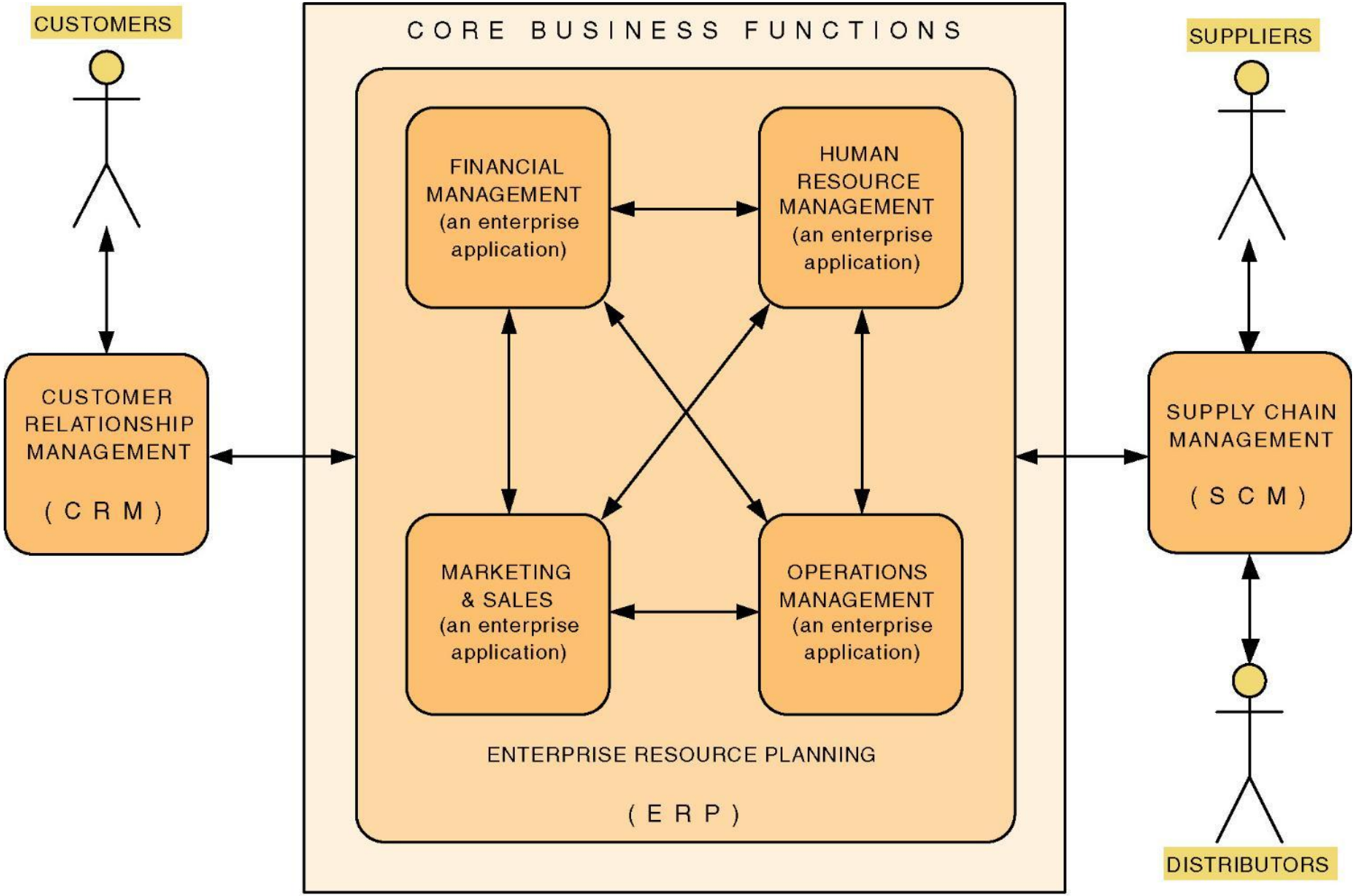
Trendy v rozvoji technológií

- Siete a internet (Web 2.0)
- Mobilné a bezdrôtové technológie
- Objektové technológie
- Kolaboratívne technológie
- Enterprise aplikácie
- Business Inteligence

Business Inteligence

- wikipedia.com: Business intelligence (BI) relates to the intelligence as information valued for its currency and relevance. It is expert information, knowledge and technologies efficient in the management of organizational and individual business. Therefore, in this sense, business intelligence is a broad category of applications and technologies for gathering, providing access to, and analyzing data for the purpose of helping enterprise users make better business decisions. The term implies having a comprehensive knowledge of all of the factors that affect your business. It is imperative that you have an in depth knowledge about factors such as your customers, competitors, business partners, economic environment, and internal operations to make effective and good quality business decisions. Business intelligence enables you to make these kinds of decisions.

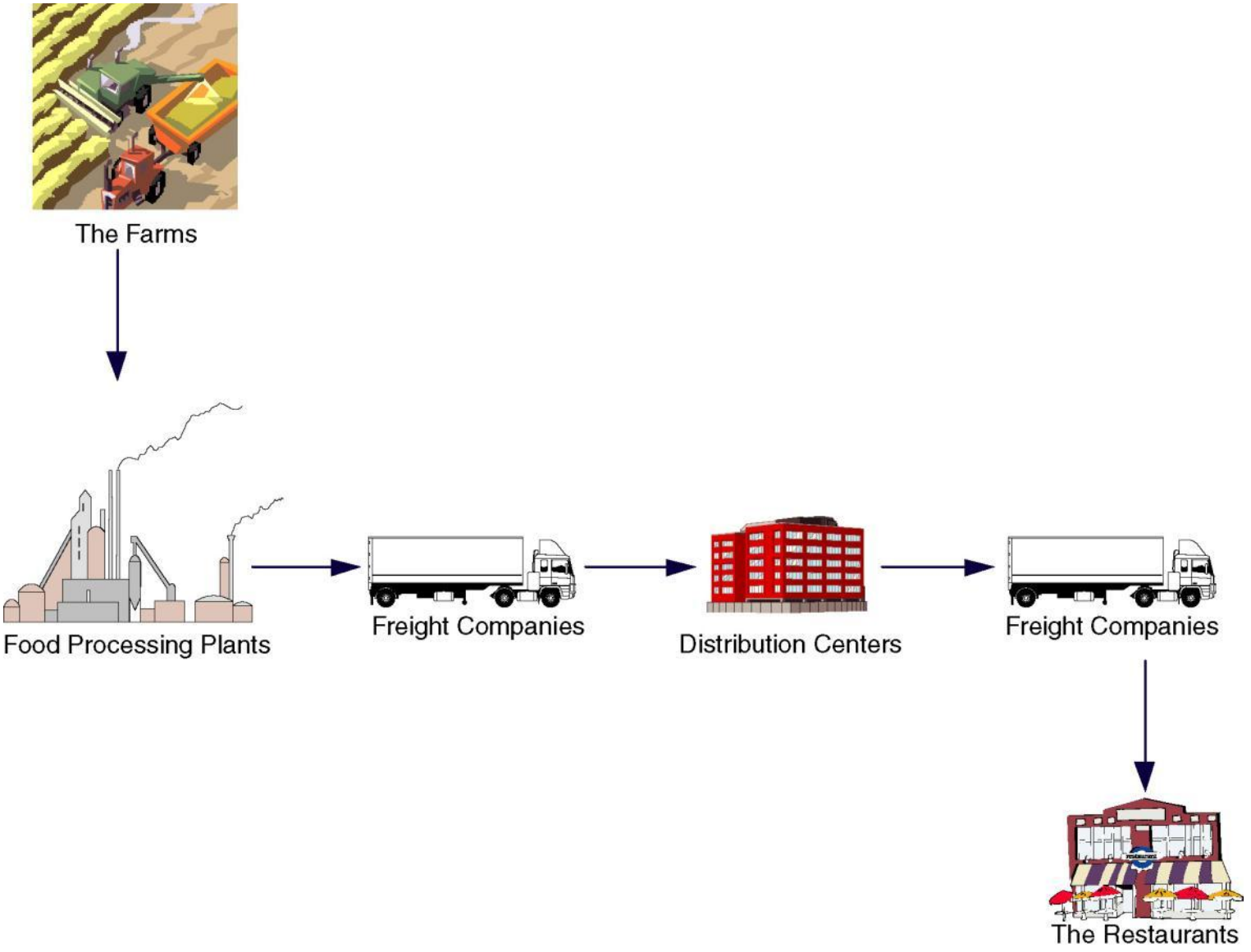
Enterprise aplikácie



ERP

- Softwarové aplikácie, ktoré sa snažia integrovane podporovať všetky hlavné firemné aktivity (core-business)
- SAP
- PeopleSoft
- Baan

Supply Chain Management



EAI a Middleware

Enterprise Application Integration (EAI) – procesy a technológie na prepájanie existujúcich aplikácií a podporu tokov dát a informácií medzi týmito aplikáciami.

Middleware – software používaný na preklad a routovanie dát medzi rôznymi aplikáciami.

- BEA Systems
- IBM (MQSeries, Websphere)
- Mercator Software
- TIBCO Software

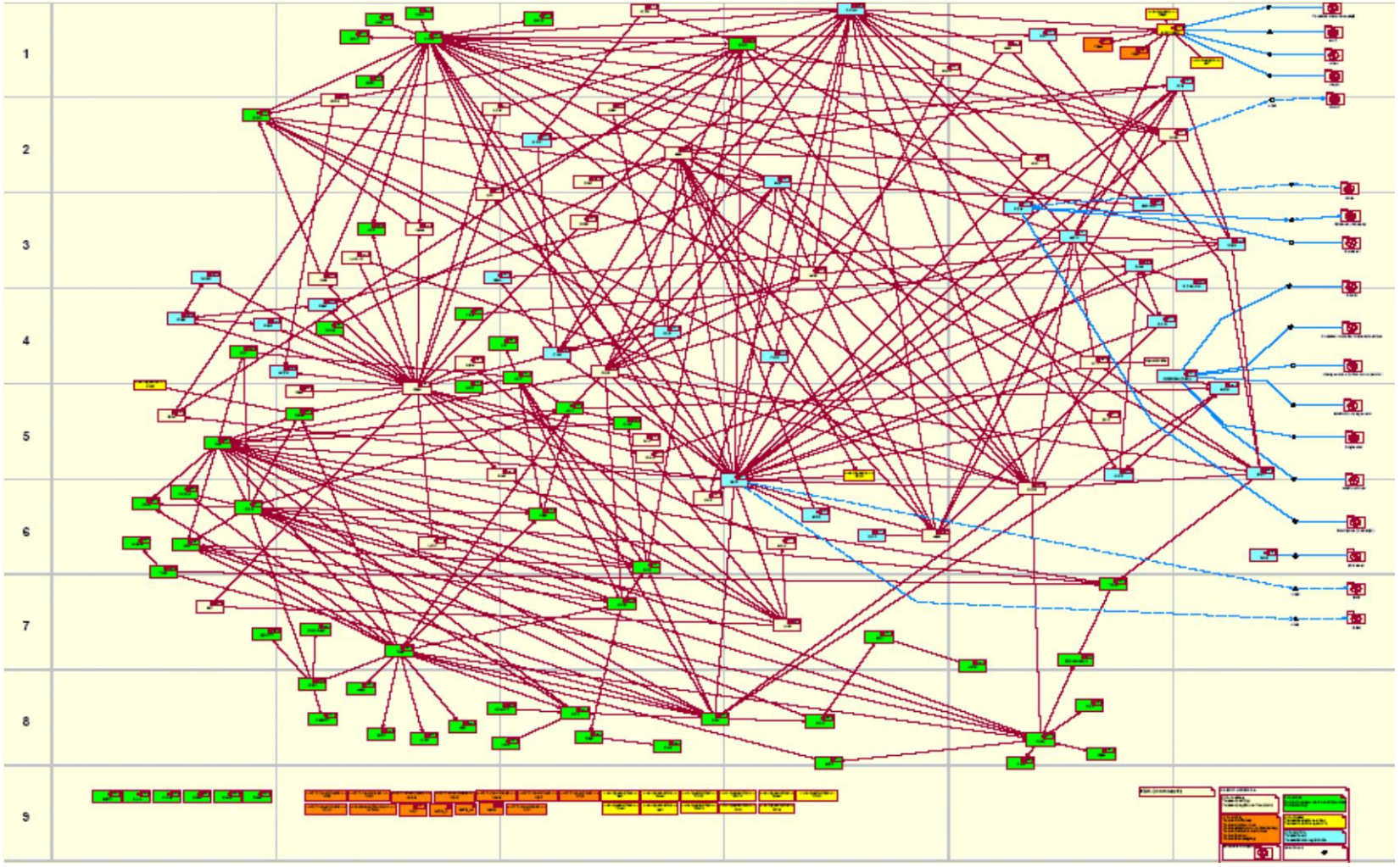
IT v súčasnej Európe

- Veľké množstvo malých neprofesionálnych sw produktov
- Súčasný boom: software robí každý. Samozvaní informatici.
- Dôsledky „internetovej bubliny“ z konca 90tych rokov

Enterprise systémy

- Vznikali postupne ako množiny malých, na vzájom nezávislých systémov
- Každý systém mal vlastné
  - Databázy
  - Štandardy
  - Programovacie techniky
  - Procesy
- Neskôr: prepájanie jednotlivých systémov, wrapovanie, integrácia, štandardizácia
- Iný prípad: prevalcovanie celku systémom ERP

Príklad stavu enterprise systému



Negatívne vplyvy na vývoj IT

- Marketingové stratégie veľkých firiem, ktoré vyvíjajú informačné technológie (Microsoft, Rational, IBM)
  - Snaha vyvíjať a predávať technológie, ktoré kúpi čo najväčší počet zákazníkov (jedna technológia musí podporovať čo najširšiu aplikačnú oblasť)
  - Čo najrýchlejší time to market (byť na trhu prvý)
  - Snaha povyšovať vlastné technológie na štandardy
  - Týka sa to nielen vývojových prostredí ale aj procesov a podporných nástrojov

IT je nedospelé

- Informačné technológie sú relatívne mladé
- V oblasti konštrukcie budov má ľudstvo skúsenosti stovky (tisíce?) rokov. V oblasti IT 50 rokov
- Dôsledok: Nikto dosť dobre nevie, ako na to. Všetci sa veľmi snažíme, ale veľmi nám to nejde

Dôsledky negatívnych vplyvov

- Produkty priemernej kvality
- Obrovské, neprehľadné a nekonzistentné technológie
- Neoverené a nezažité „štandardy“
- Nekvalita víťazí na celej čiare
- Záver: Svet informačných technológií je na tom celkovo dosť zle

Trocha štatistiky

- 2003:
  - 34 percent of IT projects are successful
  - 51 percent challenged (they are over schedule, and/or they are over time, and/or they are missing significant functionality)
  - 15 percent of projects are considered failures

Iná štatistika

- Chaos report (1995):
  - 31,1 % canceled befor completion
  - 52,7 % cost over 189% of the original estimate
  - 16,2 % on time on budget

Ako na to reaguje IT byznys?

- Každú chvíľu sa objaví nový zázračný liek na riešenie všetkých problémov
  - Objektové programovanie
  - SOA
  - RUP
  - … (doplňte sami)
- Pritom riešení, ktoré znamenajú skutočný krok vpred je veľmi málo

Ako sa v tom orientovať?

- Back to the roots:
  - Nesnažte sa za každú cenu byť moderní. Moderná technológia nie je vždy výhrou
  - Veľa dobrých vecí bolo vymyslených už v 60-tych a 70-tych rokoch
  - „Moderné“ technológie sú často kópiami toho, čo sa vymyslelo už dávno
- Back to the basics
  - Mnohé „moderné“ technológie sú zložité a neprehľadné, vychádzajú však z jendoduchého a dobrého nápadu (RUP vs. UP)

Príklady metodológií vývoja IS

- Architected Rapid Application Development (Architected RAD)
- Dynamic Systems Development Methodology (DSDM)
- Joint Application Development (JAD)
- Information Engineering (IE)
- Rapid Application Development (RAD)
- Rational Unified Process (RUP)
- Structured Analysis and Design
- eXtreme Programming (XP)

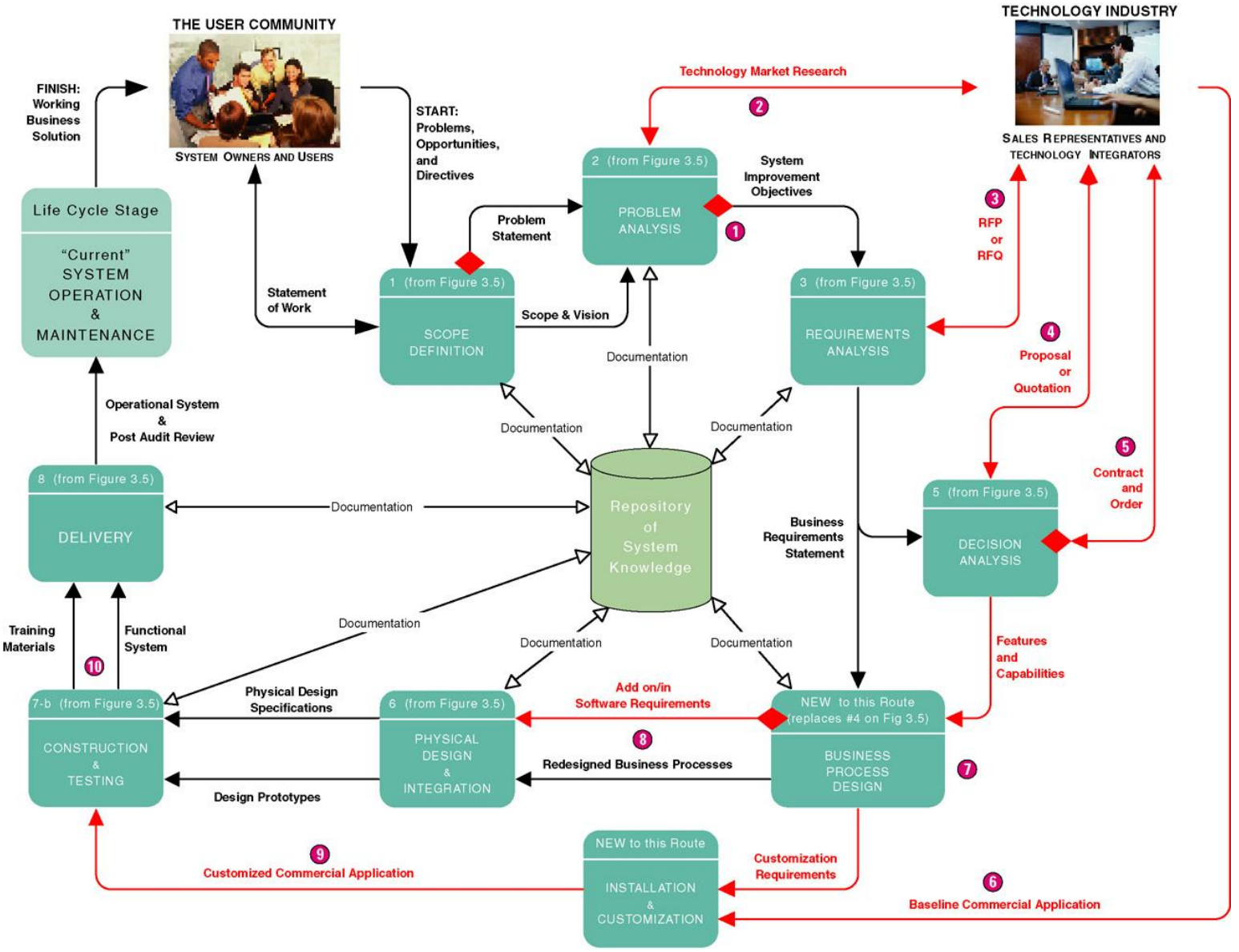
Zásady vývoja IS

- Zapojte užívateľov do vývojového procesu
- Pristupujte k vývoju IS ako k riešeniu problému
- Definujte fázy a aktivity.
- Dokumentujte **počas** vývoja.
- Definujte štandardy.
- Riaďte procesy a projekty
- Vnímajte systémy ako investície kapitálu
- Nebojte sa zrušiť projekt alebo revidovať rozsah projektu
- Divide and conquer (Rozdeľuj a panuj).
- Pri dizajne systémov počítajte s ich rastou a zmenami

Komerčné aplikačné balíky

- Aplikácie, ktoré je možné zakúpiť a prípadne upraviť (nakonfigurovať) tak, aby spĺňali požiadavky danej organizácie. Tiež nazývané *commercial off-the-shelf* (COTS) system.
  - **Request for proposal (RFP)**
  - **Request for quotation (RFQ)**
  - **Gap analysis**

Implementácia komerčného aplikačného balíka



Základné pojmy objektového dizajnu (Opakovanie)

Stručný prehľad pojmov objektového modelovania

- Objekt (inštancia objektu)
- Vnútorná pamäť
- Metóda
- Správa
- Trieda
- Asociácia

Objekt

- UML: A discrete entity with a well-defined boundary and identity that encapsulates state and behavior; an instance of a class
- Všetko je objekt!
  - Toto je ideál objektového sveta. Definícia objektu je postavená tak, že je to v podstate pravda
- Rozoznávame dva druhy objektov:
  - Také, ktoré sa nedajú meniť (celé čísla, napr. 3, 42, reťazce („Mozart“, „Pardubice“) *Reťazec „Mozart“ sa nedá zmeniť, ak ho zmením, už je to iný reťazec*
  - Také, ktoré sa dajú meniť: objekt X, ktorý v danej chvíli obsahuje reťazec „Mozart“

Objekty v tradičnej terminológii

- Objekty, ktoré sa nedajú meniť: hodnoty
- Objekty, ktoré sa dajú meniť: premenné
- Pozor! V niektorých definíciách je termín objekt vyhradený pre objekty, ktoré sa dajú meniť (premenné), pričom pre objekty, ktoré sa nedajú meniť sa (veľmi nesprávne) používa termín „literál“ *Túto definíciu používa napríklad „Object data standard od OMG“*

Zložené objekty

- Objekty môžu mať ľubovoľnú zložitosť
- Môžu obsahovať iné objekty a kolekcie objektov

Trieda (typ objektu)

- Každý objekt je z nejakej triedy
- Trieda v klasickej terminológii: typ
- Trieda teda definuje spoločné vlastnosti nejakej množiny objektov
  - Atribúty (dátová časť)
  - Metódy (správanie)

Trieda a typ

- Podľa niektorých autorov sú trieda a typ rozdielne pojmy
  - Napríklad autori jazyka UML
  - Page Jones
  - Rozdiely však nie sú dobre a jasne definované
- V tomto kurze nebudeme robiť rozdiel medzi pojmami trieda a typ
- Ak budeme potrebovať poukázať na záležitosti modelu alebo implementácie, urobíme tak explicitne

Vnútorná pamäť

- Vnútorná pamäť uchováva stav objektu
- Vnútorná pamäť je zvonku **neprístupná**
  - Vnútorná pamäť pozostáva z atribútov (niekedy aj instance variables)
  - Iný pohľad: Stav objektu (=obsah vnútornej pamäte v danom momente) je vlastne hodnotou (nemeniteľným objektom), ktorú v danej chvíli obsahuje daná objektová premenná

Metódy (operátory)

- tiež procedúry alebo funkcie objektu
- vykonávajú nejakú činnosť nad vnútornou pamäťou objektu **a iba nad ňou**
- Metódy sú zvonku neviditeľné
- Metódy sú viditeľné pre seba navzájom
- Jediná možnosť ako pracovať s pamäťou objektu

Schopnosť prijať správu zvonka

- Protokol správ: mapovanie správa → metóda
- Každej správe zodpovedá práve 1 metóda
- Jediná možnosť ako komunikovať s objektom je poslať mu správu
- Rôzny spôsob implementácie
  - často platí správa=metóda: priame volanie metódy objektu

Metódy a správy v tradičnej terminológii

- Metódy triedy = operátory nejakého typu
- Poslať objektu správu = vykonať nejakú operáciu (operation invocation)
- Dôležitý rozdiel:
  - V klasickej teórii typov nie je operátor naviazaný na nejakú konkrétnu hodnotu alebo premennú
  - V objektovom ponímaní je metóda naviazaná na konkrétny objekt
  - Príklad: A+B: na ktorý objekt je naviazaná operácia „plus“?

Zapuzdrenie (encapsulation)

- Zapuzdrenie znamená, že fyzická reprezentácia objektu nie je viditeľná a prístupná užívateľovi
- Užívateľ vie iba to, aké správy je objekt schopný prijímať
- Kód metód, ktoré správy spracovávajú, má prístup k vnútornej reprezentácii objektu. Tento kód samotný samozrejme tiež nie je užívateľovi prístupný

Nedôslednoť ohľadne pojmu zapuzdrenia

- Princíp: všetko je objekt. Dôsledok: všetko by malo byť zapuzdrené (prístupné iba prostredníctvom metód objektov)
- Avšak:
  - V mnohých prostrediach existujú tzv. public atribúty, t.j. verejne prístupné časti vnútornej reprezentácie objektu
  - Zložené objekty typu kolekcia, pole a.p. (generics) majú priamo prístupné prvky a nespĺňajú preto princíp zapuzdrenia

Fyzická dátová nezávislosť

- Výhodou zapuzdrenia je, že je možné zmeniť vnútornú reprezentáciu objektu bez toho, aby bolo nutné meniť programy, ktoré tieto objekty používajú
- Tomuto princípu hovoríme fyzická dátová nezávislosť

Rozhranie objektu (public interface)

- Rozhranie obsahuje definície metód, ktoré je možné aplikovať na daný objekt
- Definície metód sa nazývajú aj signatúry
- Signatúry sú vždy naviazané na nejakú konkrétnu cieľovú triedu

Identita objektu

- Každý (meniteľný) objekt má jedinečný identifikátor, tzv. OID.
- OID je adresa daného objektu v pamäti

„Priama definícia objektov“ (bez „použitia“ triedy)

- Teoreticky je možné definovať objekt priamo, bez použitia triedy (typov)
- Pre isté typy objektov by to mohlo byť zaujímavé. Napríklad objekt „hlavné okno programu“. Potrebujeme presne jedno, tak prečo by som ho mal vytvárať pomocou triedy?
- Táto črta je síce objektová, nebýva však implementovaná

Polymorfizmus

- Rôzne objekty majú vo svojom protokole správ rovnakú správu, každý objekt však na ňu reaguje inou metódou
- Ak niektorému objektu pošleme danú správu, nezaujíma nás, o ktorý objekt sa jedná

Dedičnosť

- Ak jedna trieda definuje všetky vlastnosti objektov rovnako ako iná trieda a naviac zavádza nové vlastnosti, hovoríme o dedičnosti
- Generalizácia a špecializácia

Class extent

- Všetky objekty danej triedy, ktoré sa v danej chvíli vyskytujú... kde?
  - V priestore danej aplikácie?
  - V danom systéme?
  - V danom namespace?
- Na túto otázku nedáva nikto explicitnú odpoveď, väčšinou sa myslí „priestor“ danej aplikácie, t.j. všetky objekty danej triedy, ktoré spravuje daná aplikácia

Základné pojmy z oblasti databázových systémov

Motivácia

- V dnešnej dobe sa pojmom databáza označuje všetko možné
- Databáza znamená v podstate “báza dát”: t.j. dáta, ktoré používa nejaká aplikácia
- Na druhej strane majú pojmy databáza a databázový systém pomerne presný význam, ktorý implikuje isté vlastnosti a má isté dôsledky
- Ak sa chceme baviť o budúcnosti databázových systémov je dôležité mať definovaný presný káder pojmov

Databázový systém

- Databázový systém je v podstate počítačový systém pre správu údajov *(computerized record-keeping system)*
- Databáza sa dá vnímať ako elektronická kartotéka (kontajner kolekcie dátových súborov)
- Užívatelia systému môžu vykonávať rôzne operácie:
  - Vytvárať nové súbory v databáze
  - Vkladať, vyhľadávať, meniť a vymazávať v nich údaje
  - Rušiť súbory

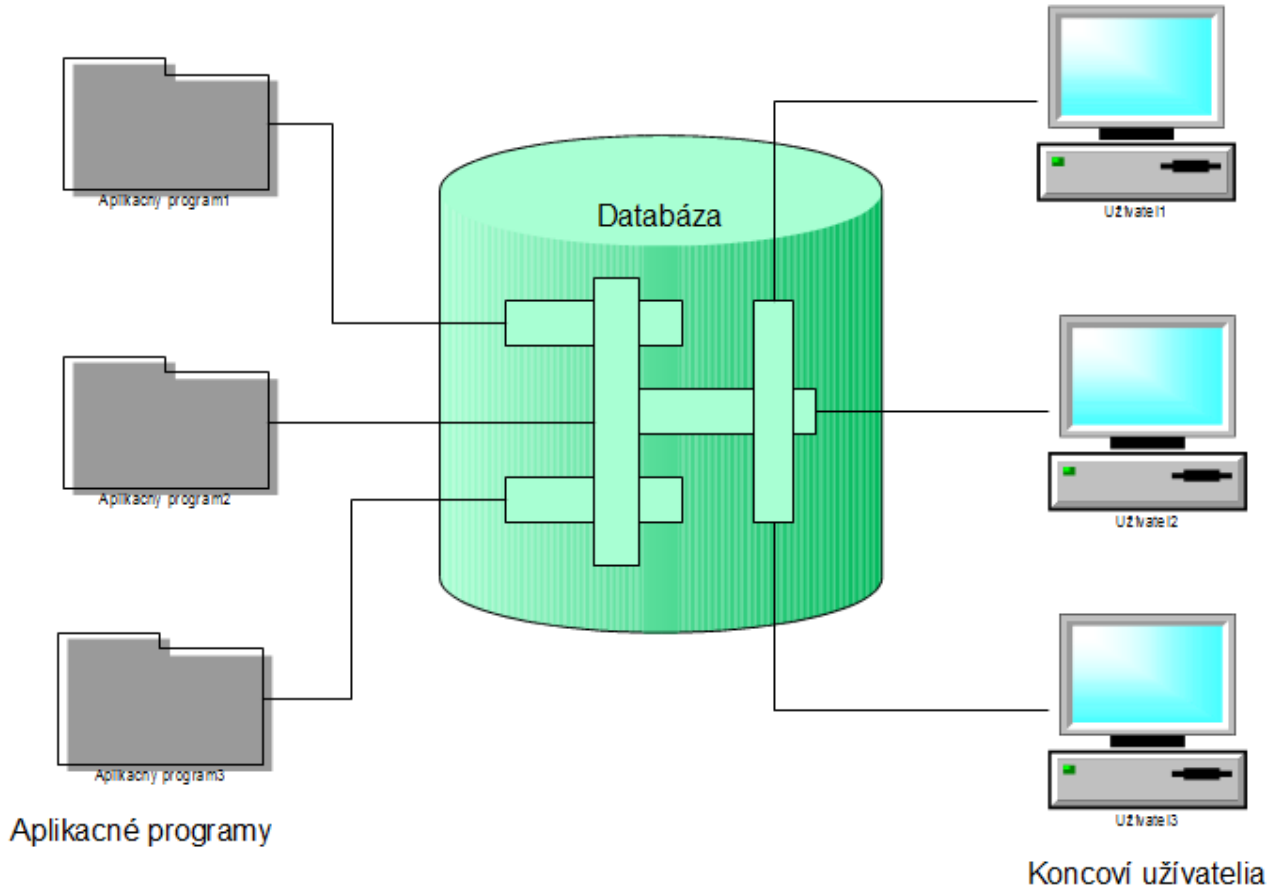
Databáza “Vinotéka”

- Táto databáza obsahuje práve jeden súbor “Vinotéka”, ktorý obsahuje údaje týkajúce sa obsahu vinotéky

BIN#	Víno	Výrobca	Rok	Fľaše	Zrelé
2	Chardonnay	Buena Vista	2008	1	2010
3	Chardonnay	Geyser Peak	2007	5	2011
6	Chardonnay	Simi	2002	4	2010
12	Joh. Riesling	Jekel	2008	1	2012
21	Fume Blanc	Ch. St. Jean	2008	4	2010
22	Fume Blanc	Robt. Mondavi	2007	2	2010

Databázový systém: komponenty

- Databázový systém sa skladá zo 4 základných komponentov:
  - Dáta
  - Hardware
  - Software
  - Užívatelia



Jedno- a viacužívateľské systémy

- V závislosti od možností počítača a druhu aplikácie sú systémy jedno- alebo viacužívateľské
- Vo viacužívateľských systémoch pristupujú k danej databáze viacerí užívatelia naraz
- Ak hovoríme o databázových systémoch, myslíme (defaultne) viacužívateľské systémy
- Tento rozdiel je z pohľadu užívateľa irelevantný: viacužívateľský systém sa užívateľoví javí ako jednoužívateľský
- Špecifické problémy týkajúce sa prístupu viacerých užívateľov sú vnútornými problémami systému

Dáta

- Dáta v databáze sú *integrované* a *zdieľané*
- Tieto dva aspekty predstavujú veľkú výhodu databáz
- Integrované dáta:
  - databázu môžeme vnímať ako zjednotenie rôznych dátových súborov, pričom redundancia medzi týmito súbormi je eliminovaná
- Zdieľané dáta:
  - údaje v databáze môžu byť zdieľané viacerými užívateľmi

- Iný aspekt:
  - Každého užívateľa bude zaujímať malá časť údajov v databáze
  - Časti, ktoré zaujímajú rôznych užívateľov, sa budú prelínať
  - Inými slovami: každému užívateľovi sa daná databáza bude javiť inak

Hardware

- Najdôležitejšie časti:
  - Vonkajšia pamäť (disky), na ktoré sa dáta ukladajú
  - Procesory a vnútorná pamäť pre vykonávanie operácií
- Hardwarom sa ďalej nebudeme zaoberať

Software

- Medzi fyzickou databázou a užívateľmi sa nachádza softwarová úroveň, ktorej sa hovorí:
  - database manager
  - database server
  - **database management system**
- DBMS vybavuje všetky požiadavky pre prístup k databáze (pridávanie a vyhľadávanie dát, …)
- DBMS skrýva pred užívateľmi detaily hardwaru a implementácie databázy

DBMS

- Poskytuje užívateľovi pohľad na databázu na istej úrovni abstrakcie: užívateľ nevidí hardwarové a implementačné detaily
- Príkladom takejto abstrakcie je relačný dátový model a jazyk SQL
- Pojem DBMS často označuje nejaký konkrétny produkt: DB2 od IBM, Sybase Server 11, Oracle, Microsoft SQL Server, …
- DBMS *instance* potom označuje nejakú konkrétnu inštaláciu tohto produktu na nejakom konkrétnom počítačovom systéme

DBMS a databáza

- Pojmy DBMS a databáza sa v bežnom živote často zamieňajú. Predovšetkým keď sa hovorí o “databáze”, myslí sa často DBMS

Užívatelia

- Aplikačný programátori
  - píšu kód v nejakom jazyku. Tento kód pristupuje k databáze (prostredníctvom DBMS)
- Koncoví užívatelia
  - pristupujú k databáze interaktívne prostredníctvom nejakej aplikácie. Touto aplikáciou môže byť aj jednoduchý *query language processor*
- Databázoví administrátori
  - sú zodpovední za správu a údržbu databázového systému (optimalizácia, prideľovanie práv, správa objektov a užívateľov, zálohovanie,...)

Databázy: použitie

- On-line spracovanie transakcií
- Podpora rozhodovania
- Data Warehousing
- ...

Dáta

- Tiež: údaje, data (engl.), gegevens (nl)
- Koreň slova vo všektých jazykoch je: *“dat”*.
- To znamená: niečo, čo je **dané:** Dané **fakty**
- Faktom zodpovedajú v logike pravdivé výroky

Krajina	Hlavné mesto
Česká republika	Praha
Slovenská republika	Bratislava
Poľská republika	Budapešť

🡐Praha je hlavným mestom Českej republiky
🡐Bratislava je hlavným mestom Slovenskej republiky
🡐Budapešť je hlavným mestom Poľskej republiky

- Databázu môžeme vnímať ako kolekciu pravdivých výrokov
- The golden rule: Žiadna update operácia na databáze nesmie spôsobiť vznik nepravdivého výroku

Dátové modely

- DBMS, ktoré v súčasnosti dominujú na trhu, sú relačné DBMS postavené okolo jazyka SQL
- Relačné DBMS poskytujú užívateľovi jednoduchú abstrakciu, v ktorej sú dáta (=výroky) v databáze reprezentované pomocou riadkov v tabuľkách
- Tejto abstrakcii sa hovorí *relačný dátový model*
- Vedľa relačného modelu existujú aj iné modely dát: objektový, hierarchický, sieťový, ...

Dátový model

- Všeobecná definícia: Dátový model je množina abstraktných objektov a operátorov, ktoré s týmito objektmi pracujú. Tieto objekty a operátory spoločne predstavujú abstrakciu (pojmy), s ktorou pracuje užívateľ. Pomocou objektov užívateľ definuje dátové štruktúry. Operátory umožňujú modelovať správanie.
- Implementácia dátového modelu: Nejaká konkrétna fyzická realizácia daného abstraktného modelu

Dátový model príklad

- Relačný dátový model (abstraktný):
  - Objektmi relačného modelu sú relácie, pohľady, dátové typy
  - Operátormi sú napríklad operátory relačnej algebry
- Implementáciou relačného modelu sú rôzne prostredia na báze SQL.
  - Napríklad Sybase
  - Niektoré operátory relačnej algebry sú definované SQL priamo (projekcia), niektoré vôbec (delenie)

Logický a fyzický pohľad

- Logický: princípy, všeobecné pravidlá, veci tak, ako BY MALI BYŤ
- Fyzický: konkrétna implementácia týchto logických princípov, to ako sa veci naozaj majú (v tejto konkrétnej implementácii!)
- Príklad: Relačný dátový model a jeho implementácia v MS SQL
- Iný príklad: UML tak ako je definovaný OMG a jeho implementácia v Eclipse

Model a jeho implementácia

- Dátový model je to, čo by užívatelia mali poznať, jeho implementácia je to, čo by poznať nemali
- Mnohé systémy v súčasnosti tento rozdiel nezdôrazňujú dostatočne (alebo vôbec)
- Často sa dohromady miešajú záležitosti principiálne (model) a záležitosti praxe
- Nás budú zaujímať predovšetkým princípy

Iný význam pojmu dátový model

- Model perzistentných dát nejakého podniku
- V prvom význame: relačný model, objektový model, hierarchický model
- V druhom význame: dátový model Univerzity Pardubice
- Paralela:
  - Prvý význam: nejaký programovací jazyk
  - Druhý význam: program napísaný v tomto jazyku
- Príklad: Relačný model Univerzity Pardubice

Dátový model: význam

- Dobrá abstraktná definícia dátového modelu je dôležitá, lebo:
  - Poskytuje základ pre komunikáciu (o čom to vlastne hovoríme?)
  - Umožňuje vytvárať kvalitné konkrétne implementácie
- Ale aj:
  - Presne si definovať doménu v ktorej pracujeme je kľúčovou úlohou systémového dizajnu
  - Inak nemôžeme staviať systémy dobrej kvality
- Ide teda o **princíp**

Teória dátových typov

- Každý abstraktný dátový model nejakým spôsobom vychádza z teórie dátových typov (jeden zo základných princípov informatiky)
- Opakovanie: čo sú základné pojmy teórie dátových typov (a informatiky vôbec)?

Výhody databáz I.

- Zdieľanie dát
- Obmedzenie redundancie
- Možnosť vyhnúť sa inkonzistenciám (do istej miery)

–Dôsledok obmedzenia redundancie. Tým, že žiadna informácia sa v databáze nenachádza dvakrát, sa zníži riziko, že by dve inštancie obsahovali rozličné informácie

- Podpora transakcií
- Možnosť udržiavať integritu
- Možnosť zabezpečenia dát (security)

Výhody databáz II.

- Možnosť vyvážiť konfliktné požiadavky
  - DBA má možnosť optimalizovať databázu tak, aby optimálne spĺňala požiadavky čo najväčšieho počtu užívateľov. To na rozdiel od maximalizácie požiadaviek z pohľadu jednej aplikácie
- Možnosť aplikovať a štandardy
  - napr. podnikové štandardy pre formátovanie a pomenovanie dát

Dátová nezávislosť

- Fyzická: aplikácie nemôžu zmeniť fyzickú reprezentáciu dát a prístupové techniky k týmto dátam.
  - Silne implementované v RDBMS, slabo v OO
- Logická: dáta sú zorganizovné nezávisle od toho, aké aplikácie ich budú používať
  - Objekty reálneho sveta majú svojú štruktúru, ktorú môžeme popísať nezávisle od toho, čo s týmito objektmi budeme robiť

Fyzická dátová nezávislosť

- Aplikácie postavené v starších vývojových prostrediach (ale aj v objektových prostrediach) majú tendenciu byť dátovo závislé
- To znamená, že:
  - Aplikácia určuje požiadavky na spôsob uloženia dát vo vonkajšej pamäti a prístupové techniky k týmto dátam
  - Znalosti o spôsobe uloženia dát a prístupových technikách sú zakódované do aplikácie samotnej

Príklad dátovo závislej aplikácie

- Nech daná aplikácia používa dátový súbor ZAMESTNANEC. Nech z dôvodu optimalizácie má tento súbor index organizovaný podľa atribútu ZAMESNANEC_ MENO
- V starších systémoch bude daná aplikácia vedieť o existencii tohoto indexu a poradí záznamov, ktoré tento index definuje. Túto znalosť bude aplikácia využívať

- Aplikácia je dátovo závislá vtedy, keď nie je možné zmeniť fyzickú reprezentáciu dát a prístupové techniky k ním bez toho, aby sme zasiahli do aplikácie
- Nebude možné napríklad zmeniť existujúci index alebo začať používať nový index
- Kód, ktorý bude potrebné zmeniť, je kód týkajúci sa prístupových techník k dátam a nie logiky aplikácie samotnej!
- Ak teda prístupové techniky a fyzickú reprezentáciu izolujeme, vyhneme sa nutnosti meniť aplikáciu

Dátová (ne)závislosť a databázy

- Dátová závislosť je v databázových prostrediach mimoriadne nežiadúca, pretože:
  - Rôzne aplikácie potrebujú rôzne pohľady na dáta. Ak by fyzická reprezentácia a prístupové techniky boli určované aplikáciou A, mohli by byť dáta pre aplikáciu B nečitateľné alebo prístup k nim by mohol byť neefektívny (uvidíme pri OODB)
  - DBA musí mať možnosť meniť štruktúru dát (napr. pridávať atribúty) a optimalizovať túto štruktúru (vytvárať indexy) bez toho, aby bolo nutné meniť všetky existujúce aplikácie
- **Jedinou aplikáciou v databázových systémoch, ktorá má znalosti o štruktúre dát a**

Fyzická dátová nezávislosť: definícia

- Fyzická dátová nezávislosť je imunita aplikácie voči zmenám
  - fyzickej reprezentácie dát a
  - prístupových techník k týmto dátam

Požiadavky na DBMS

- Aby sa nejaký softwarový systém mohol nazývať databázovým systémom, budeme od neho vyžadovať splnenie nasledujúcich požiadaviek:
  - Dátová nezávislosť
  - Abstraktný dátový model
  - Dotazovací jazyk
  - Ad hoc dotazovanie
  - Pohľady
  - Integritné obmedzenia
  - Katalóg
  - Schopnosť zotaviť sa v prípade havárie systému
  - Konkurenčný prístup
  - Optimalizácia
  - Bezpečnosť

Abstraktný dátový model

- Sú (presne) definované abstraktné objekty, s ktorými môžeme v danom modeli pracovať
  - V relačnom dátovom modeli sú to relácie
- Sú definované základné operácie, ktoré môžeme s týmito objektmi vykonávať
  - Relačná algebra
- Dátový model je uzavretý (všetky operácie produkujú iba objekty, ktoré sú súčasťou modelu)
- Podporuje zložité dáta, typy a dedičnosť

(Dotazovací) jazyk

- Existuje (abstraktný) jazyk, ktorý umožňuje pracovať s objektami dátového modelu
  - definovať typy (konkrétnych) objektov a operácie na nich (typ Zamestnanec)
  - deklarovať premenné týchto typov
  - vykonávať operácie s nimi
- Príklad: TutorialD, SQL, Linq

Ad hoc dotazovanie

- Existuje nástroj, v ktorom je možné veľmi jednoducho a rýchlo vytvárať jednoduché jazykové konštrukcie (dotazy) predovšetkým za účelom výberu dát z databázy
- Príklad: Query Analyzer pre MS SQLServer, Tood pre Oracle databázy, QueryTool pre Postgres,...

Pohľady

- Možnosť vytvárať virtuálne objekty v databáze
- Tieto virtuálne objekty predstavujú pohľady na danú databázu
- Príklad: Views v jazyku SQL

```
CREATE VIEW hiredate_view
AS
SELECT c.FirstName, c.LastName, e.EmployeeID, e.HireDate
FROM HumanResources.Employee e JOIN Person.Contact c on
  e.ContactID = c.ContactID
```

- S virtuálnymi objektami musí byť možné pracovať rovnako ako so skutočnými objektami.
- Pre užívateľa musí byť transparentné, či pracuje s pohľadom alebo skutočným objektom

Integritné obmedzenia

- Systém musí poskytovať možnosť definovať integritné obmedzenia:
  - typové obmedzenia
  - referenčná integrite

Katalóg

- Katalóg je časť databázy, ktorá obsahuje metainformácie o tejto databáze
  - názvy a vlastnosti objektov
  - mená užívateľov a ich prístupové práva
- Objekty katalógu by mali byť prístupné rovnako ako užívateľské objekty, t.j. pomocou objektov z príslušného abstraktného modelu (napr. ako relácie)

Schopnosť zotaviť sa v prípade havárie systému

- Systém by mal byť schopný zotaviť sa v prípade havárie
- Transakčný prístup
- Zálohovanie

Konkurenčný prístup

- Systém musí byť schopný korektným spôsobom obsluhovať konkurenčné požiadavky viacerých užívateľov
  - Bezpečnosť
  - Uzamykanie

Optimalizácia

- Systém by mal poznať (automatické) mechanizmy pre optimalizáciu dotazov. Na základe znalostí štruktúr dát a prístupových techník k nim by mal byť schopný zorganizovať vyhodnocovanie dotazu tak, aby prebehlo čo najefektívnejšie

Bezpečnosť

- Systém musí poskytovať možnosť definovať rôzne prístupové k jednotlivým objektom databázy

Súčasné databázové systémy

- Na trhu dnes jednoznačne dominujú relačné DBMS
  - sú postavené na dobre definovanom abstraktnom modeli
- Väčšina ich je postavená na jazyku SQL.
  - *SQL nie je úplnou (dokonca ani veľmi dobrou) implementáciou relačného modelu*
- SQL databázy spĺňajú väčšinu uvedených požiadaviek

Postrelačné databázové systémy

- V súčasnosti vznikajú nové druhy databázových systémov, ktoré sa snažia o splnenie uvedených požiadaviek. Tým sa hovorí “postrelačné” databázy
- Medzi postrelačné databázy sa dá počítať aj dobrá implementácia relačného modelu
- Ďalej sem patria objektové a objektovo-relačné databázy
- Niekedy sa hovorí aj o XML databázach

Objektové databázy

- Postavené na objektovom “modeli”.
  - Vzhľadom na to, že neexistuje obecne platná a akceptovaná definícia objektového modelu, je základ objektových databáz nie príliš pevný
- Implementácie:
  - db4o
  - ObjectStore
  - ...

Objektovo-relačné databázy

- Objektovo relačné databázy vychádzajú z relačného modelu, ktorý “rozširujú” o objektové črty
- Existujú napríklad “objektové rozšírenia” jazyka SQL
- Inou alternatívou objektovo-relačných databáz je dobrá implementácia relačného modelu
- Relačný model sám o sebe totiž obsahuje “objektové” črty

Definície abstraktných dátových modelov

- Relačný model bol definovaný už v 70. roku 1970 v článku T. Codda „A Relational Model of Data for Large **Shared** Data Banks“
- Objektovo relačné modely ako aj objektové modely sú definované pomocou rôznych „manifestov“
- O definíciu objektového modelu sa pokúša o.i. OMG
- De facto štandardnou definíciou je v súčasnosti definícia podľa jazyka UML

Objektový model

- V súčasnosti neexistuje všeobecne akceptovaná formálna definícia objektového modelu
- Existujú rôzne snahy takúto definíciu vypracovať
- OMG (Object Management Group) je pritom jedným z centrálnych hráčov
- OMG je konzorcium firiem, univerzít a iných inštitúcií, ktoré má za cieľ vypracovanie a presadenie rôznych objektových štandardov

- Typickým príkladom činnosti OMG je presadenie jazyka UML ako štandardu pre objektové modelovanie
- V oblasti dát existovala podskupina OMG, ktorá sa volala Object Data Management Group. Výsledkom jej práce je dokument “The Object Data Standard”. Tento dokument mal definovať abstraktný objektový dátový model
- Tento model sa však nepresadil a skupina bola pred pár rokmi rozpustená
- Najnovšou iniciatívou je Object Database Technology Working Group

- Táto skupina má tiež za cieľ poskytnúť definíciu objektového modelu, tentokrát postavenú na nejakom matematickom formalizme
- Táto definícia bude vychádzať z práce profesora Kazimierza Subietu z Poľsko-japonského inštitútu pre informačné technológie
- Profesor Subieta vypracoval model, ktorému hovorí Stack-Based Architecture (SBA)
- V tomto modeli zavádza istý formalizmus, ktorý má reprezentovať objektový model:
  - prvý dojem: absolútne neintuitívne

Objektový model podľa ODMG

- Základné pojmy sú *objekt* a *literál*.
  - Objekt má id, literál id nemá
- *Objekty* a *literály* sú kategorizované podľa *typov*
- Všetky prvky daného *typu* majú spoločný *stavový priestor* (=spoločné *vlastnosti*) a spoločné *správanie*
- *Stav objektu* je definovaný hodnotami množiny *vlastností* daného *objektu*. Týmito *vlastnosťami* môžu byť *atribúty objektu* samotného alebo *odkazy* na iné *objekty* (*vzťahy* s inými *objektmi*)
- Hodnoty *vlastností objektu* sa v čase menia

- *Správanie objektu* je definované množinou *operácií*, ktoré je s týmto *objektom* možné vykonávať
- *Operácie* môžu mať množinu vstupných a výstupných *parametrov*. Každý *parameter* je definovaného *typu*. Každá *operácia* môže vraciať výsledok nejakého *typu*
- *ODMS* skladuje *objekty* a umožňuje ich zdieľanie rôznymi užívateľmi a aplikáciami
- *ODMS* je založený na *schéme*. *Schéma* je definovaná pomocou *ODL* a obsahuje definície *typov objektov*, ktoré môže daný *ODMS* obsahovať

Objektový model podľa UML

- UML je rozsiahly jazyk umožňujúci modelovať rôzne aspekty softwarových systémov
- Podmnožina jeho pojmov (static view) je určená na modelovanie dát
- Základnými pojmami tejto podmnožiny sú *klasifikátor* a *asociácia*
- *Klasifikátor* je prvok modelu, ktorý popisuje veci obsahujúce hodnoty
- Existuje viacero druhov *klasifikátorov*, medzi nich patria: *triedy, rozhrania* a *dátové typy*

- Veci, týkajúce sa správania, patria tiež medzi klasifikátory. Sem patria *prípady užitia* a *signály*
- *Trieda* popisuje množinu *objektov*, ktoré majú rovnaké *atribúty, operácie, metódy, vzťahy* a *správanie*
- *Objekt* je diskrétna entita s jasne definovanými hranicami a identitou, ktorá zapuzdruje *správanie*. Je to inštancia *triedy*.
- *Atribút* je popis pomenovaného prvku špecifikovaného typu v *triede.* Každý objekt triedy obsahuje oddelene hodnotu daného typu

- *Operácia* je špecifikácia transformácie alebo dotazu, vykonanie ktorého môžeme od daného objektu žiadať
- *Metóda* je implementácia operácie
- *Správanie* je špecifikácia toho, ako sa mení stav daného objektu v čase.

Manifesty

- Diskusie o smer vývoja databázových systémov sa odohrávajú vo forme manifestov
- Existujú tri významné manifesty:
  - OO manifest
  - OR manifest
  - The third manifesto

OO manifesto

- Atkinson a kol. The Object Oriendet Database System Manifesto
- Je rozdelený na
  - Mandatory features: the golden rules (13)
  - Optional geatures: the goodies (5)
  - Open choices (4)
- Začiatkom 90 rokov sa predpokladalo, že objektové databázy nahradia relačné databázy. Nedošlo k tomu. V súčasnosti objektový svet priznáva, že relačné databázy nenahradia a snažia sa o koexistenciu

OR manifesto

- alebo aj “The Third-Generation Database Systems Manifesto”
- Iniciatíva skupiny “*The Committee for Advanced DBMS Function”*
- Táto skupina js zložená okolo ľudí z univerzity Berkley, pár ďalších univerzít a firiem Digital Equipment a Oracle
- Manifest vychádza z relačného modelu, ktorý rozširuje o niektoré objektové črty

- Manifest prezentuje 3 základné princípy a 13 doporučení
- Základnými princípmy sú
  - Požiadavka na podporu zložitých dátových štruktúr
  - Požiadavka na úplnú podporu relačného modelu (ktorý sa označuje ako databázy druhej generácie)
  - Požiadavka otvorenosti voči iným subsystémom (napríklad programovacím jazykom)

Objektový model podľa OO manifestu

- OODBMS musí podporovať zložené objekty
  - Typickými príkladmi zložených objektov sú pole, zoznam, collection, množina,..
- OODBMS musí podporovať objektové identifikátory
- Zapuzdrenie
- Typy a triedy
  - Typ sumarizuje spoločné vlastnosti nejakej množiny objektov
  - Trieda je niečo iné, avšak voľba použitia týchto termínov je prenechaná užívateľovi

- Typ má rozhranie a implementáciu, pričom iba rozhranie je viditeľné pre užívateľov
- Rozhranie pozostáva z množiny operácií a ich signatúr
- Musia byť podporované typové hierarchie
- Systém musí podporovať late binding
- Systém musí byť výpočtovo kompletný
- Systém musí byť rozšíriteľný
- ...*a musí mať ďalšie vlastnosti databázovej povahy*

The Third Manifesto (TTM)

- TTM je iniciatívou skupiny okolo C. Datea. Chris Date je spoluautorom relačného modelu
- “Third” preto, lebo vyšiel ako tretí v poradí
- TTM vychádza z relačného modelu avšak vyžaduje jeho konsekventnú implementáciu v spojení s dobrou implementáciou teórie typov
- Táto kombinácia umožňuje odstrániť nedostatky, ktoré sa vytýkajú súčasným SQL databázam a splniť požiadavky na modelovanie “zložitých dátových štruktúr”

- TTM má výhodu veľkej tradície a silného zázemia SQL databáz
- To, že táto tradícia a zázemie existujú, nie je náhoda. Je to spôsobené tým, že základné princípy sú nielen
  - dobre formulované a
  - postavené na formálnom modeli

  ale sú aj
  - **intuitívne** a ľahko pochopiteľné
  - využívajú **prirodzený** spôsob organizovania informácií a vedomostí vo forme tabuliek

OODBMS db4o

Úvod

- db4o (database for object) je typickým predstaviteľom objektovo-orientovaných DBMS
- Ukážeme si jeho základné črty
- Vytvoríme si jednoduchú aplikáciu
- Porovnáme naše riešenie s relačným riešením
- Ukážeme si niektoré typické spôsoby argumentácie a uvažovania tvorcov objektových databáz

db4o: Základné informácie

- Open source databáza
- Vytvorená pre Javu a .NET
- Ako najväčšia výhoda sa prezentuje to, že nie je potrebné OR mapovanie
- Údajne je 55 rýchlejšia než Hibernate či MySQL
- db4o je “full-featured” DBMS
- je “embeddable” (dbms sa môže zabudovať do rôznych aplikácií)
- 45000 registrovaných užívateľov a 2,5 milióna downloadov

One-Line-of-Code Database

Typický príklad použitia:

```
public void store(Car car)
{
    ObjectContainer db = Db4o.openFile("car.yap");
    db.store(car);
    db.commit();
    db.close();
}
```

Jedná sa pochopiteľne o veľmi jednoduchý a rýchly spôsob ako naprogramovať uloženie perzistentného objektu

Použiteľnosť

- Db4o je primárne určená pre jednoužívateľské stand-alone aplikácie (pre PDA, telefóny)
- Pozná však aj client/server mód
- Poskytuje replikačný mechanizmus, ktorý umožňuje synchronizáciu medzi
  - rôznymi db4o databázami
  - db4o databázami a a relačnými databázami (!)

Native queries

- Od verzie 5 poskytuje db4o možnosť dotazovania pomocou Native queries mechanizmu
  - ktorý sa podľa nich stal základom jazyka LINQ
- Typický príklad:

```
IList<Student>students =
  db.Query<Student>(delegate(Studentstudent)
  { return student.Age < 20
    && student.Grade == gradeA; });
```

Iné spôsoby dotazovania

- Query by example

```
Car car = new Car();
car.setName(“Ferrari”);
List cars = db.get(car);
```

- S.O.D.A (?): a powerful node-based query API for dynamic query creation at runtime. It allows triggering any custom code on servers, thereby reducing bandwidth and speeding query execution.

Základné princípy

- db4o umožňuje jednoduchým spôsobom realizovať perzistenciu objektov v prostrediach .NET a Java
- Môže sa pri tom samozrejme jednať o ľubovoľne zložité objekty (objektové štruktúry)
- umožňuje vyhľadávať objekty v perzistentnom úložisku (“databáze”)
- umožňuje pracovať s perzistentnými objektovými štruktúrami takmer rovnako ako keby to boli štruktúry vo vnútornej pamäti

ability to easily store...	RDBMS	non-native OODBMS	db4o
flat object structures	X	X	X
tall object trees		X	X
evolving object structures			X

db4o engine

- Celý dbms je realizovaný v jednej dll: Db4objects.Db4o.dll
- Táto dll býva inštalovaná v závislosti do bin adresára príslušnej verzie .NET:

  /db4o-7.4/bin/net-2.0/Db4objects.Db4o.dll

  /db4o-7.4/bin/compact-2.0/Db4objects.Db4o.dll

  /db4o-7.4/bin/net-3.5/Db4objects.Db4o.dll

  /db4o-7.4/bin/compact-3.5/Db4objects.Db4o.dll

- Inštalácia: pozostáva z prilinkovania tejto dll ako referencie do projektu

Dokumentácia

- K systému existuje
  - Popis API
  - Tutorial
  - Niekoľko “white paper” publikácií
- Všetky sa dajú nájsť na stránke http://developer.db4o.com (je nutné zaregistrovať sa)

Namespace Db4objects.Db4o

- Obsahuje základné triedy systému Db4oFactory a IObjectContainer
- Db4objects.Db4o.Db4oFactory:
  - Otvorenie databázového súboru
  - Naštartovanie servera
  - Pripojenie k existujúcemu serveru
- Db4objects.Db4o.IObjectContainer je najdôležitejším rozhraním
  - Predstavuje samotný db4o DBMS

Db4objects.Db4o.IObjectContainer

- IObjectContainer môže byť buď DBMS v jednoužívateľskom režime alebo klientským pripojením ik inému db4o DBMS
- Každý IObjectContainer vlastní (?) jednu transakciu.
  - Všetky operácie sú súčasťou transakcie
  - Pri štarte servera sa automaticky spustí transakcia
  - Každý Commit() či Rollback() automaticky spustí novú transakciu
- Každý IObjectContainer udržiava vlastné referencie k uloženým objektom *(stored and instantiated(?) objects)*

- IObjectContainery musia zostať otvorené po celú dobu počas ktorej pracujeme s objektami v nich uloženými. Pri zatvorení IObjectContainera sú všetky objekty odstránené z vnútornej pamäte

Namespace Db4objects.Db4o.Ext

- Rozširuje Db4objects.Db4o
- Metódy IObjectContainera sú rozdelené do týchto dvoch namespaceov. Db4o obsahuje základné metódy, Ext rozširujúce
- Každý IObjectContainer je zároveň IExtObjectContainer

Namespace Db4objects.Db4o.Config a Query

- Db4objects.Db4o.Config obsahuje typy potrebné na konfiguráciu db4o
- Db4objects.Db4o.Query obsahuje triedu Predicate potrebnú na vytváranie Native Queries

Príklad

```
public class Pilot
{
   string _name;
   int _points;

   public Pilot(string name, int points)
   {
      _name = name;
      _points = points;
   }

   public void AddPoints(int points)
   {
      _points += points;
   }
}
```

Trieda tiež obsahuje properties Name a Points

Otvorenie a zatvorenie databázy

```
IObjectContainer db =
      Db4oFactory.OpenFile(Util.YapFileName);
try
{
   // do something with db4o
}
finally
{
   db.Close();
}
```

- Pozor: pri zatvorení databázy sú z vnútornej pamäte odstránené všetky objekty uložené v databáze
- Databázu je dobré otvoriť pri spustení aplikácie

Vloženie objektov

```
   Pilot pilot1 = new Pilot("Michael
Schumacher", 100);
   db.Set(pilot1);
   Console.WriteLine("Stored {0}", pilot1);
   Pilot pilot2 = new Pilot("Rubens
Barrichello", 99);
   db.Set(pilot2);
```

- Odteraz budeme vychádzať z predpokladu, že databáza je otvorená
- Objekt sa uloží zavolaním metódy Set(), ktorej sa ako parameter predá objekt, ktorý chceme uložiť

Metóda IObjectContainer.Set

- Deklarácia:

```
void Set(Object obj)
```

- Metóda Set uloží objekt *obj* do databázy. To znamená:
  - Ak objekt v databáze ešte neexistuje, tak sa do nej vloží
  - Ak objekt v databáze existuje, tak sa updatuje
- Od verzie 7.4. existuje nová metóda s rovnakou(?) sémantikou: IObjectContainer.Store

Výber údajov

- Db4o poskytuje 3 spôsoby vyhľadávania a výberu dát:
  - Query By Example (QBE)
  - Native Queries
  - S.O.D.A.

Vyhľadávanie pomocou QBE

- QBE využíva „prototyp“ objekt
- Pomocou prototypu sa vyhľadajú iné objekty, ktoré sa naňho podobajú
- „Podobajú sa“ znamená:
  - sú rovnakého typu
  - majú rovnaké hodnoty pre tie atribúty, ktoré v prototype nemajú „defaultné“ hodnoty
- Výsledok sa vráti ako IObjectSet

Príklady

Výber pilota podľa mena:

```
Pilot proto = new Pilot("Michael Schumacher", 0);
IObjectSet result = db.Get(proto);
```

Výber pilota podľa počtu bodov:

```
Pilot proto = new Pilot(null, 100);
IObjectSet result = db.Get(proto);
```

Výber všetkých pilotov I:

```
Pilot proto = new Pilot(null, 0);
IObjectSet result = db.Get(proto);
```

Výber všetkých pilotov II:

```
IObjectSet result = db.Get(typeof(Pilot));
```

Metóda IObjectContainer.Get

- QBE interface pre výber objektov
- Deklarácia:

```
IObjectSet Get(Object template)
```

- Nenulové *(non-null)* členy *(members)* template objektu sa porovnajú so všetkými objektami tej istej triedy. Členy primitívnych typov *„are ignored if they are 0 or False respectively“*

IObjectSet

- Reprezentuje množinu výsledkov, ktoré vráti nejaký dotaz
- Počas doby, keď s IObjectSetom pracujeme, musí zostať príslušná databáza otvorená!
- Metódy
  - HasNext
  - Next
  - Reset (nastaví kurzor na začiatok zoznamu)
  - Size

QBE: zhrnutie

- Tradičný mechanizmus dotazovania v OODBMS
- Veľmi primitívny
  - Neumožňuje porovnanie iné než =
  - Neumožňuje žiadne logické výrazy
  - Žiadne zložitejšie operácie (join)
- Samotní tvorcovia systémov radia používať radšej iné dotazovacie mechanizmy (native queries)

Native Queries

- Hlavný dotazovací mechanizmus db4o
- Využíva sémantiku hostiteľského prostredia (C#, VB, Java):
  - je preto dobre integrovateľný s aplikačnými programmi
  - je zabezpečená typová bezpečnosť, compile-time typové kontroly
- Dotazy sa vyhodnocujú pomocou metódy IObjectContainer.Query
- Existuje 11 rôznych spôsobov volania (overloads) tejto metódy

Princíp

- Native Queries umožňuju vykonať nejaký kus kódu oproti všetkým objektom danej triedy (testovacie výrazy dotazu).
- Ak sa tieto testy pre nejaký objekt vyhodnotia ako TRUE, objekt je zaradený do výslednej množiny
- db4o sa snaží tieto testy optimalizovať a vykonávať ich – tam kde je to možné – oproti indexom

Problémy klasických dotazovacích mechanizmov

- Dotaz býva tradične schovaný v nejakom reťazci.
- Obsah tohto reťazca je pre hostiteľské prostredie nezrozumiteľný.
- Nie je preto možné:
  - vykonávať typové kontroly
  - optimalizovať dotazy
  - využívať výhody moderných IDE (automatické kontroly kódu a jeho refactoring, ...)

Native queries: princípy dizajnu

- Cieľom dizajnu je zrealizovať dotazovací mechanizmus, ktorý je dobre integrovateľný s hostiteľským prostredím a odstraňuje uvedené problémy.
- To znamená, že musí spĺňať nasledujúce kritériá:
  - Musí byť 100% native: Všetky dotazy musia byť úplne formulovateľné v hostiteľskom jazyku
  - Musí byť 100% objektovo orientovaný
  - Musí byť 100% typovo bezpečný
  - Musí byť optimalizovateľný

Dizajn (1)

- Predikáty samotné sú ľahko formulovateľné v hostiteľskom jazyku:

```
student.Age < 20 && student.Name.Contains("f")
```

- Potrebujeme nájsť spôsob, ako predať tomuto predikátu nejaký objekt z databázy na vyhodnotenie a ako predať výsledok vyhodnotenia späť do aplikácie:

```
(Student student){
                    return student.Age < 20 &&
                    student.Name.Contains("f");
                  }
```

Dizajn (2)

- Toto potrebujeme zabaliť tak, aby sa z toho stal kód použiteľný v hostiteľskom jazyku
- To dosiahneme zapuzdrením tohto kódu do nejakého objektu
- Aby bol tento objekt použiteľný pre dotazovanie pomocou native queries db4o, musí spĺňať nasledujúce podmienky
  - Musí byť odvodený od triedy *Predicate*
  - Metóda obsahujúca predikát muí byť typu *boolean* a musí sa volať *Match*

Príklad 1:

- Jednoduchý dotaz: výber všetkých objektov nejakého typu:

```
IList <Pilot> pilots = db.Query<Pilot>(typeof(Pilot));
```

Príklad 2: definícia dotazu

```
public class ArbitraryQuery : Predicate
{
   private int[] _points;

   public ArbitraryQuery(int[] points)
   {
      _points=points;
   }

   public bool Match(Pilot pilot)
   {
      foreach (int points in _points)
      {
         if (pilot.Points == points)
         {
            return true;
         }
      }
      return pilot.Name.StartsWith("Rubens");
   }
```

Príklad 2: použitie dotazu

```
IObjectSet result = db.Query(new
          ArbitraryQuery(new int[]{1,100}));
```

Pomenované a anonymné triedy

- Predchádzajúci spôsob definície predikátu je použiteľný ako v Jave tak v .NET
- Ide vlastne o definíciu predikátu pomocou klasickej pomenovanej triedy, ktorá na to, aby mohla byť použiteľná na dotazovanie, je odvodená od nejakej štandardnej triedy *(Predicate)*
- .NET navyše umožňuje definovať predikáty pomocou tzv. anonymnej triedy

Odbočka: delegáti a anonymné metódy

- Delegát je typ, ktorý reprezentuje nejakú metódu (=zneužitie pojmu „typ“ na programátorské triky)
- Delegát môžeme volať rovnako ako akúkoľvek inú metódu pomocou parametrov a návratovej hodnoty
- Príklad:

```
public delegate int PerformCalculation(int x, int y);
```

- Delegátovi môžeme priradiť ľubovoľnú metódu z akejkoľvek inej triedy. Stačí, keď má rovnakú signatúru ako delegát

Delegát: príklad

```
public delegate void Del(string message);

// Create a method for a delegate.
public static void DelegateMethod(string
  message)
{
    System.Console.WriteLine(message);
}

// Instantiate the delegate.
Del handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

Delegáti: použitie

- Delegáti sú jedným zo spôsobov realizácie polymorfizmu
- Jedna deklarovaná metóda (delegát) môže byť implementovaná rôznymi spôsobmi
- Trocha sa to podobá na rozhrania

Anonymné metódy

- V predchádzajúcom príklade sme si ukázali použitie a definíciu delegáta pomocou pomenovanej triedy *(Del, PerformCalculation)*
- Inou alternatívou sú tzv. anonymné metódy
- Príklad:

```
// Create a handler for a click event

button1.Click = delegate(System.Object o,
  System.EventArgs e)                {
  System.Windows.Forms.MessageBox.Show("Click!"); };
```

- Anonymné metódy sú teda spôsobom ako ľubovoľný kus kódu povýšiť na objekt
- To znamená, že tam, kde program očakáva

Anonymné metódy a native queries

- db4o využíva anonymné metódy na predanie dotazu metóde Query

```
IList <Pilot> pilots = db.Query <Pilot>
                        (delegate(Pilot pilot)
   {
      return pilot.Points == 100;
   });
```

```
IList <Pilot> result = db.Query<Pilot>
                        (delegate(Pilot pilot)
   {
      return pilot.Points > 99
      && pilot.Points < 199
      || pilot.Name == "Rubens Barrichello";
   });
```

Podpora Linq

- Podobným konceptom ako Native Queries od db4o je jazyk Linq od Microsoftu
- Spĺňa rovnaké princípy dizajnu
- Je dostupný vo Visual Studiu 2008 a frameworku .NET verzii 3.5
- Podobá sa trocha na SQL ale je dobre integrovaný s hostiteľským prostredím
- Umožňuje pristupovať rovnakým spôsobom k rôznym typom zdrojov dát (SQL, objektové databázy, XML,...)

Linq: príklad

```
IEnumerable<Pilot> result = from Pilot p in db
    where p.Name.StartsWith("Michael")
    select p;
```

Zložité objekty

- Náš príklad rozšírime o vozidlo (podľa tutorialu db4o dáme pilotovi vozidlo):

```
public class Car
{
    string _model;
    Pilot _pilot;

    public Car(string model)
    {
        _model = model;
        _pilot = null;
    }
}
```

Vloženie zložitých objektov do databázy

```
Car car1 = new Car("Ferrari");
Pilot pilot1 = new Pilot("Michael Schumacher",
  100);
car1.Pilot = pilot1;
db.Set(car1);

Pilot pilot2 = new Pilot("Rubens Barrichello", 99);
db.Store(pilot2);
Car car2 = new Car("BMW");
car2.Pilot = pilot2;
db.Set(car2);
```

- Zložité objekty sa teda do databázy vkladajú rovnako ako jednoduché objekty: pomocou metódy Set

Výber zložitých objektov

- Zložité objekty sa vyberajú presne tak isto ako jednoduché objekty

```
public class RetrieveCarsPredicate : Predicate
{
   readonly string _pilotName;

   public RetrieveCarsPredicate(string pilotName)
   {
      _pilotName = pilotName;
   }
   public bool Match(Car candidate)
   {
      return candidate.Pilot.Name == _pilotName;
   }
}

// retrieveCarsNative
string pilotName = "Rubens Barrichello";
```

Výber zložitých objektov pomocou Linq

```
IEnumerable<Pilot> result = from Car c in db
where c.Model.StartsWith("F")
&& (c.Pilot.Points > 99 && c.Pilot.Points < 150)
select c.Pilot;
```

Updatovanie zložitých objektov

- Pomocou metódy Set
- **„Update depth“**: V prípade skutočne zložitých objektov (objektov s vysokým zaťažením) môže byť updatovanie z hľadiska výkonu náročnou operáciou
  - ak je nutné uložiť všetky úrovne objektu
- Update depth umožňuje obmedziť počet úrovní, ktoré sa pri update operácii traverzujú
- Defaultná hodota je 1: updatujú sa iba členy primitívnych typov a typu string!!!

Závery

- Nie je to databáza ale mechanizmus perzistencie
- Koniec koncov sa dáta tak či tak ukladajú ako akési tabuľky
- Vychádza z rovnice relácia = type extent

Architektúra softwarových systémov

Obsah

- Definícia architektúry
- Architektonické modely
- Architektonické princípy
- Architektonické štýly
- Architektonické vzory
- Architektonické postupy

Definícia architektúry

- **Klasická definícia:** The art and science of **designing** and **erecting** buildings.

- *Architecture is the art of how to waste space*

Informačný (softwarový) systém

Informačný systém (IS) je množina

- ľudí
- techniky
- dát
- procesov

ktoré sú vo vzájomnej interakcii za účelom

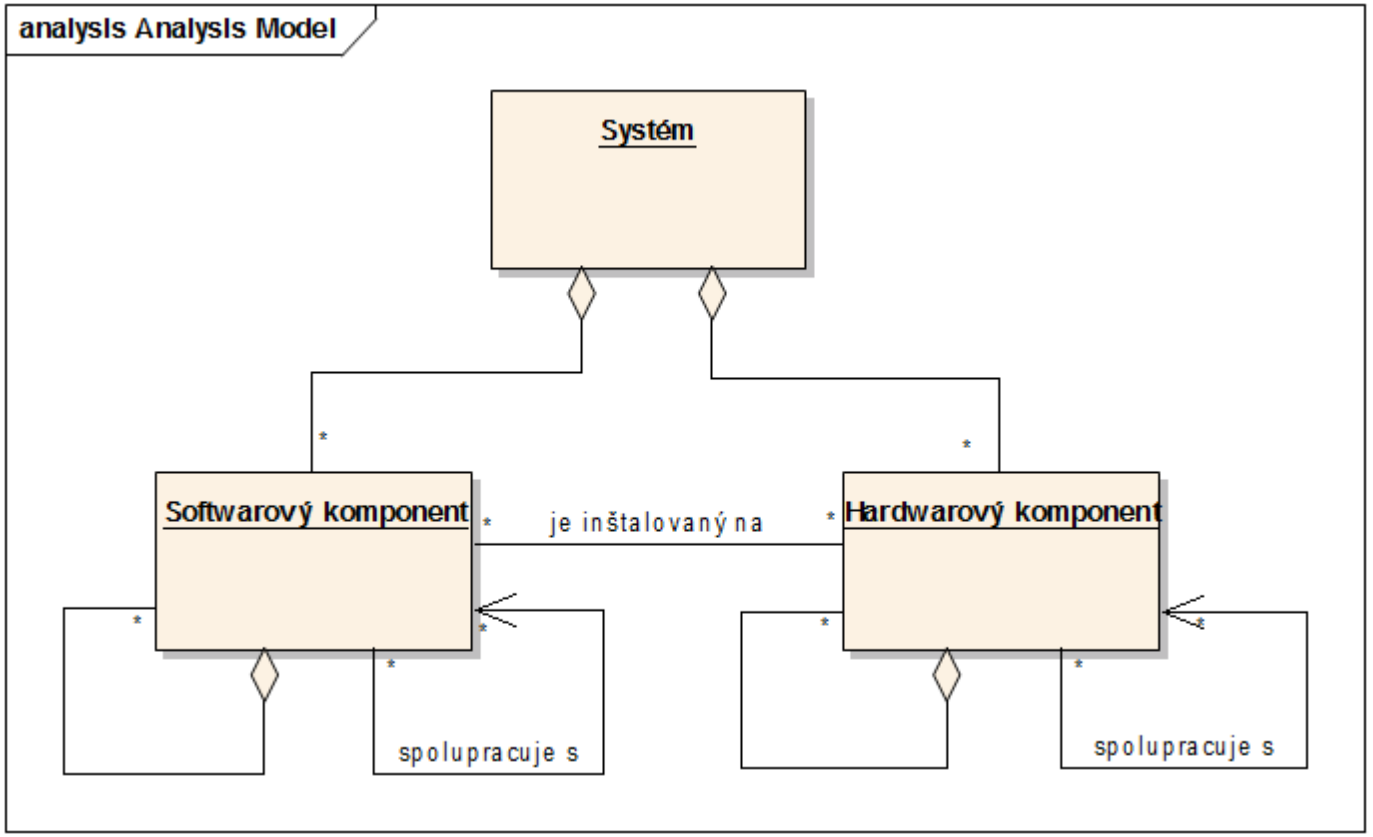
- zberu
- spracovania
- ukladania
- poskytovania (vo forme výstupu)

informácií potrebných na fungovanie danej organizácie.

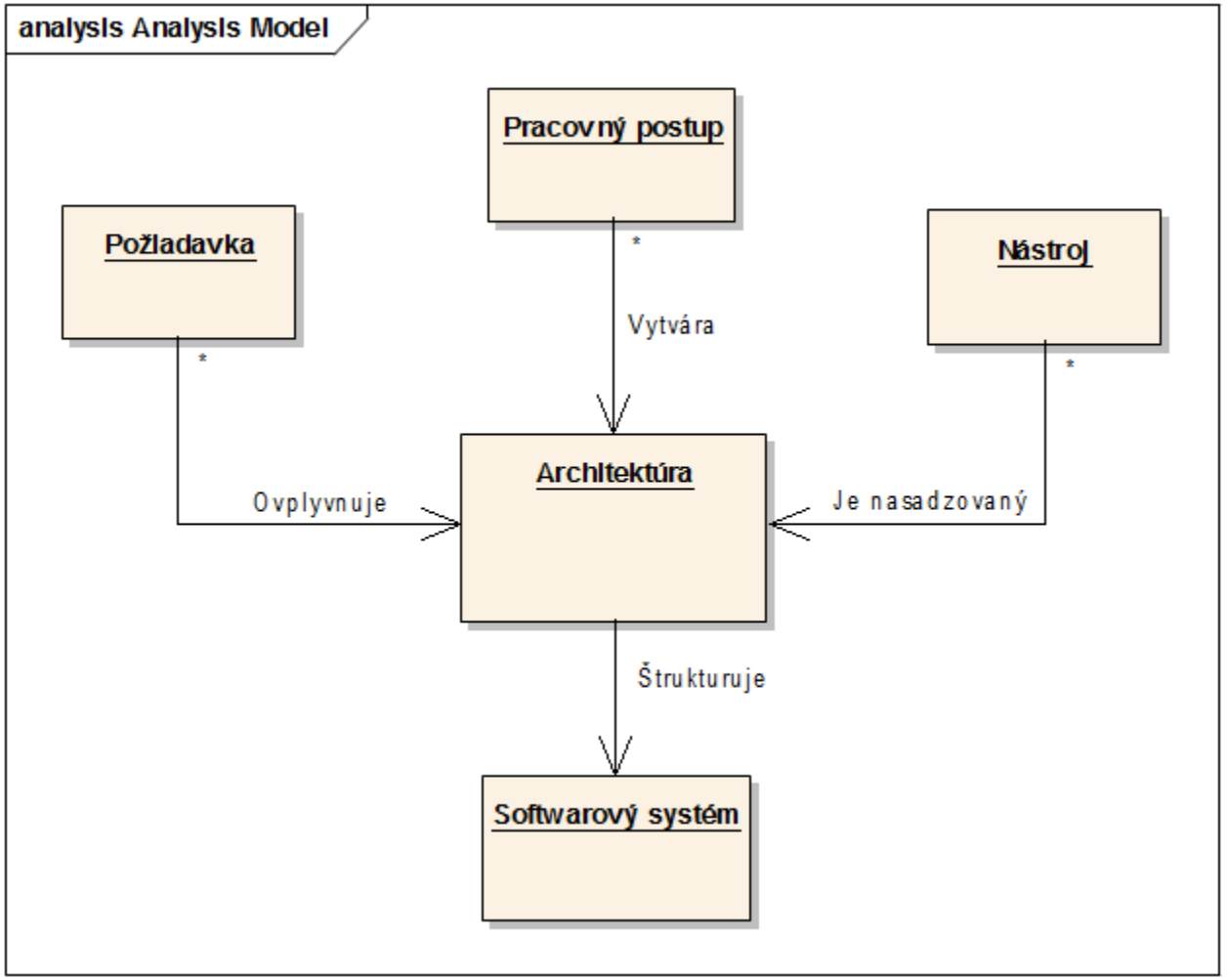
Architektúra softwarového systému

- **Štrukturálny pohľad:**
  - Architektúra popisuje štruktúru daného systému: jeho komponenty, ich viditeľné vlastnosti a vzťahy
- **Disciplinárny pohľad:**
  - Architektúra ako disciplína sa zaoberá činnosťami spojenými s koncepciou a realizáciou softwarových systémov

Komponenty softwarového systému



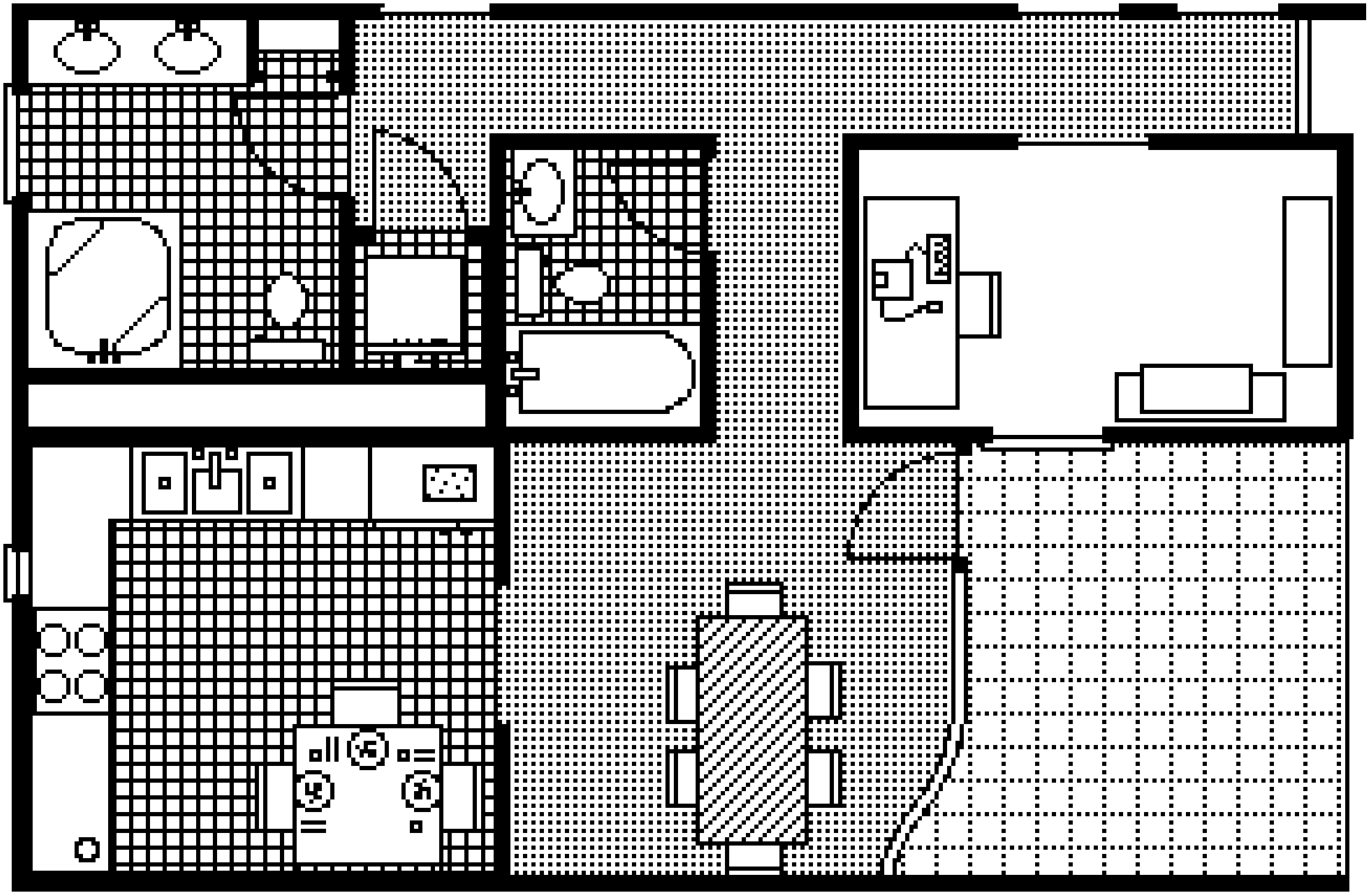
Architektúra ako disciplína



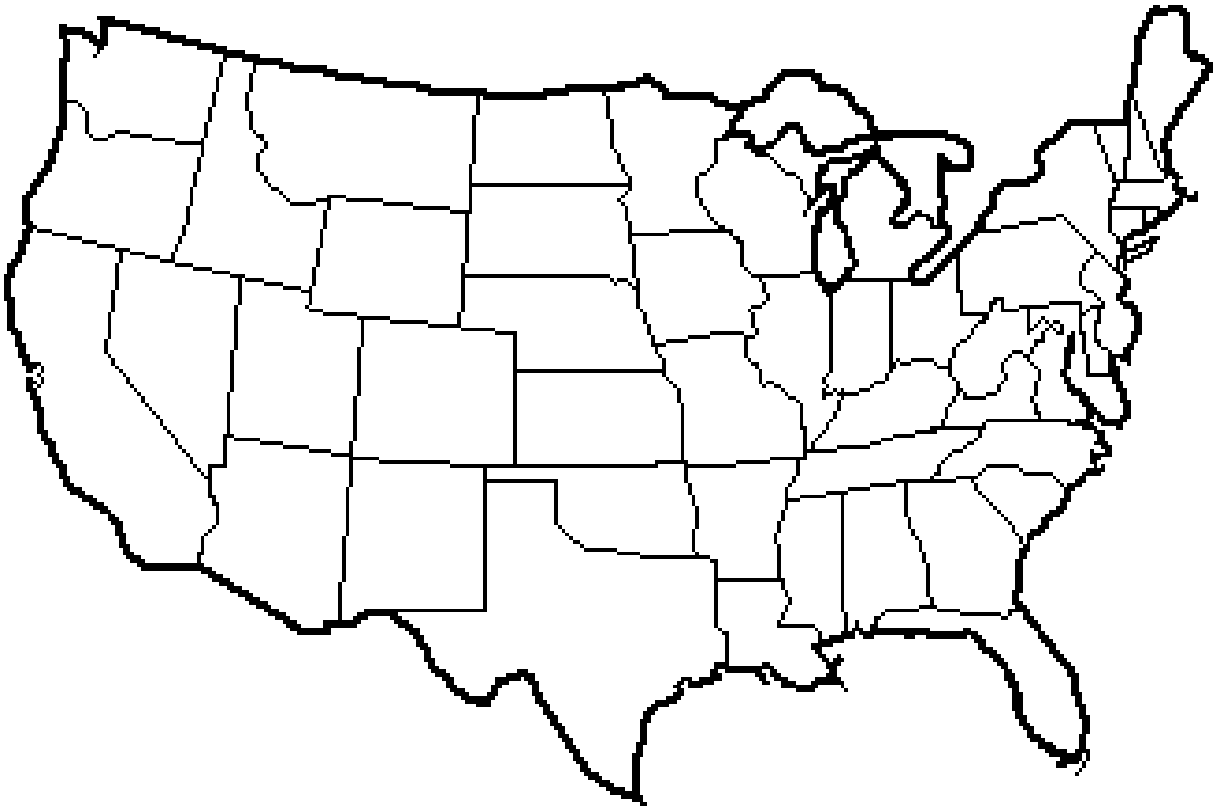
Čo je model?

- Model– abstrakcia reálneho objektu. Zobrazuje časť vlastností daného objektu, iné vlastnosti zámerne nezobrazuje
- Model predstavuje **úplný** systém. Umožňuje rôzne pohľady na daný systém podľa záujmu toho, kto model používa.

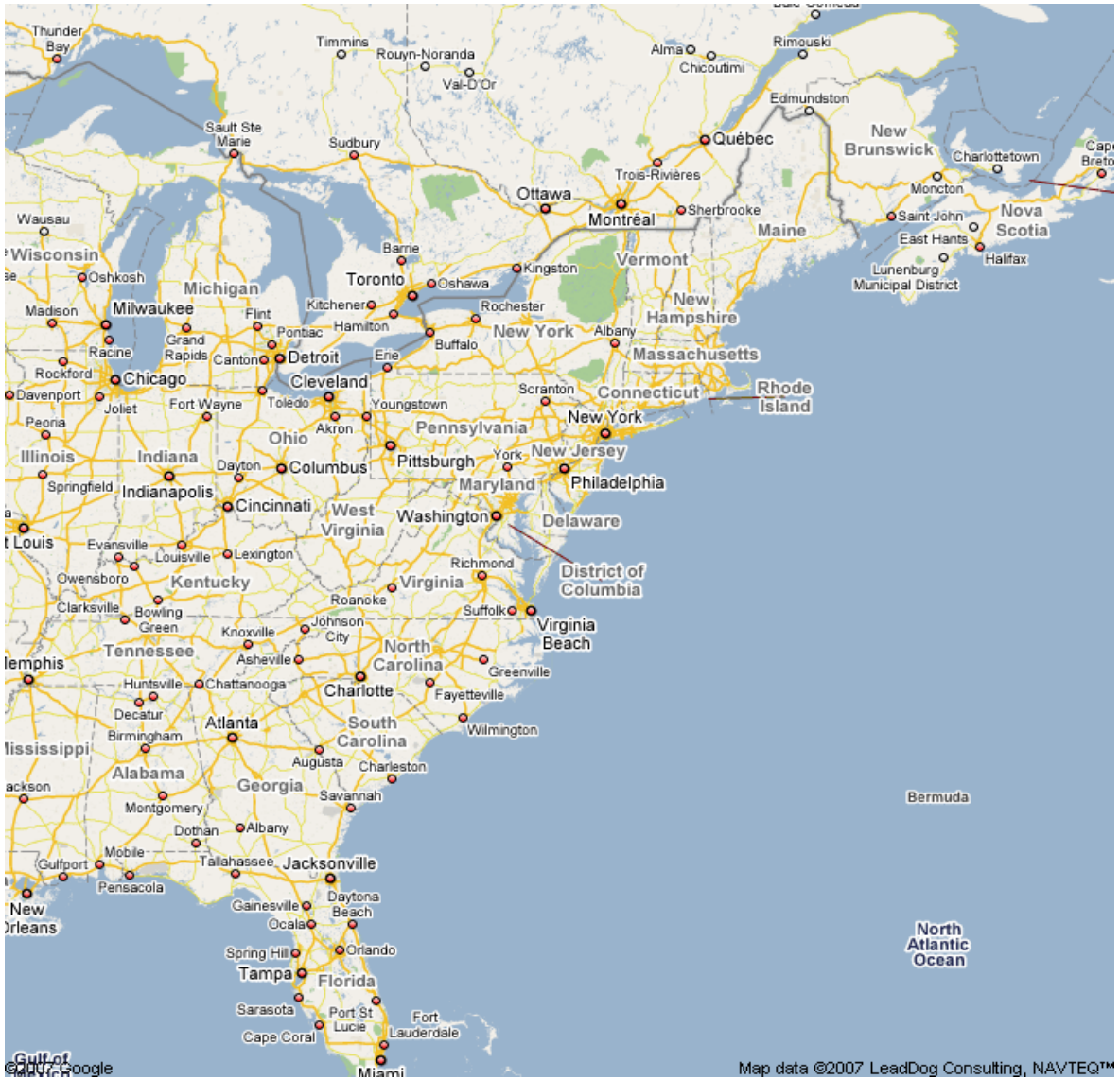
Príklad modelu



Iný príklad modelu



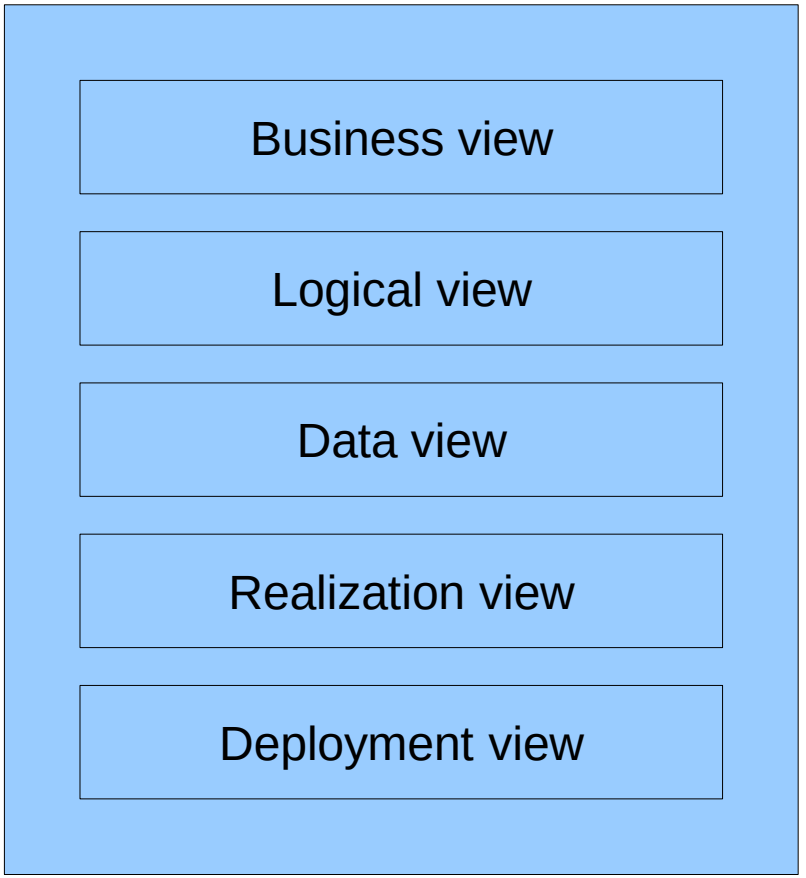
Ešte jeden príklad modelu



Všeobecné vlastnosti modelov (Yourdon)

- Mali by byť grafické s dostatočnou podporou .
- Mali by poskytnúť možnosť dekomponovať problém systémom top-down.
- Mali by byť minimálne redundatné
- Mali by čitateľovi pomôcť odhadnúť správanie systému
- Mali by byť pre čitateľa transparentné

Príklad abstraktného modelu architektúry



- Vzťahy medzi jednotlivými pohľadmi sú komplexné a mnohosmerné

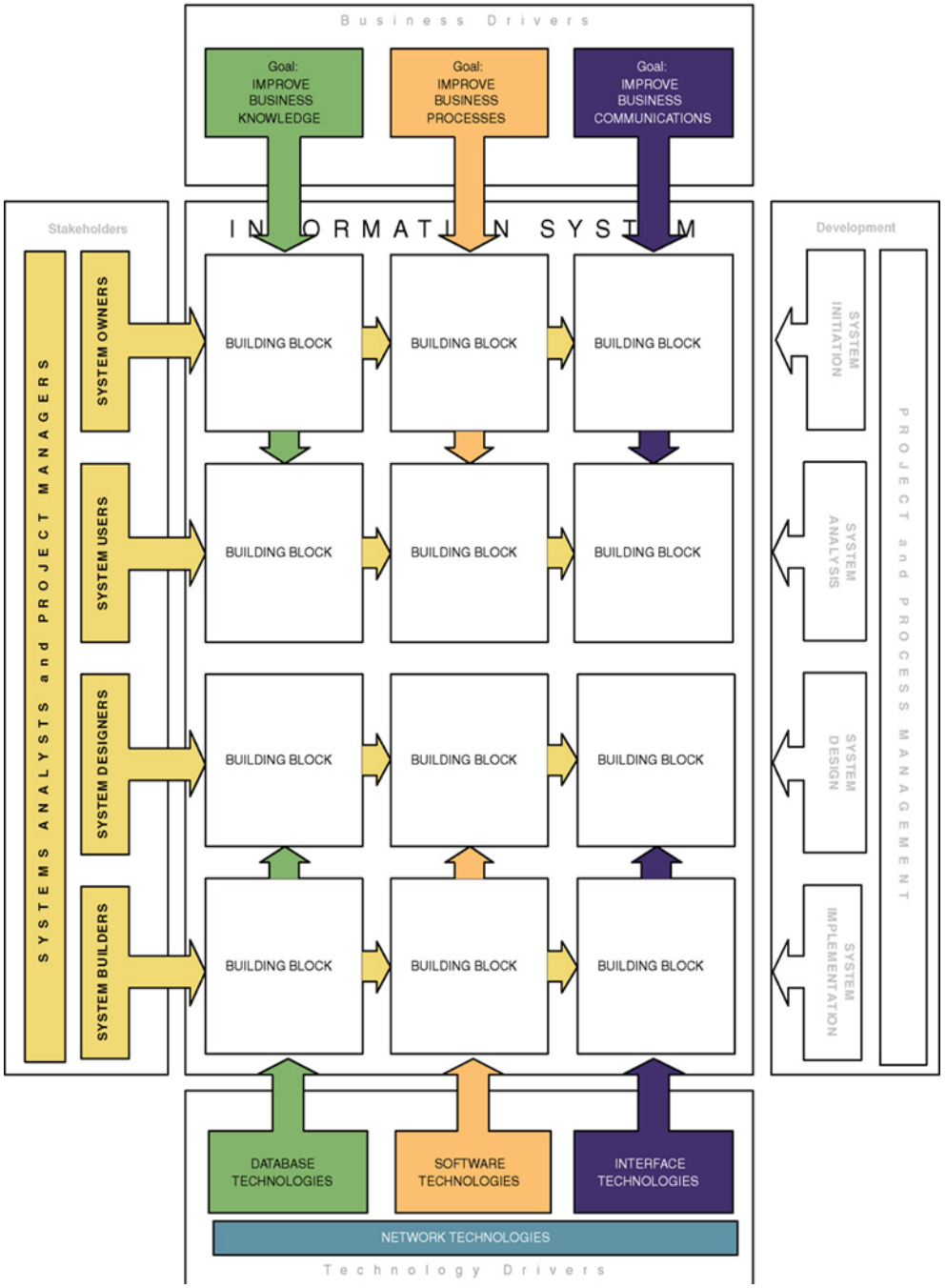
Pohľady na architektonický model

- **Business view:** business procesy a prípady užitia, odborné modely, organizačné modely, stakeholders,...
- **Logical view:** logické modely dát a ich spracovania, dekompozícia systému na komponenty, rozhrania komponentov, „responsibilities“ komponentov, hranice systému, scenáre dôležitých prípadov užitia, ...

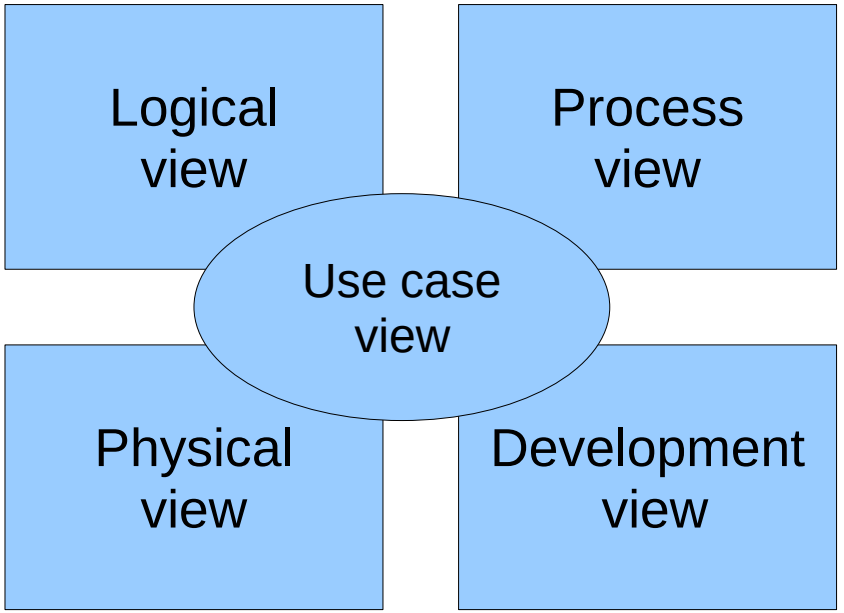
Úrovne architektonického modelu (pokračovanie).

- **Data view:** definície tabuliek a atribútov, dátové toky medzi komponentmi, dôležité zdroje dát, vzťah komponentov a perzistentných dát, ...
- **Realization view:** voľba nástrojov (programovacieho jazyka), pracovné postupy a štandardy, organizácia a manažment zdrojového kódu, testovanie, version management,...
- **Deployment view:** rozdelenie komponentov na fyzické uzly, inštalácia a konfigurácia, výkonnosť, ...

Zachmanov framework



4+1 model


Logical view
Process view
Use case view
Physical view
Development view

Pojem „doména triedy“

- Odkiaľ sa berú triedy?
- Porovnajte triedy: Student, ListBox, Collection, Date, Connection
- Všetky tieto triedy sa používajú iným spôsobom a hlavne: sú z inej oblasti – domény
- Význam termínu doména si vysvetlíme pomocou príkladov

Doména základu (foundation domain)

- Triedy z domény základu sú využiteľné v mnohých aplikáciách z mnohých rôznych oblastí na širokom spektre počítačových architektúr
- Doména základu zahŕňa triedy s najširšou možnou opakovanou použiteľnosťou
- Obsahuje tri základné skupiny tried:
  - Základné triedy
  - Štrukturálne triedy
  - Sémantické triedy

Základné triedy

- Základné triedy sú napríklad Integer, Boolean, Char
- V mnohých jazykoch nie sú implementované ako triedy, ale ako základné dátové typy

Sémantické triedy

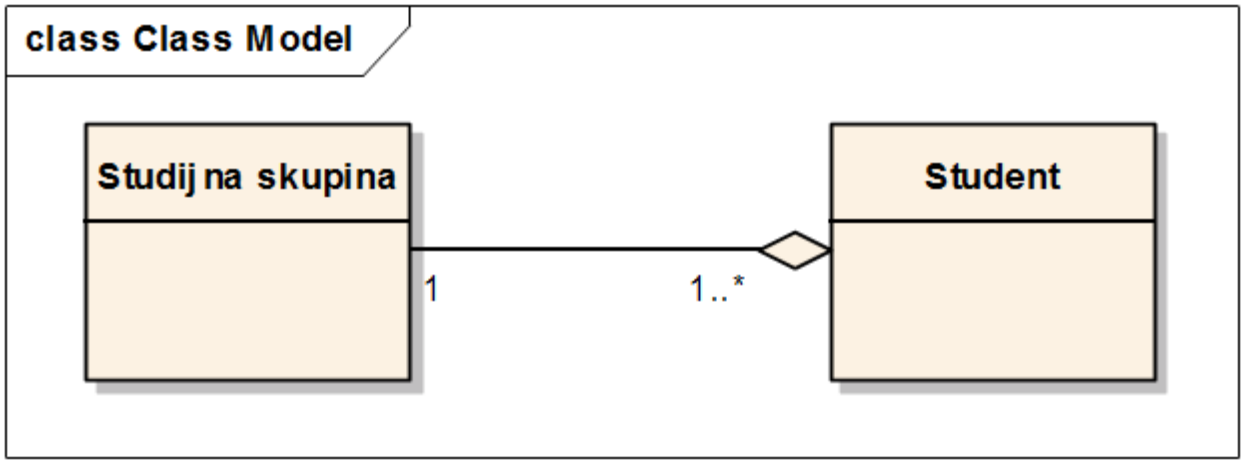
- Podobné ako základné triedy, majú však o niečo širší sémantický význam
- Napríklad Date, Time, Money, Teplota
- Ich hodnoty je možné vyjadriť v nejakých jednotkách

Štrukturálne triedy

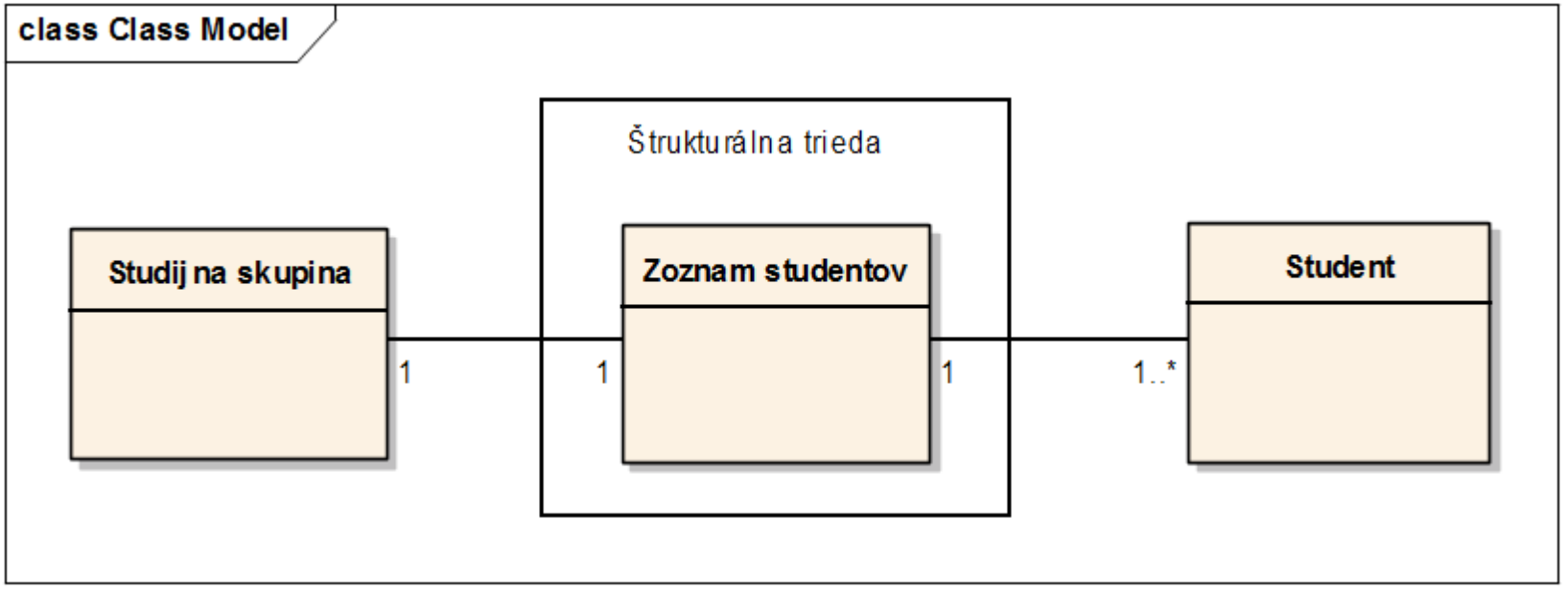
- Štrukturálne triedy implementujú štruktúry (iných tried)
- Napríklad triedy Zásobník, Fronta, Strom, Zoznam, Set,...
- Často sa označujú ako kontajnerové triedy
- Často sa vytvárajú s využitím princípu obecnosti

Analytické modely a štrukturálne triedy

Analýza



Návrh



Doména architektúry

- Triedy z domény architektúry sú využiteľné v mnohých aplikáciách rôznych odvetví.
- Obmedzujú sa však na jednu počítačovú architektúru
- Triedy komunikácie s prístrojmi (Port, RemoteDevice)
- Triedy práce s databázami (Transaction, Backup, Connection)
- Triedy užívateľského rozhrania (Okno, ListBox)

Doména činnosti (business domain)

- Triedy z domény činnosti sú užitočné v mnohých aplikáciách, ale iba v rámci jednej oblasti
- Oblasťami sú napríklad bankovníctvo, železničná doprava, atď
- Doména činnosti má tri skupiny tried:
  - Triedy atribútov
  - Triedy rolí
  - Triedy vzťahov

Triedy rolí

- Vyplývajú z toho, aké roly hrajú „veci“ v danom obore.
- Napríklad: Klient (banky) alebo Pacient
- Tieto triedy budú pri analýze domény činnosti zrejme najzreteľnejšie a budú identifikované ako prvé
- Reprezentujú „veci“ samotné. Pri modelovaní sa na „veci“ dívame vždy z pohľadu roly, ktorú hrajú v danom systéme. Preto triedy rolí a nie triedy „vecí“

Triedy atribútov

- Zachycujú vlastnosti vecí vo svete danej oblasti
- Napríklad: ZostatokNaÚčte, TelesnáTeplota (pacienta)
- Podobajú sa triedam Peniaze a Teplota z domény základu: nie je to však to isté
- Pre vlastnosti tried z domény atribútov budú platiť isté pravidlá, ktoré pre pre triedy z domény základu neplatia
- Napríklad ak TelesnáTeplota > 42, tak máme vážny problém

Triedy vzťahov

- Vyplývajú zo vzťahov „vecí“ vo svete daného oboru
- Patrí sem napríklad VlastníctvoÚčtu alebo DohľadPacienta
- Často sa budú realizovať štrukturálnymi triedami z domény základu. Zasa však platí nie je to to isté

Doména aplikácie

- Trieda v doméne aplikácie sa používa v jedinej aplikácii (alebo v malom počte súvisiacich aplikácií)
- Triedy rozpoznávania udalostí: softwarové konštrukcie hľadajúce výskyt určitých udalostí. Napr. MonitorTeplotyPacienta, ktorý stráži udalosť „TeplotaPrekročila38Stupňov“
- Triedy správy udalostí: vykonávajú príslušné aktivity danej činnosti. Napr. NaplánovanieOperácie Pacienta

Domény a znovupoužiteľnosť

Architektonické princípy: všeobecné (Vitruvius)

- Krása (Venustas)
- Trvanlivosť (Firmitas)
- Užitočnosť (Utilitas)

Architektonické princípy

- Loose coupling
- High cohesion
- Design for change
- Incrementality
- Traceability
- Abstraction
- Information Hiding
- Separation of concerns
- Modularity
- Self-documentation

Loose coupling

- Loose coupling describes a resilient relationship between two or more systems or organizations with some kind of exchange relationship. Each end of the transaction makes its requirements explicit and makes few assumptions about the other end
- Jednotlivé komponenty systému by mali byť medzi sebou previazané čo najmenej (slabé väzby)
- Dajú sa definovať miery previazanosti

Miera previazanosti: Zaťaženie (Encumbrance)

- Wikipedia: Encumbrance is a legal term of art for anything that affects or limits the title of a property, such as mortgages, leases, easements, liens, or restrictions.
- Zaťaženie meria celkové „doplnkové strojné vybavenie“ triedy. Inými slovami, koľko iných tried daná trieda potrebuje k svojmu fungovaniu

Množina priamych odkazov na triedy (Direct class-reference set)

- Množina tried, na ktoré sa trieda T priamo odkazuje
- Trieda T sa môže na inú triedu U priamo odkazovať nasledujúcimi spôsobmi:
  - T dedí z U
  - T má nejaký atribút (alebo property) triedy U
  - T má nejakú operáciu so vstupným parametrom typu U
  - T má metódu, ktorá posiela správu s návratovým parametrom typu U
  - T má metódu, ktorá má lokálnu premennú typu U
  - T predáva U ako vlastný parameter nejakej parametrizovanej triede
  - T má friend class U (C++)

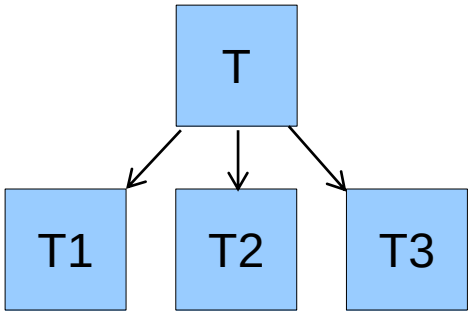
Nepriame odkazy na triedy

- Nech množina priamych odkazov triedy T zahŕňa triedy $\{T_1, T_2, ..., T_n\}$ Potom množina **nepriamych odkazov na triedy** danej triedy T je zjednotením množiny priamych odkazov triedy T a množín nepriamych odkazov tried T_1, T_2, ..., T_n
- Táto definícia je zjavne rekurzívna. Táto rekurzia končí väčšinou u tried z domény základu, pretože ich množina priamych odkazov je prázdna

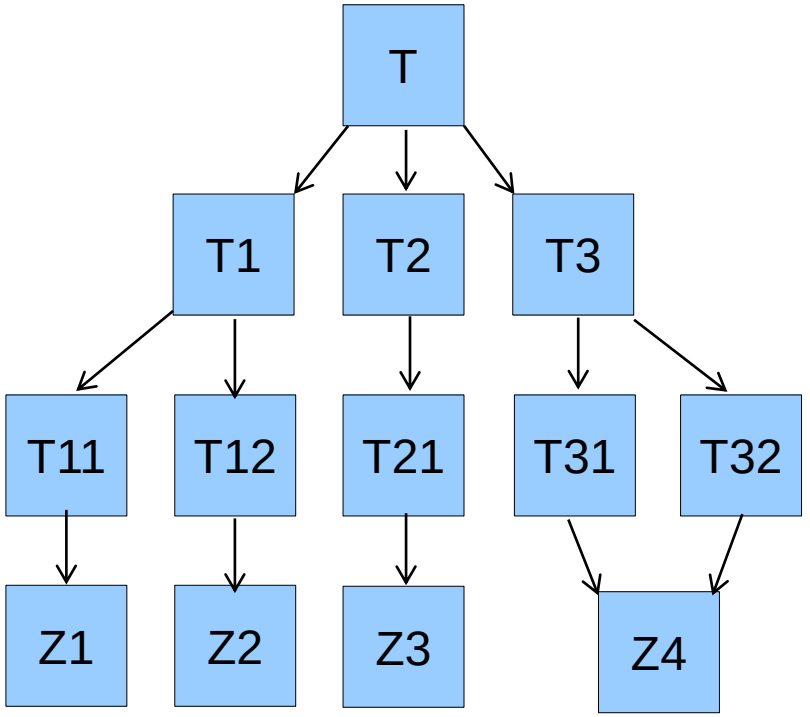
Zaťaženie triedy: definícia

- **Priame zaťaženie** triedy je veľkosť jej množiny priamych odkazov na triedy
- **Nepriame zaťaženie** triedy je veľkosť jej množiny nepriamych odkazov na triedy
- K označeniu veľkosti množiny priamych odkazov danej triedy na triedy sa niekedy používa aj termín **celkové viazanie triedy (total class coupling)**

Zaťaženie triedy: príklady



Trieda T a jej množina priamych odkazov na triedy



Trieda T a jej množina nepriamych odkazov na triedy

Použitie zaťaženia

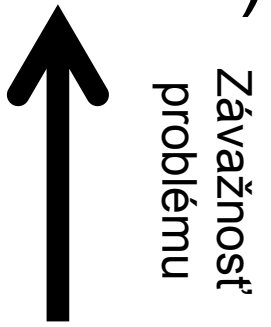
- Zaťaženie je mierou zložitosti triedy
- Vysoké zaťaženie triedy indikuje chybu v návrhu
- Znižuje sa znovupoužiteľnosť danej triedy:
  - Ak chcem použiť existujúcu triedu X, ktorá má vysoké zaťaženie, musím spolu s ňou použiť aj všetky triedy, ktoré ju zaťažujú

High cohesion (=súdržnosť)

- Cohesion is a measure of how strongly-related and focused the various responsibilities of a software module are. Cohesion is an ordinal type of measurement and is usually expressed as "high cohesion" or "low cohesion" when being discussed.
- Ide teda o väzby a závislosti **vnútri** jedného komponentu, tie by mali byť čo najväčšie

Súdržnosť triedy

- Súdržnosť triedy je miera vzájomnej závislosti prvkov v externom rozhraní danej triedy
- Trieda s nízkou (zlou) súdržnosťou má sady prvkov, ktoré k sebe nepatria
- V súvislosti so súdržnosťou rozlišujeme tri typy nesprávneho návrhu (zmiešaná súdržnosť):
  1. Zmiešaná súdržnosť inštancií
  2. Zmiešaná súdržnosť domén
  3. Zmiešaná súdržnosť rolí



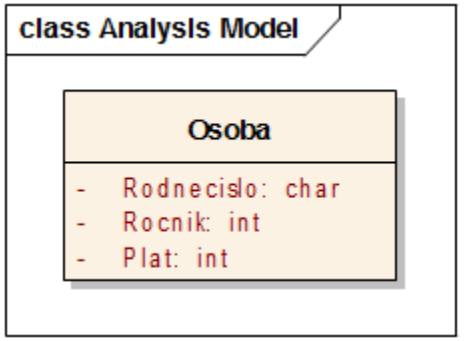
Zmiešaná súdržnosť inštancií

- **Definícia:** Trieda so zmiešanou súdržnosťou inštancií má nejaké prvky, ktoré sú pre určité objekty triedy nedefinované
- Zmiešaná súdržnosť inštancií obvykle definuje zle definovanú hierarchiu tried
- Takisto môže viesť k nepekným programovým konštrukciám náročným na údržbu

Zmiešaná súdržnosť inštancií: príklad

- Nech trieda Osoba sa používa na ukladanie informácií jednak o študentoch ako aj zamestnancoch fakulty

  Potom študenti nemajú definovaný plat a zamestnanci ročník

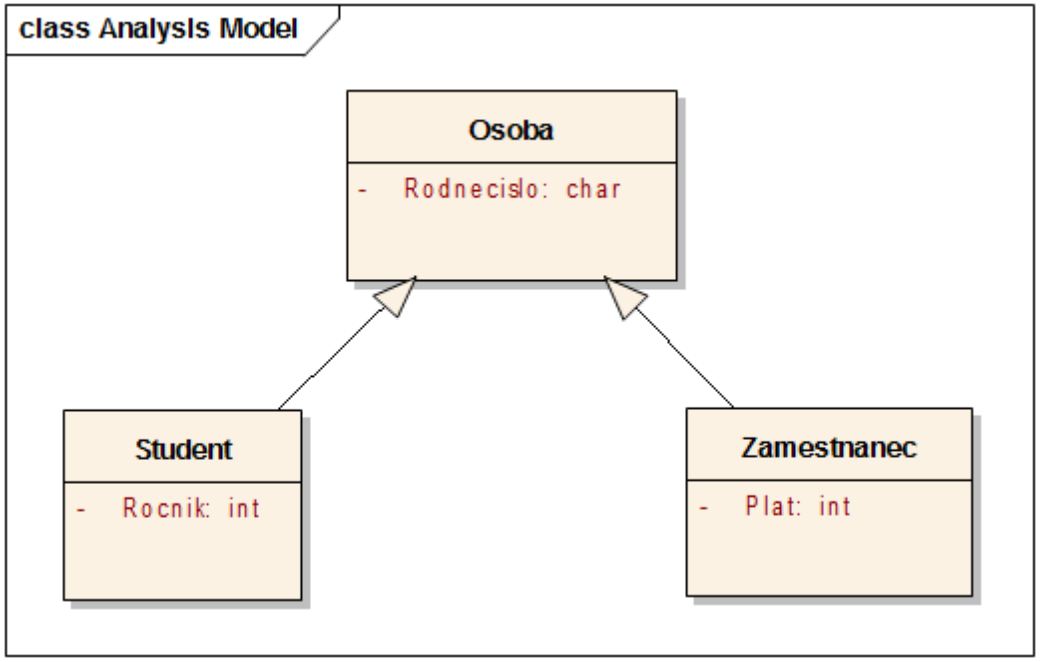


- Mohli by sme plat pre každého študenta nastaviť na 0, nebola by to však pravdivá informácia
- Kód bude obsahovať konštrukcie typu

```
If plat=0 Then {PracaSoStudentom}
Else {PracaSoZamestnancom}
```

Zmiešaná súdržnosť inštancií: riešenie

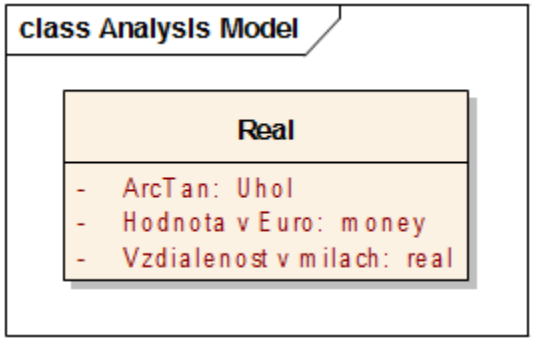
Vytvoríme dve nové triedy Student a Zamestnanec, pričom každá bude obsahovať informácie relevantné pre ten ktorý typ objektu



Zmiešaná súdržnosť domén

- **Definícia:** Trieda so zmiešanou súdržnosťou domén obsahuje nejaký prvok, ktorý priamo zaťažuje danú triedu nevlastnou triedou inej domény (resp. triedou z vyššej skupiny)
- Trieda B je **nevlastná** (extrinsic) triede A, pokiaľ je A možné úplne definovať bez odkazu na B
- Trieda B je **vlastná** (intrinsic) triede A, pokiaľ B sa vyskytuje v definícii A. Trieda B teda zachytáva nejaké vlastnosti A
- Trieda Slon je napríklad nevlastná triede Osoba. Trieda Dátum je triede Osoba vlastná

Zmiešaná súdržnosť domén: príklad



- Trieda real je triedou z domény základu.
- V tomto príklade je priamo zaťažená triedami Uhol a Money.

- Pretože triedu Real by bolo možné úplne zadefinovať bez odkazov na Uhol a Money, ide v tomto prípade o nevlastné triedy
- Pretože triedy Uhol a Money sú zo skupiny sémantických tried, má trieda Real zmiešanú súdržnosť domén

Typický príklad zmiešanej súdržnosti domén

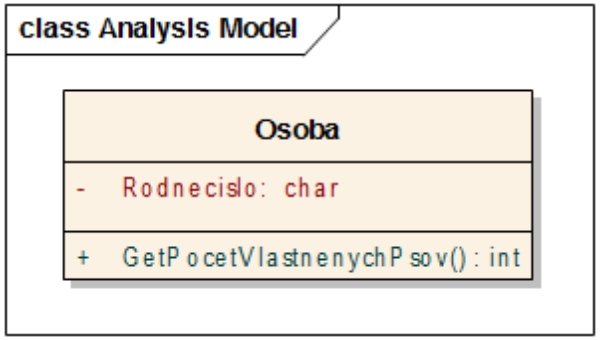
- Často používané „pravidlo“ OO návrhu: Vec by mala sama vedieť, ako niečo vykonať (Dokument by mal vedieť, ako sa sa má vytlačiť)
- To znamená, že dokument by mal mať znalosti o tlačiarni, komunikácii s ňou,...
- To sú však znalosti z domény architektúry a v triede Dokument nemajú čo robiť: chabý dizajn!

Zmiešaná súdržnosť rolí

Definícia: Trieda T so zmiešanou súdržnosťou rolí obsahuje nejaký prvok, ktorý priamo zaťažuje danú triedu nejakou nevlastnou triedou ležiacou v tej istej doméne ako T

Zmiešaná súdržnosť rolí: príklad

Trieda Osoba:



- nemá zmiešanú súdržnosť inštancií, pretože ak napríklad Jano nevlastní psov, tak je správne, že GetPocetVlastnenychPsov vrati 0
- nemá ani zmiešanú súdržnosť domén, Pretože Osoba aj Pes sú z tej istej domény
- napriek tomu, dizajn nie je dobrý, pretože Pes a Osoba sú rozdielne koncepty. Pes necharakterizuje osobu, trieda Pes je teda triede osoba nevlastná

Problémy dizajnu z predchádzajúceho príkladu

- Kde sa to celé zastaví? Čo keď budem chcieť vedieť, koľko Jano vlastní áut, mačiek,...?
- Najväčším problémom je však nízka znovupoužiteľnosť takto navrhnutej triedy Osoba. Ak by som ju chcel použiť v nejakej inej aplikácii, musel by som ju použiť aj so psami, autami a mačkami, aj keby to tú novú aplikáciu vôbec nezaujímalo
- Naším cieľom je vytvárať triedy s maximálnou možnou súdržnosťou!

Design for change

- Software a požiadavky naňho sa neustále menia
- Zmeny nie sú vždy predvídateľné
- Predvídateľné zmeny: napríklad existujúce požiadavky, ktoré nie je možné zrealizovať v rámci daného projeku
- Architektúra by mala byť navrhnutá tak, aby zmeny (predvídateľné aj nepredvídateľné) bolo možné „ľahko“ zrealizovať.
- Vedný odbor: Unanticipated Software Evolution

Design for change: ako na to?

- Stačí dodržiavať všetko čo sa naučíte v tomto predmete

Separation of concerns

- Princíp: rozdeľuj a panuj
- Rôzne aspekty daného problému je treba od seba oddeliť a každý čiastkový problém riešiť sám o sebe
- Rozdelenie architektúry na jednotlivé komponenty, pričom každý komponent rieši istú časť problému
- Toto je vlastne iný pohľad na súdržnosť

Information hiding

- The principle of information hiding is the hiding of design decisions in a computer program that are most likely to change, thus protecting other parts of the program from change if the design decision is changed.
- The protection involves providing a stable interface which shields the remainder of the program from the implementation (the details that are most likely to change).

Abstraction

- Abstrakcia znamená vynechanie niektorých aspektov modelovaného problému a zvýraznenie tých aspektov, ktoré sú pre nás dôležité
- Základný princíp softwarového inžinierstva
- Abstrakcia pomocou rozhraní:
  - Explicitné rozhrania
  - Oddelenie rozhraní a ich implementácií
  - Návrh pomocou kontraktov

Incrementability

- Softwarové systémy by mali vznikať inkrementálne
- Takýto postup umožňuje zvládať veľké a zložité systémy
- Pomerne rýchlo umožňuje predviesť čiastočné výsledky zákazníkovi a na základe jeho reakcie upraviť ďalší postup
- Využívanie prototypov

Traceability

- Dôležité na pochopenie architektúry nejakého systému
- Popisy implementovaných štruktúry systému (kódu) majú byť dostupné pomocou modelov uložených v inej forme než je samotný kód
- Má byť možné vidieť vzťahy medzi jednotlivými úrovňami modelov prvkov
- Požiadavka → Analytická trieda → Návrhová trieda → Kód → (chybové hlásenie)

Modularity

- Architektúra má pozostávať z dobre definovaných stavebných blokov (building blocks), ktorých funkčné zodpovednosti (responsibilities) sú jasne ohraničené
- Stavebné bloky by malo byť ľahké vymeniť
- Realizácia napríklad pomocou komponentnej technológie

Self-documentation

- Architekt alebo vývojár sú zodpovední za poskytnutie a uskladnenie všetkých informácií týkajúcich sa nimi vytváraných artefaktov a to v tej chvíli, keď daný artefakt vytvárajú
- Tento princíp umožňuje napríklad generovanie dokumentácie

Open-closed princíp

- Closed for change:
  - Existujúce triedy nebudeme meniť
- Open for extension
  - Zmeny budeme dosahovať pomocou dedičnosti a realizácie rozhraní

Princíp prispôsobenia typu

- Nech T je typ a P je jeho podtyp (P dedí z T). Potom P je prispôsobené T vtedy, ak objekt typu P môže byť poskytnutý v ľubovoľnom kontexte, kde je očakávaný objekt typu T. ***(Liskovovej substitučný princíp)***
- Princíp prispôsobenia typu platí pre všetky reprezentácie typu (teda tam, kde je očakávaný objekt typu T, môže byť poskytnutá ľubovoľná reprezentácia P.

Príklad

- Nech T je elipsa
- Nech P je kružnica
- Každý objekt, ktorý je kružnicou, je zároveň elipsou.
- Každá operácia, ktorá očakáva elipsu, by mala byť spokojná, keď dostane kružnicu

Kužnica versus elipsa

- Poznámka: o tom, či kružnica je elipsa sa v programátorskom svete vedú siahodlhé diskusie:
  - The Ellipse-Circle Dilemma
  - (http://ootips.org/ellipse-circle.html)
  - MORE ON TYPE INHERITANCE
  - (http://www.dbdebunk.com/page/page/1383837.htm)
- Táto diskusia sa týka hraníc použiteľnosti objektovej orientácie

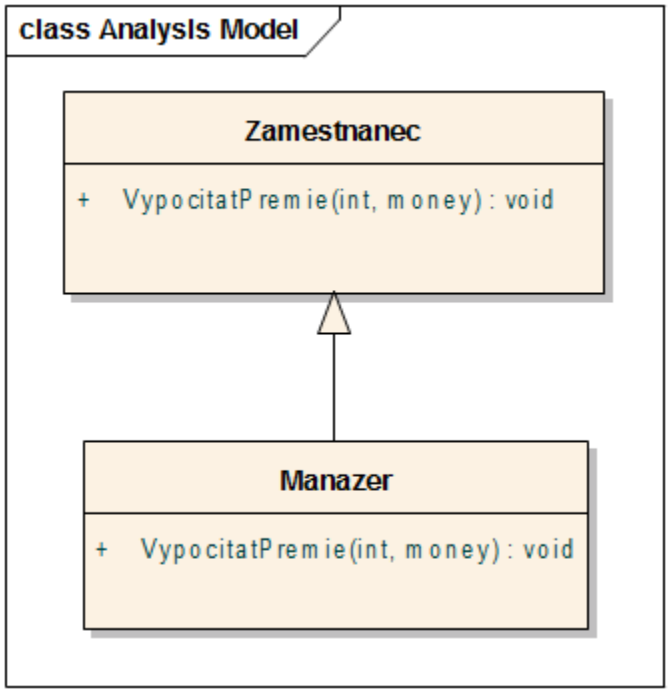
Pravidlo

V každom rozumnom objektovo orientovanom návrhu by mal byť každý typ prispôsobený svojmu nadriadenému typu

Princíp kontravariancie a kovariancie

- Typ je prispôsobený svojmu nadriadenému typu vtedy, ak spĺňa princípy kovariancie a kontravariancie
- Princíp kontravariancie: predbežná podmienka každej operácie typu nie je silnejšia než zodpovedajúca podmienka v nadradenom type
- Princíp kovariancie: Následná podmienka každej operácie je prinajmenšom taká silná ako zodpovedajúca podmienka v nadradenom type

Príklad kontravariancie a kovariancie

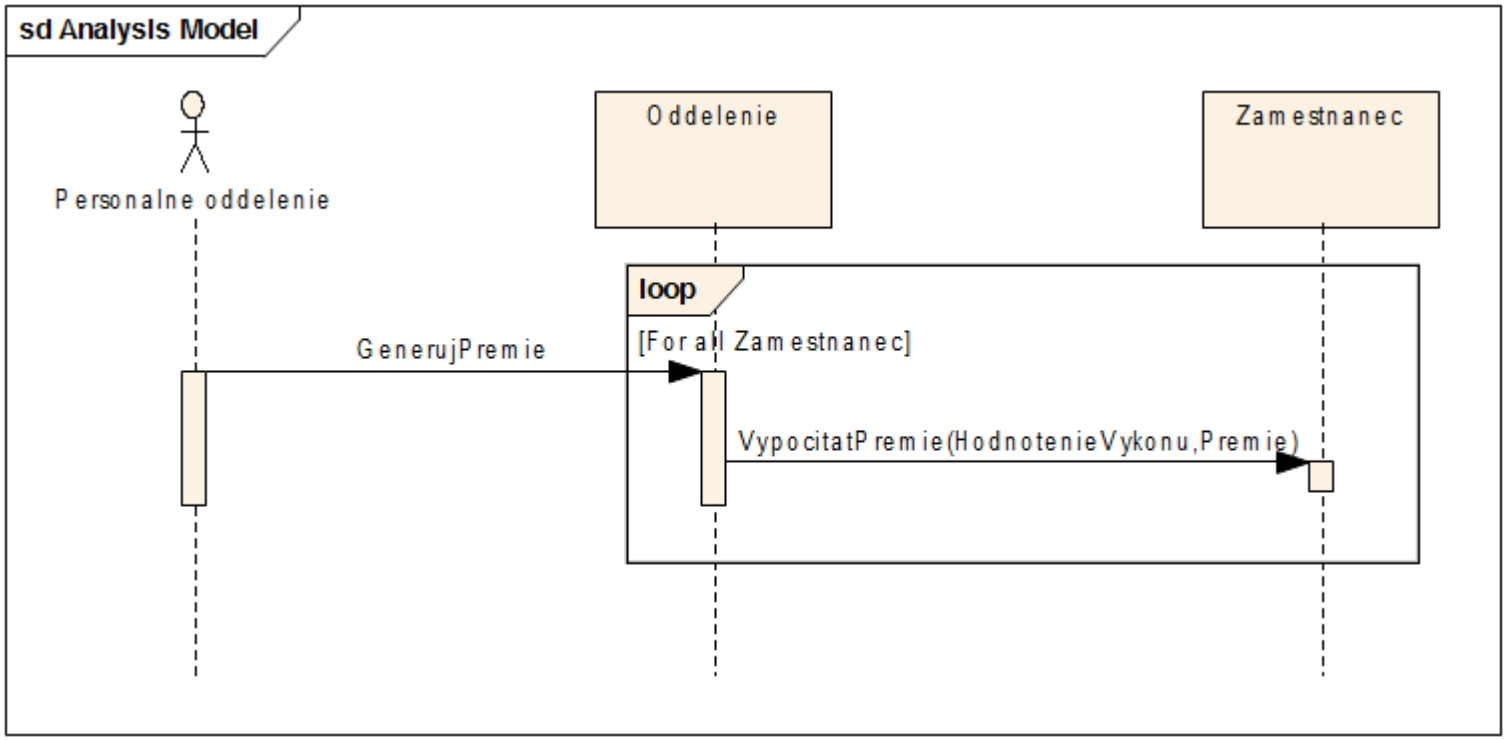


- VypocitatPremie:
  - Vstup: hodnotenie vykonu
  - Vracia: vysku premii
- Trieda Manazer prepisuje tuto metodu (premie pre manazerov sa pocitaju inak nez pre beznych zamestnancov)
- Hodnotenie vykonu zamestnancov: Hodnoty 1..5 (precondition)
- Precondition tejto (predefinovanej) operácie v type Manazer nesmie byť silnejšia než v type Zamestnanec; teda nesmela by byť napríklad 2..4. ale mohla by byť 0 až 7

Príklad (pokračovanie)

- Postcondition operácie VypocitatPremie pre typ Zamestnanec je, že návratová hodnota je z rozsahu KČ10000,- až KČ 30000,-
- Postcondition tejto operácie predefinovanej v triede Manazer musí byť silnejšia (alebo rovnako silná).
- Ak táto predefinovaná operácia bude vraciať hodnoty 25000 až 30000, je všetko v poriadku. Ak by vracala hodnoty 25000 až 35000, máme problém.

Príklad (vysvetlenie)



GenerujPremie bude zrejme prechádzať nejaký zoznam zamestnancov. Nevie, či ma práve do činenia s manažérom alebo s obyčajným zamestnancom

Vysvetlenie príkladu (pokračovanie)

- Ten, kto bude vyvíjať operáciu Oddelenie.GenerujPremie bude počítať s predbežnou podmienkou zamestnanca, t.j. Hodnotenie 1 až 5.
- Ak by predbežná podmienka Manazera bola silnejšia, to znamená 2 až 4 a Oddelenie by mu poslalo hodnotenie 1, vykonanie operácie by skrachovalo! Ak Manazer namiesto 1 až 5 akceptuje 0 až 7, nič sa nestane.

Vysvetlenie príkladu (pokračovanie 2)

- Oddelenie na druhej strane očakáva návratovú hodnotu 10000 až 30000.
- Ak by mal Manazer postcondition 25000 až 35000 a niektorému manažérovi by operácia vygenerovala odmenu 35000, Oddelenie by nevedelo čo s tým a zhavarovali by
- Ak má Manager rovnakú alebo silnejšiu postcondition (napr. 25000 až 30000) je všetko OK, lebo sú to hodnoty, ktoré oddelenie dokáže akceptovať

Požiadavky na prispôsobenie typu: zhrnutie

- Nech P je podtyp typu T a prepisuje jeho operáciu op
- V rozmeroch zdieľaných typmi P a T musí byť stavový proces P a T totožný alebo musí stavový priestor P ležať v medziach stavového priestoru T
- Signatúra P.op musí byť totožná so signatúrou T.op
- Precondition P.op musí byť rovnaká alebo slabšia než precondition T.op
- Postcondition P.op musí byť rovnaká alebo silnejšia ako postcondition T. op

Princíp uzavreného chovania

- Princíp prispôsobenia typu zaručuje robustné typové hierarchie pre operácie čítania (retrieval) teda pre operácie, ktoré nemenia stav objektu
- Ak budeme stav objektu meniť, potrebujeme ešte niečo navyše: princíp uzavreného chovania
- Tento princíp vyžaduje, aby chovanie zdedené podtypom P od typu T rešpektovalo invariant podtypu P
- Pozor: na to sa dá často myslieť až pri návrhu P (rešpektuje dedené chovanie môj invariant?)

Uzavrené chovanie (príklad)

- Typ Mnohouholník;
- Podtyp Trojuholník
- Operácia Presunúť bude zrejme rovnako dobre fungovať pre Trojuholník ako pre Šesťuholník
- Operácia PridajVrchol bude fungovať tiež, ale trojuholník už nebude trojuholníkom

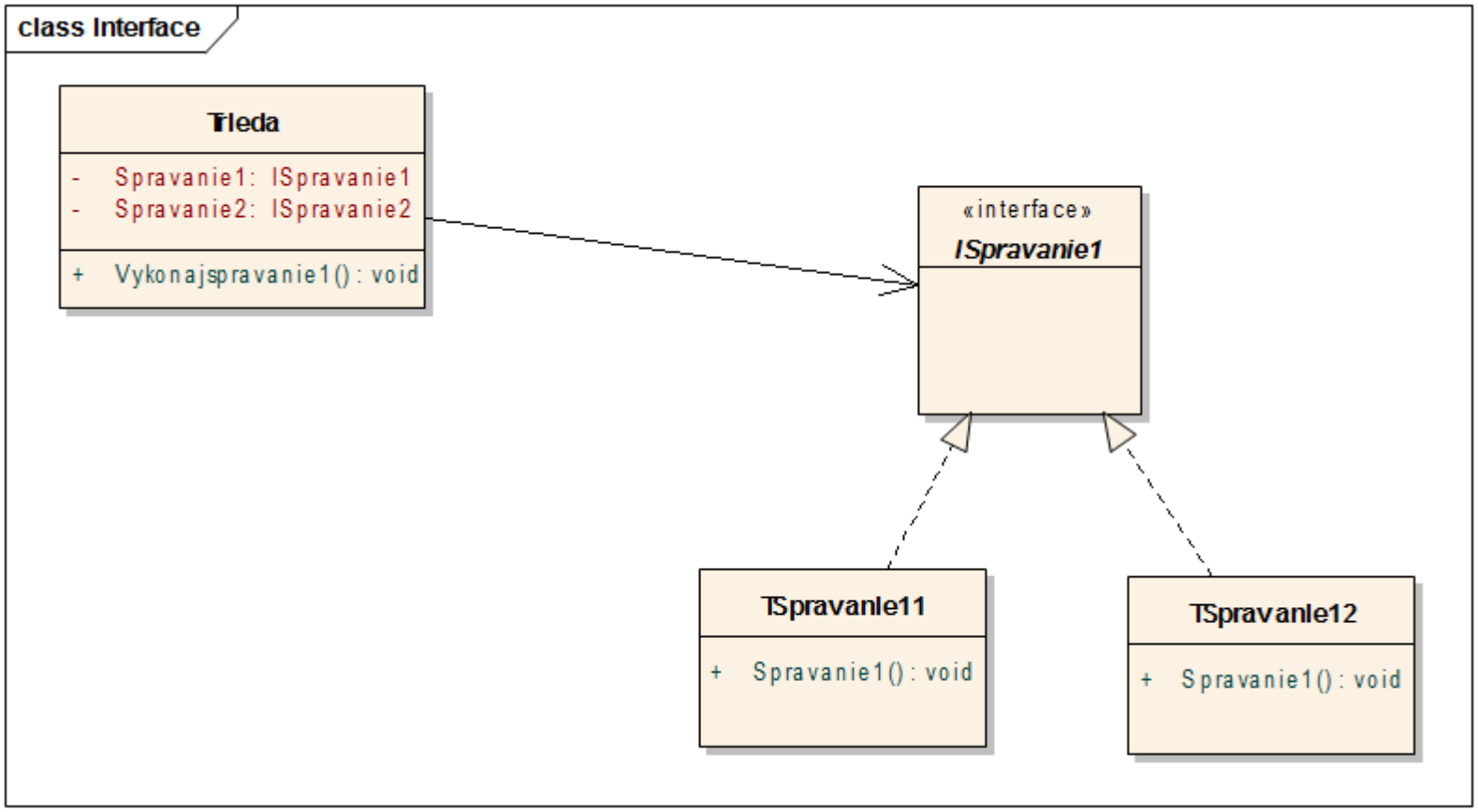
Rozhranie - definícia

- Rozhranie (interface) je špecifikácia správania
- Rozhranie môže byť implementované rôznymi triedami
- Rozhranie je kontrakt
- Tým, že triedy implementujú nejaké rozhranie, podporujú požadované správanie, čo umožňuje systému pracovať rovnakým spôsobom s objektami, ktoré inak nemajú nič spoločné
- Rozhranie je dynamickou alternatívou dedenia

Rozhrania: použitie

- Rozhrania sú základným prostriedkom realizácie princípu voľných väzieb (louse coupling)
- V dôsledku toho umožňujú znížiť zaťaženie triedy
- Triedy používajúce rozhrania nemajú pevné väzby. S konkrétnymi triedami sa previažu až vtedy, keď je to potrebné (počas behu programu?)
- Pomocou rozhraní môžeme až na poslednú chvíľu rozhodnúť, akú konkrétnu triedu použijeme

Rozhranie: UML reprezentácia a použitie



Rozhrania: vlastnosti

- Rozhrania špecifikujú správanie. Deklarujú metódy. A nič viac!
- Nemajú atribúty
- Nemajú inštancie

Architektúra a logika

Softwarový systém je formálny systém.

- Formálne systémy sú to, čo sú schopné počítače pochopiť a interpretovať
- Formálny znamená, že všetky pojmy, s ktorými pracujeme sú presne a jednoznačne definované
- Základom každého formálneho systému je logika. Logické chyby týkajúce sa pojmov sa teda týkajú samotnej podstaty a môžu mať veľké dôsledky
- Venujte preto pozornosť formalizmom, ktoré zavádzate. Keď si nenavrhnete dobre pojmy, tak nemôžete postaviť dobrý systém

- V súčasnosti sme v počítačovom priemysle bohužiaľ nútení pracovať s konceptmi obsahujúcimi veľké množstvo logických chýb
- Zrejme platí aj: *“Všetky rozdiely, ktoré sa netýkajú logiky, sú nepodstatné”*.
- Príklady logických rozdielov:
  - rozdiel medzi modelom a implementáciou
  - rozdiel medzi premennou a hodnotou
  - rozdiel medzi hodnotou a jej zobrazením

Princíp konceptuálnej integrity

- Dôležitým princípom návrhu systémov je konceptuálna integrita:

  – *„Je lepšie mať systém, z ktorého sú vynechané niektoré anomálie ale ktorý pritom reflektuje jednu množinu návrhových princípov, než systém, ktorý obsahuje mnoho dobrých ale nezávislých a nekoordinovaných myšlienok“ (F. Brooks: „The mythical man-month“)*

- Princíp ďalej hovorí:

  – *“Čistý a elegantný softwarový produkt musí prezentovať koherentný mentálny model. Konceptuálna integrita je najdôležitejším faktorom jednoduchosti používania systému a tvorí základ kvality produktu”*

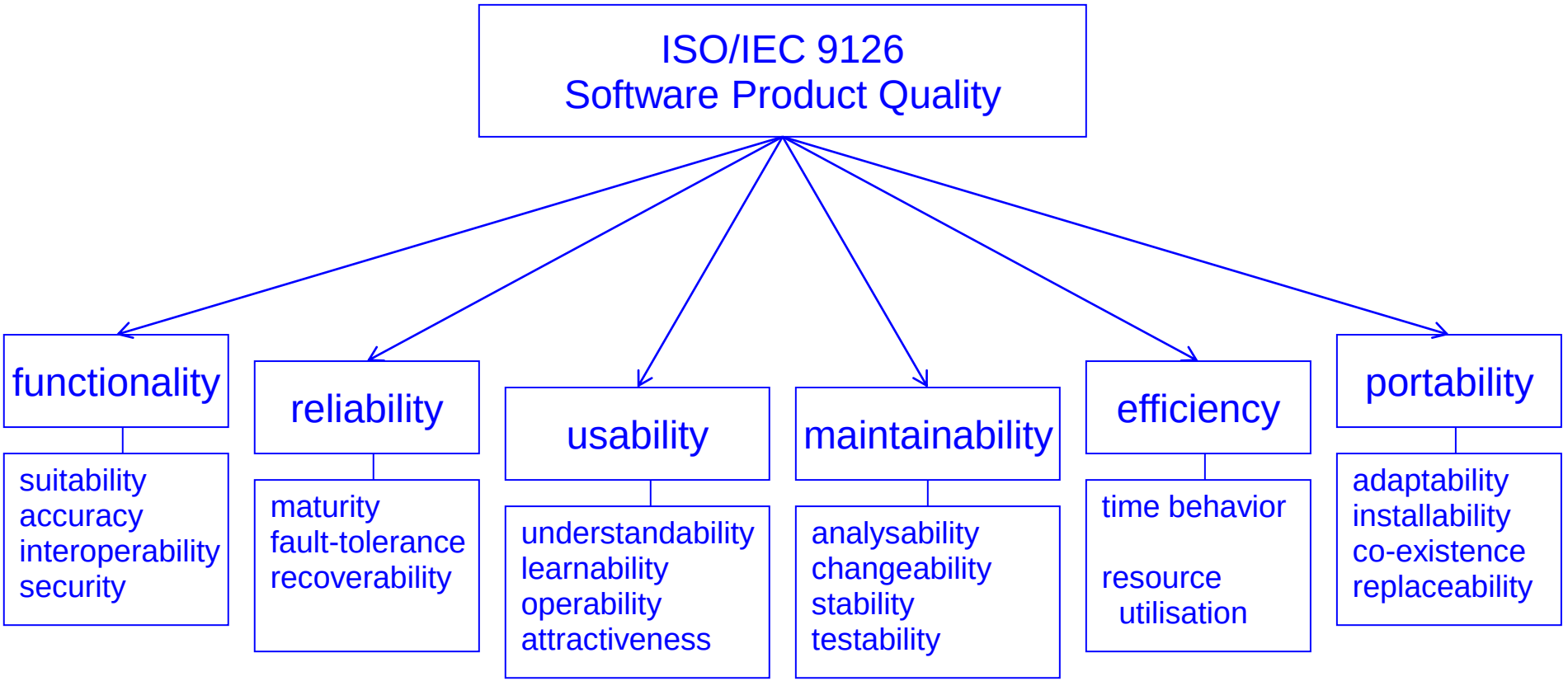
- Aby sme vôbec mohli hovoriť o konceptuálnej integrite, musíme mať samozrejme najskôr koncepty. Pre tieto koncepty platí, že:
  - Musia byť jasne pomenované a definované
  - Je lepšie, keď ich je menej, ale sú starostlivo zvolené
  - Musia s nimi súhlasiť všetci účastníci
  - Nesmú medzi nimi existovať vnútorné rozpory (musia byť konzistentné)
- Príkladom môže byť DBMS, ktorý tvrdí, že používa relačný model. Ak sa DBMS rozhodne tento model používať, tak ho musú používať úplne. Nesmie vynechať žiadnu vlastnosť tohoto modelu a ospravedlňovať takéto vynechanie napríklad zjednodušním syntaxe

- Jednou z charakteristík konceptuálnej integrity je pomerne malý počet konceptov
- Ten sa v dnešnej kultúre veľkého počtu možností bohužiaľ presadzuje pomerne ťažko

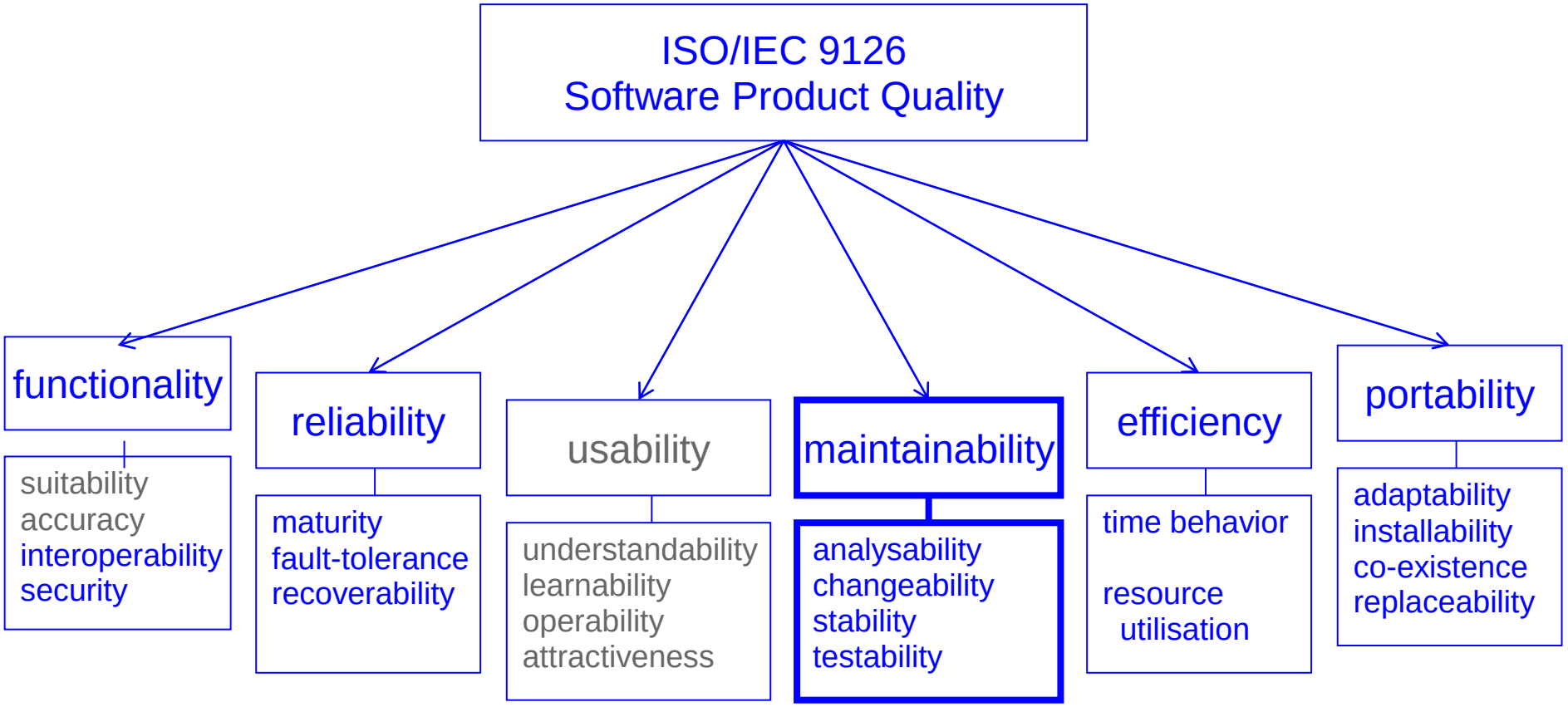
  –*nie je dôležité, akú majú veci kvalitu, hlavne keď ich je veľa*

- Zaujímavým príkladom toho ako konceptuálna integrita používaného nástroja vplýva na kvalitu výsledného produktu je meranie kvality softwaru, ktoré sa nedávno uskutočnilo v Ústredí pre výpočtovú techniku Belastingdienst/NL (B/CICT)
- Časť vytváraných softwarových produktov sa hodnotila z hľadiska nasledujúcich kritérií:

ISO/IEC 9126, Part 1
Product quality model: internal and external



ISO/IEC 9126, Part 1
Product quality model: technical quality



Mapping source code properties onto quality sub-characteristics

	Volume	Complexity	Unit size	Duplication	Unit testing	
Analysability	X		X	X		
Changeability		X		X		
Stability					X	
Testability		X	X		X	

Source code properties and metrics

Volume

- LOC, within the context of a single language
- Man years via backfiring function points

Complexity per unit

- McCabe's cyclomatic complexity, SEI risk categories, %LOC for each category

Duplication

- Duplicated blocks, threshold 6 lines, %LOC

Unit size

- LOC, risk categories, %LOC for each category

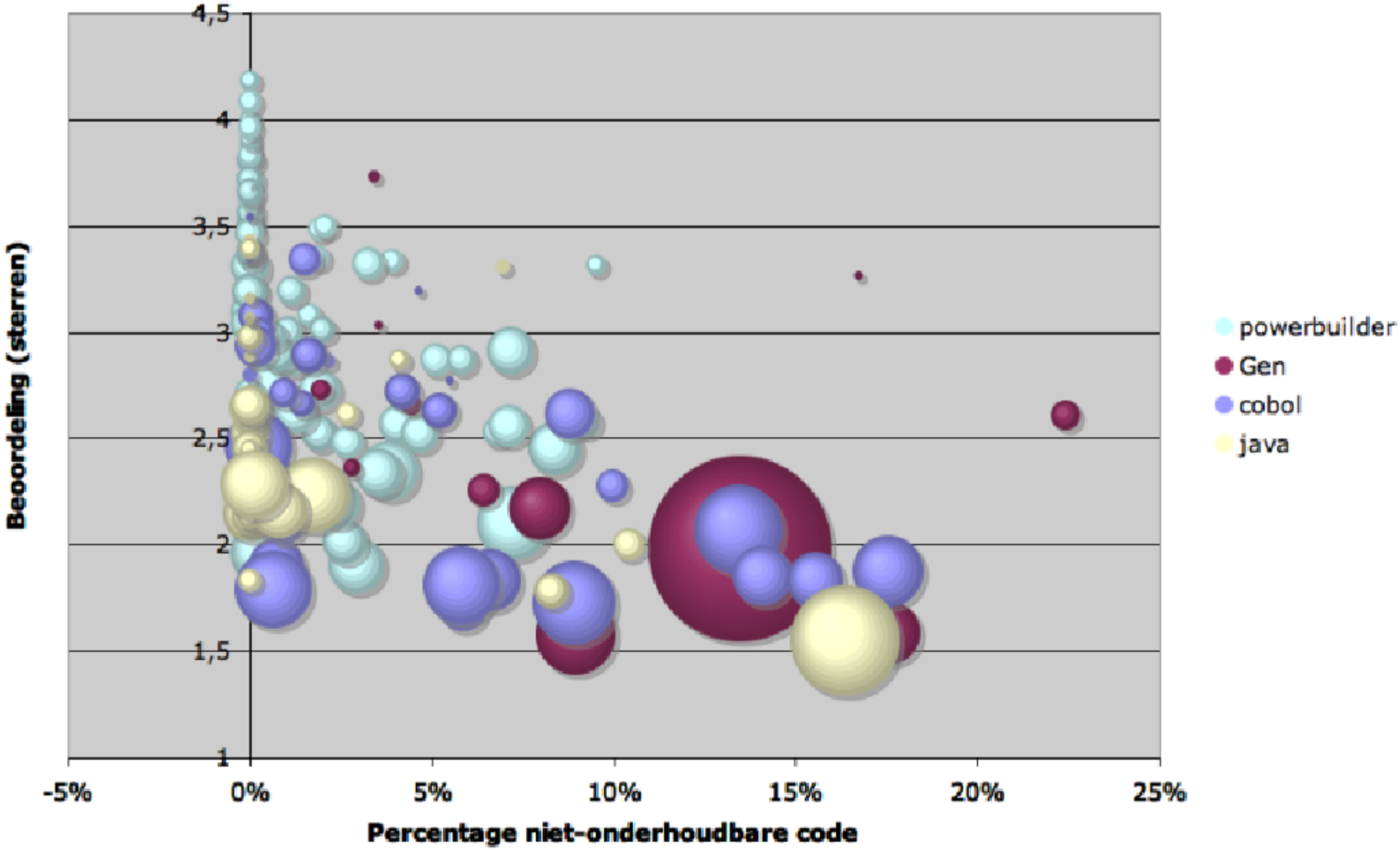
Unit testing

- Unit test coverage
- Number of assert statements (as validation)

Meranie

- U jednotlivých softwarových produktov sa posúdila kvalita podľa uvedených kritérií. Každý produkt sa vyhodnotil známkou 1 (najhoršia známka) až 5 (najlepšia známka)
- Nasledujúci diagram ukazuje výsledky merania v závislosti od veľkosti produktu (na diagrame reprezentovaná veľkosťou kolieska) a vývojového prostredia (farba)

Výsledky merania



Výsledky: jeden zo záverov

- Najlepšie výsledky dosiahli malé produkty vytvorené v prostredí PowerBuilder Sybase
- Nedá sa to dokázať, ale zrejme jedným z aspektov, ktoré prispeli k vysokému hodnoteniu je to, že sa jedná o malý nástroj, ktorý užívateľovi poskytuje pomerne málo možností ako dosiahnuť to, čo chce (malý počet konceptov), ale na druhej strane je dobre vymyslený
- Na rozdiel od toho dosiahli pomerne zlé výsledky produkty vytvorené v Jave!

- Jazyk UML je typickým príkladom nízkej konceptuálnej integrity
  - Obsahuje príliš veľké množstvo konceptov
  - Nezdôrazňuje logické rozdiely (žiaden z rozdielov, ktoré sme si uviedli u TTM)
  - Obsahuje mnoho pojmov pre tú istú vec (trieda, typ, rozhranie)
  - Koncept stereotypu, ktorý otvára dvere pre porušenie všetkých ostatných princípov

Návrh: dôležité aspekty

Motto: Ak sa niečo môže pokaziť, určite sa to pokazí

Logging

- V prípade chybových situácií budete musieť chybu analyzovať a odstrániť.
- Na to musíte presne vedieť, čo systém v momente chyby robil
  - Na ktorom mieste zdrojového kódu chyba nastala
  - S akými dátami, hodnotami parametrov, nasteveniami atď. systém pracoval
- Prostriedok: Log
- Log je textový súbor, do ktorého systém registruje to, čo robí

- Čo logovať?
  - Samozrejme nie každý riadok kódu
  - Dôležité udalosti. Tie miesta v kóde, ktoré reprezentujú dôležité momenty transakcií
    - Prístupy do databázy
    - Začiatky a konce funkcií a procedúr
    - Začiatky a konce spracovania daného objektu (napr. začiatok a koniec práce s daným zákazníkom)
    - Musí byť jasné, ktorá verzia softwarového modulu chybu spôsobila

Logy archivujte

- Ak Vás zákazník dá na súd, tak sa hodí mať:
  - Dáta
  - Informácie o tom, čo sa s tými dátami robilo. Tie sú prístupné jedine v logoch
- Dáta a logy sa bežne archivujú do doby premlčania (zhruba 5 rokov)

Buďte defenzívni

- Keď nastane chyba, musí sa:
  - Transakcia prerušiť. Nepokračovať ako keby sa nič nedialo!
  - Informujte užívateľa (a správcu systému) pomocou jednoznačnej a pochopiteľnej hlášky
  - Pokiaľ je to možné, mal by sa vykonať automatický rollback
  - Pokiaľ to nie je možné, musí sa správcovi systému poskytnuť dostatok informácií pre ručný rollback

Príklad 1

- Kupovanie letenky:
  - Nastane chyba pri vykonávaní on-line platby
    - V dôsledku straty spojenia zákazníka s internetom
  - Rezervácia sa musí zrušiť
  - Je slušné poslať zákazníkovi mail, že transakcia bola zrušená
  - Je dobre zapamätať si na chvíľu (1 deň), čo zákazník robil, aby mohol rýchlo pokračovať, až sa znova prihlási

Podpora logov v .NET

Príklad 2

- Nastane chyba pri importe dát z externého súboru
- Polovica dát sa naimportovala, v polovici sa zistí, že údaj, ktorý sa má naimportovať, nespĺňa kontroly (dátum 33.02.2011)
- Možnosti:
  - Rollback celého importu
  - Naimportované dáta nechať naimportované. V prípade reštartu však import v tomto prípade musí kontrolovať, či dáta už existujú

Error handling

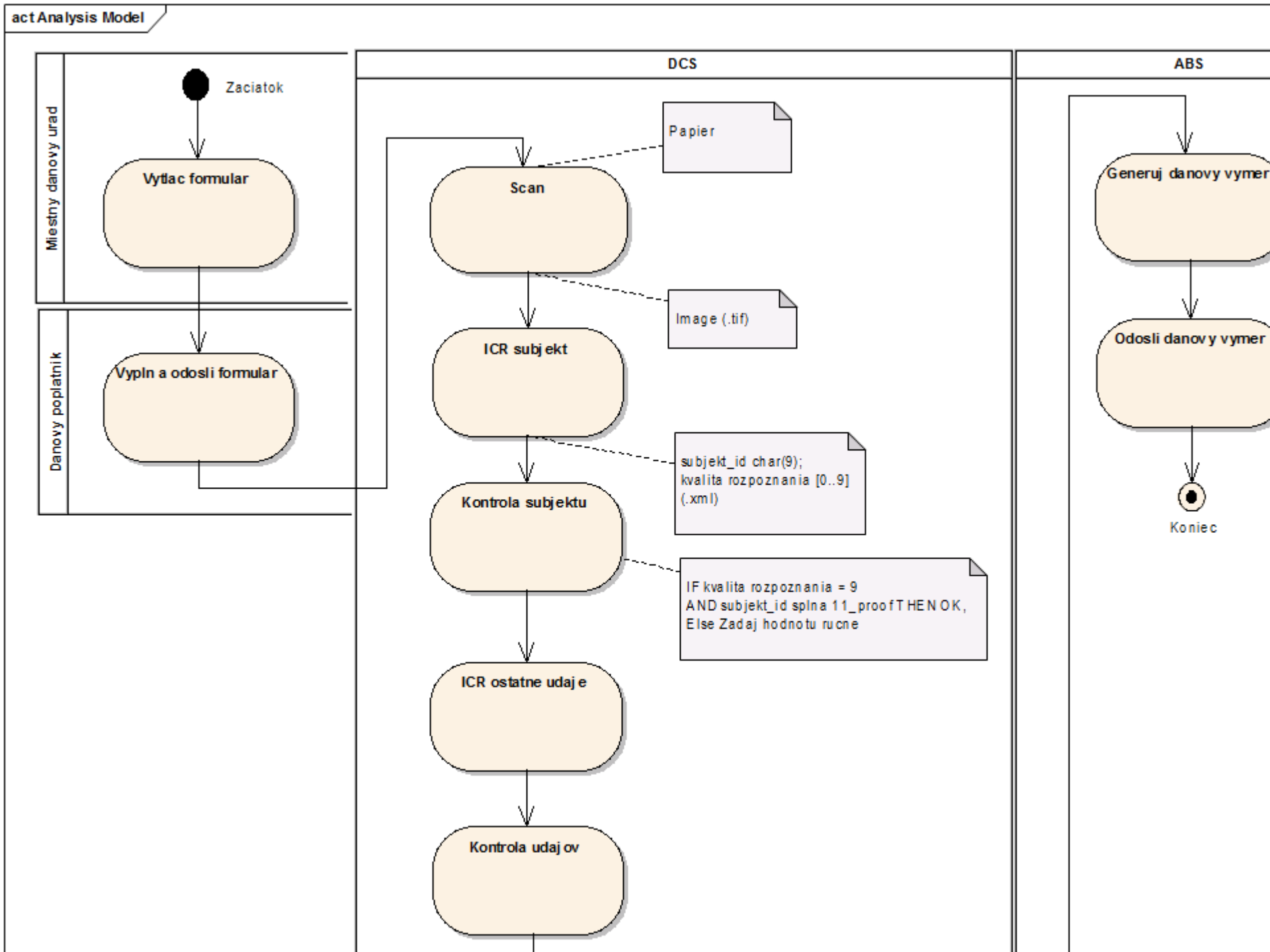
- Vytvorte si zoznam chýb, ktoré môžu nastať, priraďte im jednoznačné kódy a hlášky. Celý tím musí používať rovnaký zoznam
- Dohodnite sa na spôsobe zachycovania a riešenia chybových situácií (error handling). Zdokumentovať v coding guidelines

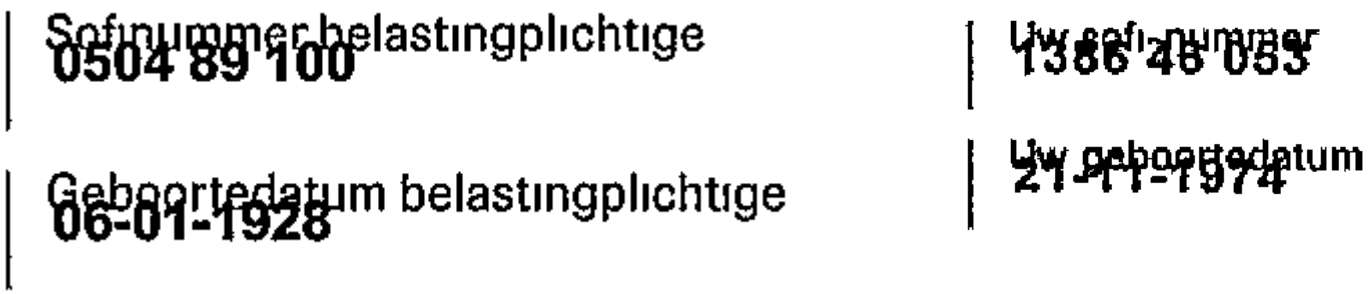
Reštartovanie transakcií

- Transakcie prerušené v dôsledku chýb musí byť možné reštartovať
- To je ľahké, ak sú dobre zorganizované rollbacky

Kontrolovanie preconditions

- Dobrým prostriedkom pre zvýšenie kvality softwaru je kontrola preconditions
  - Aké dáta je proces schopný spracovať
    - Typové kontroly
  - Aký je stav spracovania na začiatku transakcie
    - Nepustíme sa do predajnej transakcie pokiaľ nemáme overenú platnosť kreditnej karty zákazníka
  - Logické kontroly
    - Overíme, či človek, ktorému sa chystáme vyrúbiť daň, ešte žije


act Analysis Model
Miestny danovy urad
Danovy poplatnik
Zaciatok
Vytlac formular
Vypln a odosli formular
DCS
Papier
Scan
Image (.tif)
ICR subjekt
subjekt_id char(9);
kvalita rozpoznania [0..9]
(.xml)
Kontrola subjektu
IF kvalita rozpoznania = 9
AND subjekt_id splna 11_proofTHEN OK,
Else Zadaj hodnotu rucne
ICR ostatne udaje
Kontrola udajov
ABS
Generuj danovy vymer
Odosli danovy vymer
Koniec



- Rozpoznávací software rozpozná tieto sofinummer ako 099959999 a je si na 100% istý, že túto hodnotu rozpoznal dobre
- Rozpoznaná hodnota 099959999 spĺňa kontrolu 11-proof
- 099959999 je skutočné a existujúce sofinr, patrí však človeku, ktorý zomrel v januári 1994

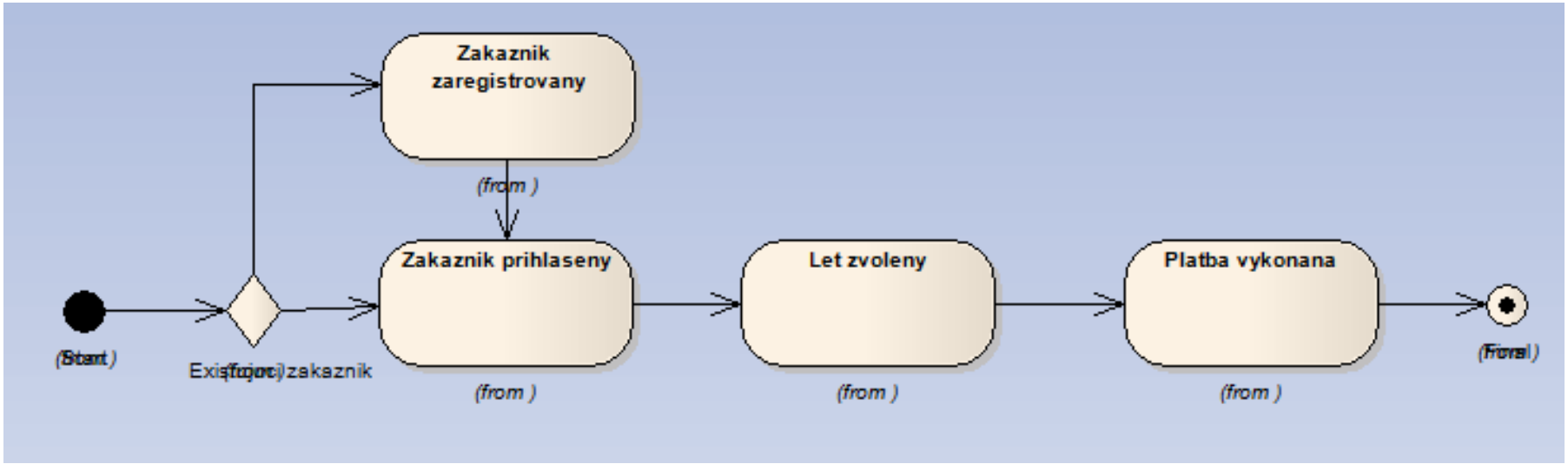
Stavové stroje

- V jednoduchej forme umožňujú kontrolu stavu spracovania
- V pokročilej forme umožňujú automatizované riadenie tokov spravovania (workflow engine)
  - Navrhnúť takéto stroje nie je jednoduché
- Obmedzíme sa na stavové stroje v jednoduchej forme

Stavy objektov

- Dôležité momenty spracovania si označte ako stavy
- Spracovanie objektu sa dá vnímať ako séria stavov
- Aby sa mohla vykonať transakcia, ktorá dostane objekt do stavu x, musí byť objekt v stave x-1
  - Toto sa dá na začiatku transakcie dobre skontrolovať

Príklad: kupovanie letenky



Výhody a nevýhody stavových automatov

- Výhody
  - Dajú sa graficky zobraziť
  - Jendoducho sa dajú rozšíriť o nový stav
- Nevýhody
  - Ak je objekt v stave x, tak predpokladáme, že prešiel stavmi x-1, x-2 a.t.ď
  - To však platí len vtedy, keď všetky procesy konsekventne kontrolujú predchádzajúce stavy
    - A to samozrejme nemusí byť pravda
  - Iná nevýhoda: nie všetky spracovania sú prísne sekvenčné. V takýchto prípadoch sa stavové automaty môžu rýchlo stať zložitými a neprehľadnými
    - Vznikajú konštrukcie typu „Ak je objekt v stave1 alebo stave2 a zároveň v stave3 potom...“

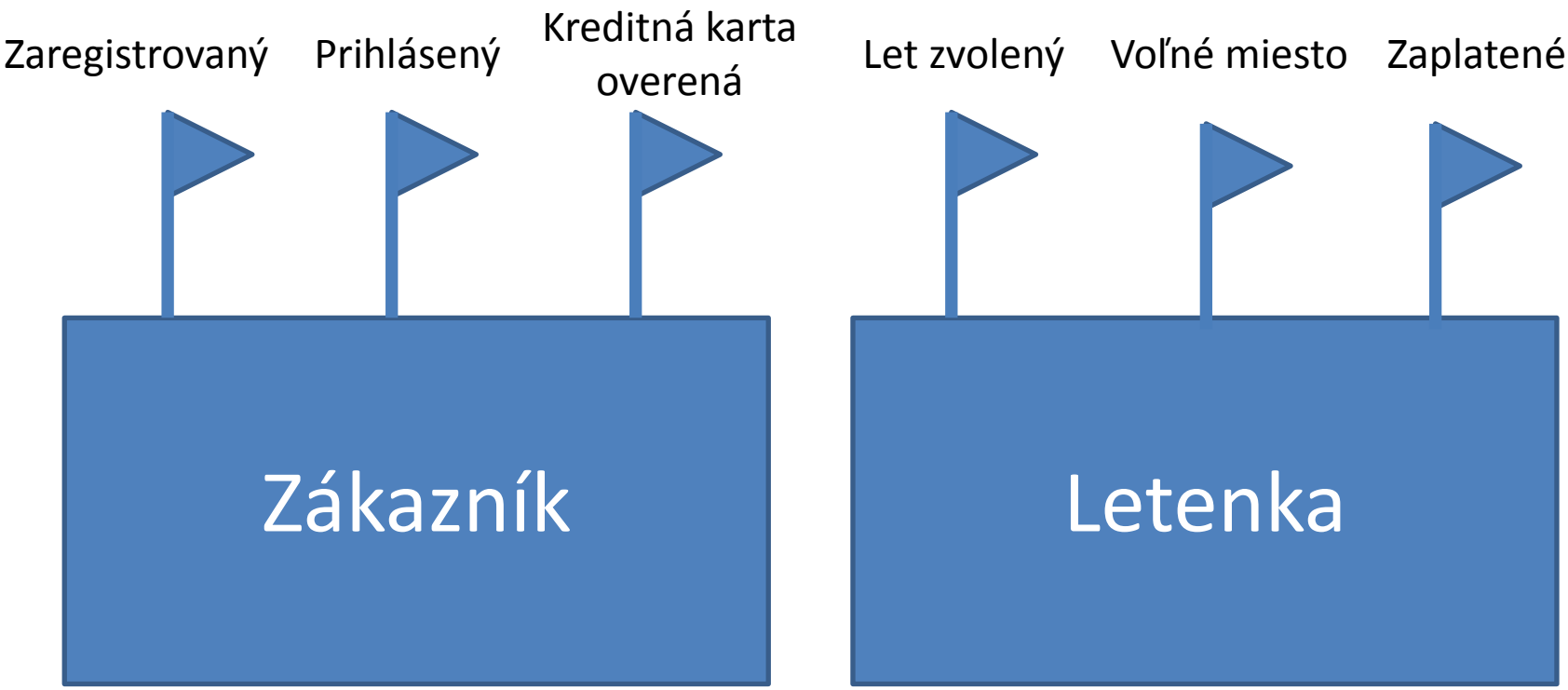
Iná možnosť: „vlajočky“ *(česky: praporky)*

- Objektom, ktoré prechádzajú spracovaním, sa priradí sada binárnych indikátorov (vlajočiek)
- Každá vlajočka má stavy „dolu“ a „zdvihnutá“
- Každá vlajočka reprezentuje nejakú transakciu
- Každá transakcia definuje, ktoré vlajočky musia byť v polohe „zdvihnutá“ na to, aby sa transakcia mohla uskutočniť
- Na začiatku spracovania sú všetky vlajočky dolu

Vlajočky: výhody a nevýhody

- Výhody:
  - Každá transakcia má úplný prehľad, čo sa s objektom dialo
  - Transakcia nie je závislá od toho, ako kontrolujú spracovanie iné transakcie
  - Väčšia flexibilita pri modelovaní (každá kombinácia vlajočiek reprezentuje nejaký stav)
- Nevýhoda:
  - Keď chceme pridať transakciu, musíme pridať aj zodpovedajúcu vlajočku = zmeniť model!

Vlajočky: príklad



K transakcii Zaplatiť sa môže pristúpiť až vtedy, keď sú vlajočky Zákazník.Prihlásený, Zákazník. Kreditná karta overená, Letenka.Let zvolený a Letenka.Voľné miesto v polohe „Zdvihnutá“

Metainformácie

- Uskladňujte si metainformácie
  - Metainformácie sú informácie týkajúce sa nie objektu samotného, ale jeho spracovania
  - Príklady metainformácií
    - Jednoznačný identifikátor
    - Kľúč pre vyhľadávanie v archíve
    - Kedy sme objekt vytvorili
    - Kedy sme objekt zaarchivovali
    - Kedy sme ukončili spracovanie
    - V prípade ručného spracovania: Kto objekt menil a kedy

- Metainformácie ukladajte a archivujte rovnako ako objekty samotné
- Niekedy sa vyžaduje, aby metainformácie boli uložené na inom mieste než objekty samotné

Správa verzií a konfigurácií

- SCCM: Software Configuration and Change Management
- Často sa pod týmto súhrnným názvom myslí predovšetkým správa verzií zdrojového kód
- Patrí tu však aj správa konfigurácií

Konfigurácia

- Sada artefaktov, ktoré spoločne predstavujú softwarový produkt vo forme, v ktorej je nainštalovaný u užívateľa
  - Programy samotné
  - Knižnice, assemblies
  - Konfiguračné súbory a databázy
  - Nastavenia registra
  - Autorizácie a prístupové práva

Význam správy konfigurácií

- Bez dobre zorganizovanej správy konfigurácií nebudete schopní podporovať zákazníka pri používaní softwaru
  - Odstraňovať chyby
  - Inštalovať nové verzie
  - Optimalizovať výkonnosť

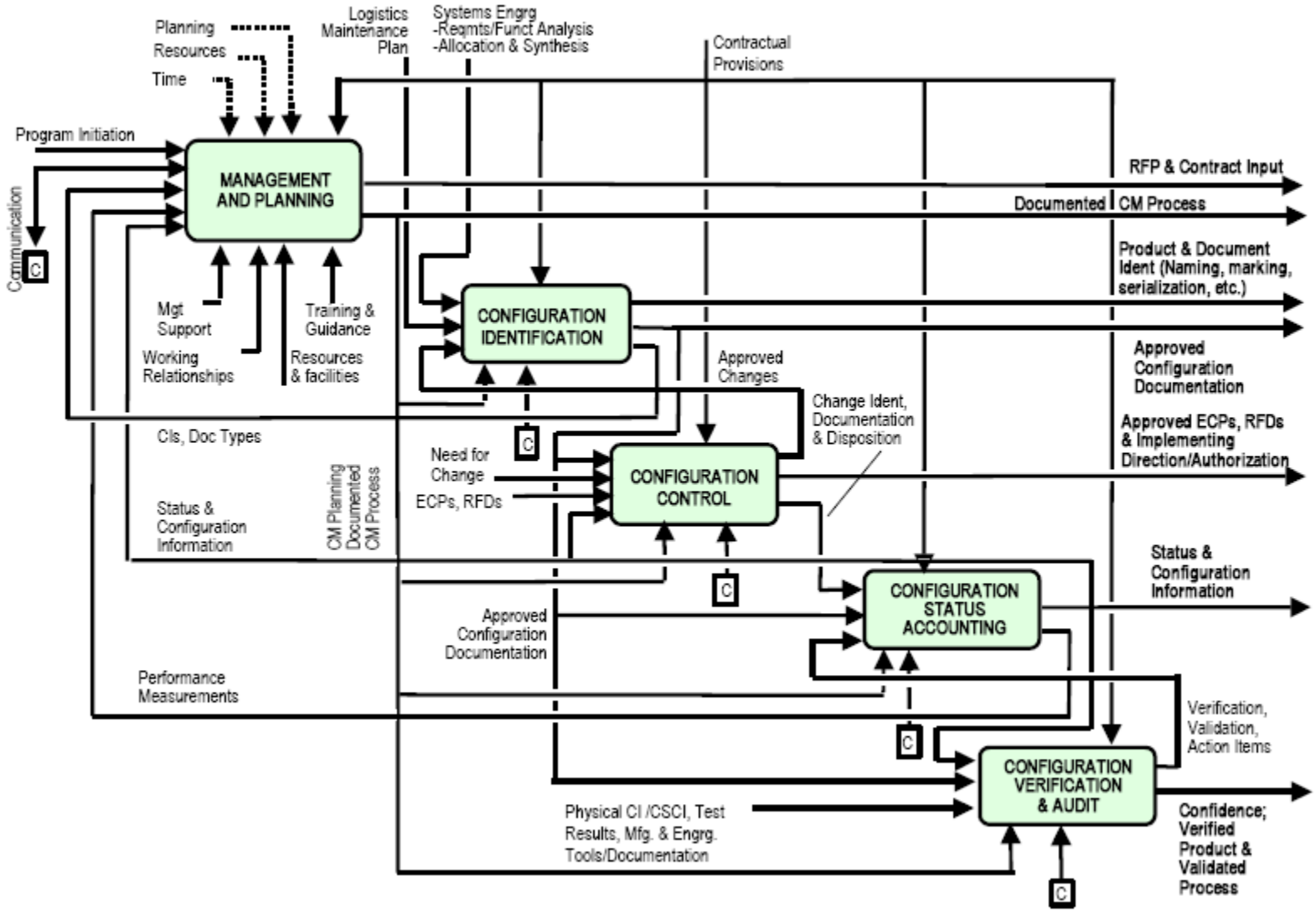
Ako na to?

- Existujú o tom celé knihy
- Pre bežné systémy však postačuje jednoduchá administrácia:
  - Každý zákazník používa niektorú verziu produktu (release)
  - Pre každú používanú verziu si zostavte (a bezpečne uložte) zoznam všetkých artefaktov a ich verzií, z ktorých je daný release zostavený
  - Ak sa konfigurácia zmení, máme vždy nový release!

Metodika UCM

- UCM: Unified Change Management
- Kľúčovým slovom je CHANGE
- Metodika a sada nástrojov pre podporu správy verzií a konfigurácií
  - Predovšetkým pre správu verzií, princípy sú však dobre použiteľné aj pre správu konfigurácií
- Nástroje ClearCase a ClearQuest

Change management vo veľkom

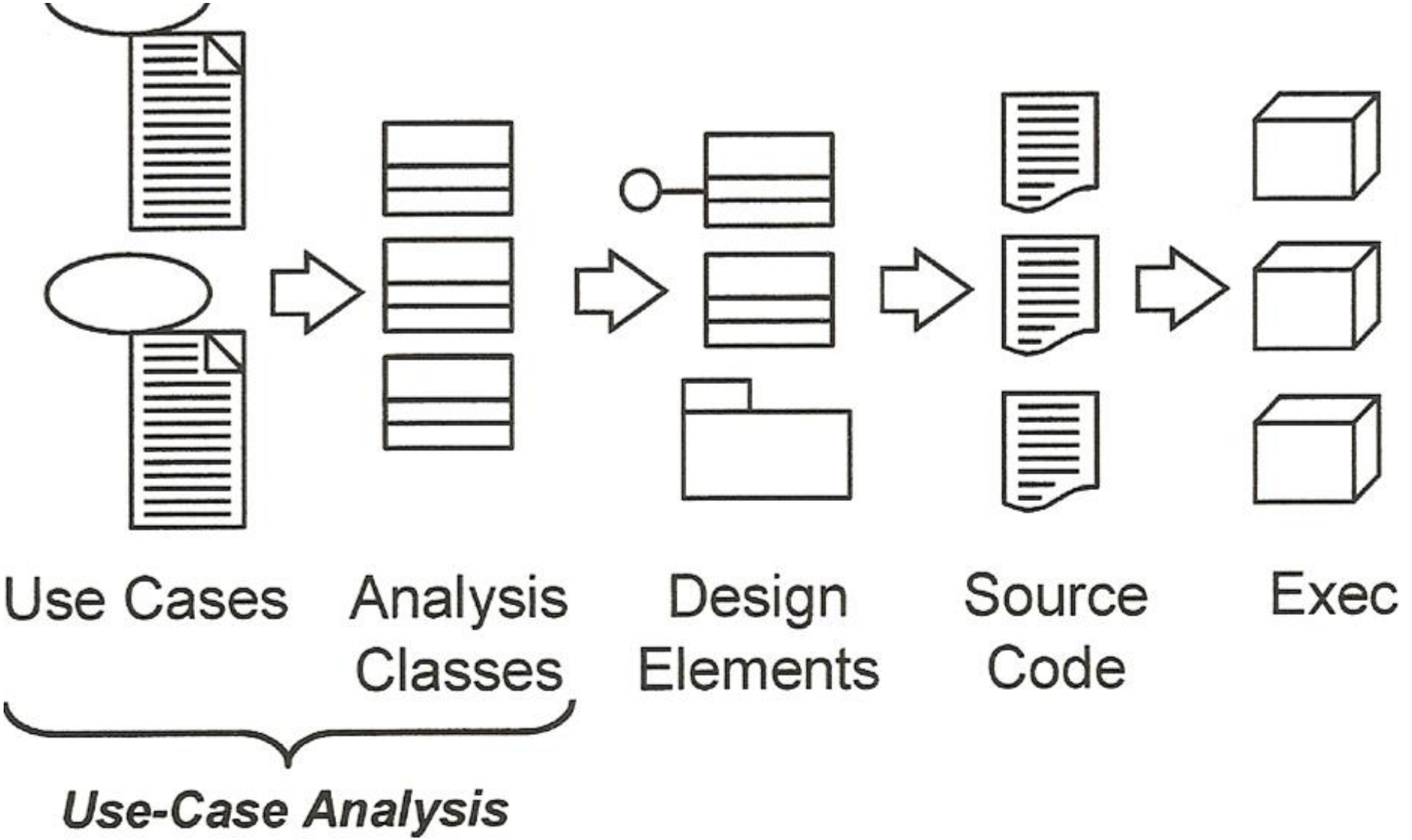


Model View Controler
a návrh pomocou rozhraní

Analýza prípadov užitia: opakovanie

- Pre každý prípad užitia:
  - Na základe popisu prípadu užitia vyhľadáme analytické triedy (napríklad pomocou metódy CRC)
  - Priradíme správanie popísané v prípade užitia jednotlivým triedam
- Pre každú nájdenú triedu
  - Popíšeme zodpovednosť triedy (responsibilities)
  - Definujeme atribúty a vzťahy

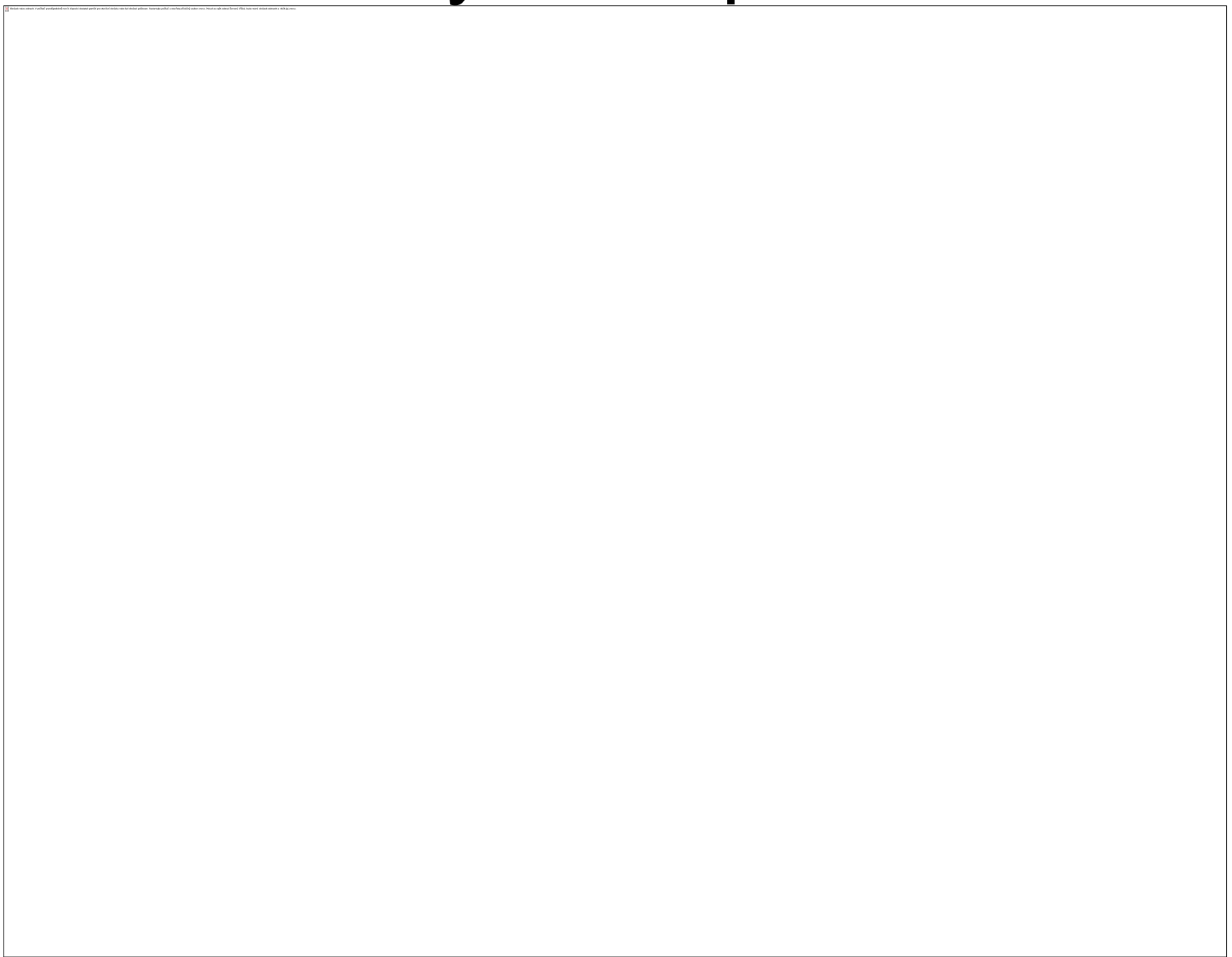
Analytické triedy v kontexte procesu vývoja



Tri druhy tried

- Pri analýze prípadov užitia môžeme nájsť tri druhy tried, ktoré reprezentujú:
  - **Hranice** medzi systémom a jeho užívateľmi
  - **Informácie,** ktoré systém používa a spracováva
  - **Riadiacu logiku** systému

Tri druhy tried: príklad 1



Prípad užitia „Registrácia odpracovaných hodín“

Zamestnanec má možnosť zadať do systému koľko hodín odpracoval za uplynulý týždeň na jednotlivých projektoch. Tok udalostí:

1. Systém zobrazí aktuálnu časovú kartu užívateľa. Ak karta neexistuje, systém ju vytvorí
2. Systém vyberie z databázy projektov informácie o projektoch
3. Užívateľ vyberie projekt, na ktorom pracoval a zadá počet odpracovaných hodín v jednotlivých dňoch
4. Systém uloží zadané informácie

Čo sú to návrhové triedy

- Návrhové triedy sú triedy, ktorých špecifikácie sú na takej úrovni, že trieda môže byť implementovaná
- Pochádzajú z dvoch zdrojov:
- Z problémovej domény (z analytického modelu prostredníctvom upresňovania analytických tried)
- Z domény riešenia (knižnice už existujúcich tried a komponentov)

Class model prípadu užitia „Maintain Timecard“

Class model internetového obchodu

Tri druhy tried: vzor MVC

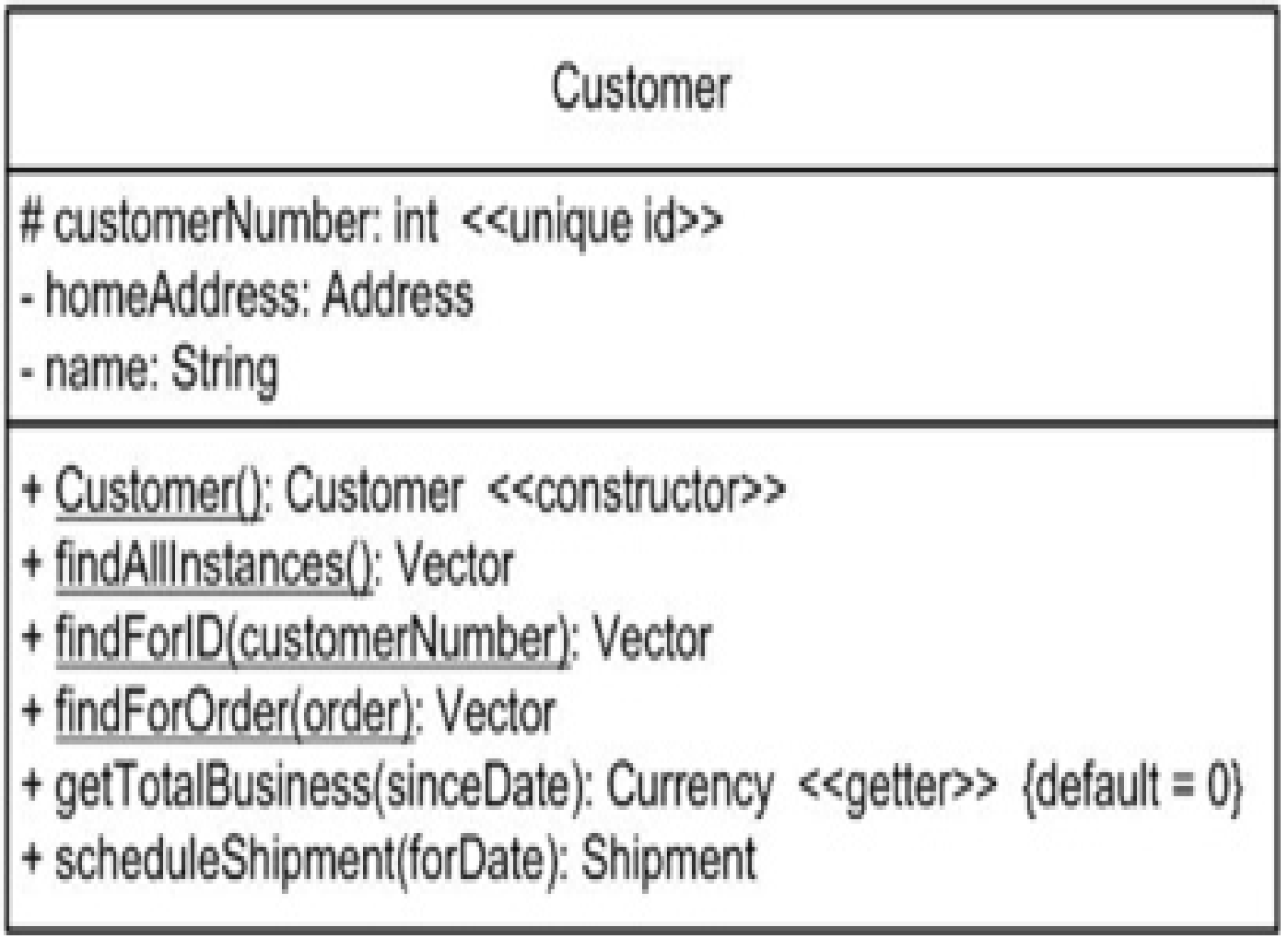
- Rozdelenie nájdených tried na boundary, control a entity je vzor MVC
- M: Model
- V: View
- C: Controler

Odbočka: stereotypy v UML

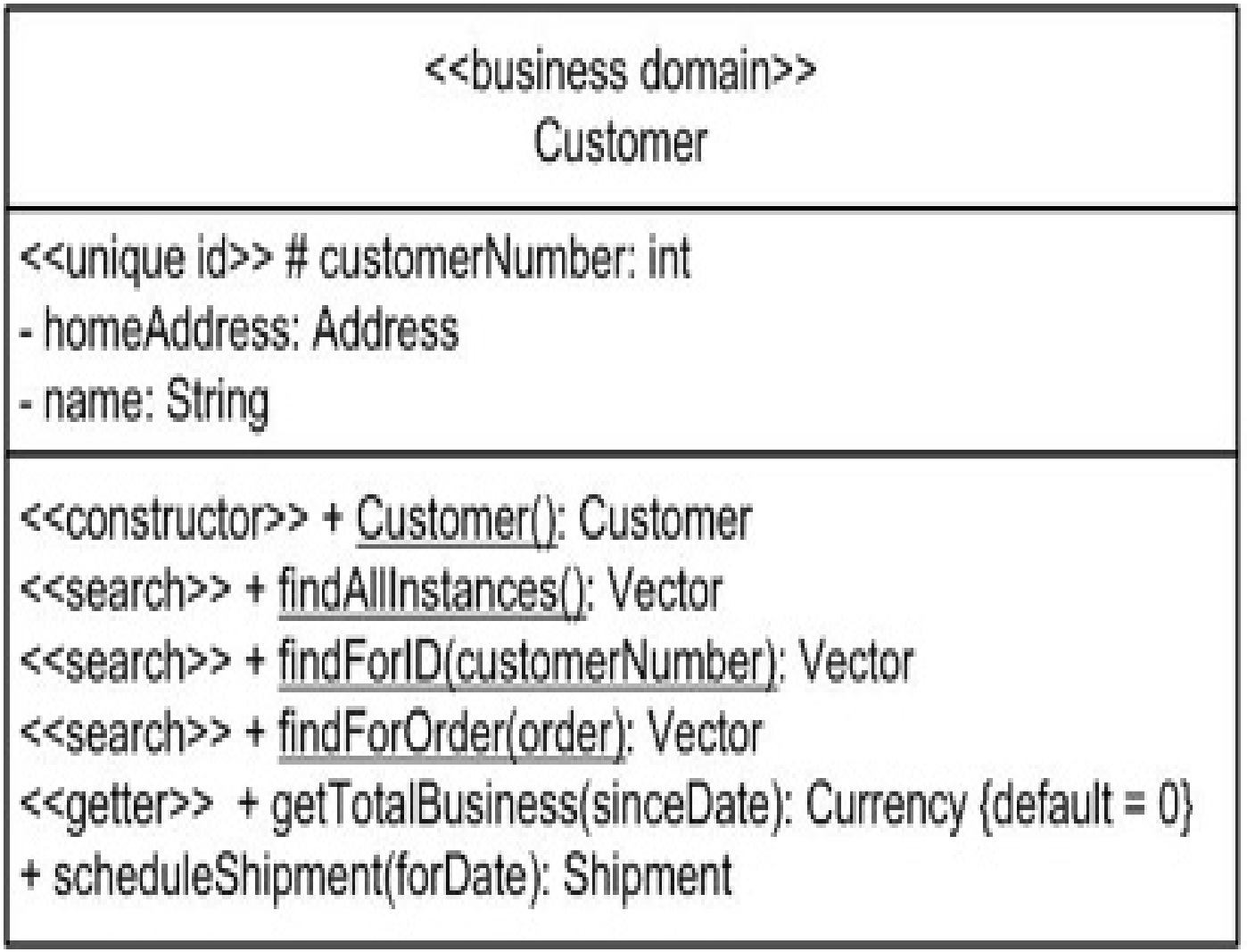
- „Rozširovací“ mechanizmus jazyka UML
- Objekty jazyka je možné presnejšie klasifikovať pomocou stereotypov
- UML pozná štandardné stereotypy. Užívateľ má možnosť definovať svoje vlastné stereotypy
- Stereotypy sa udávajú pomocou notácie nazývanej „tagged value“:

<<stereotyp>>

Stereotyp: príklad



Stereotyp: príklad 2



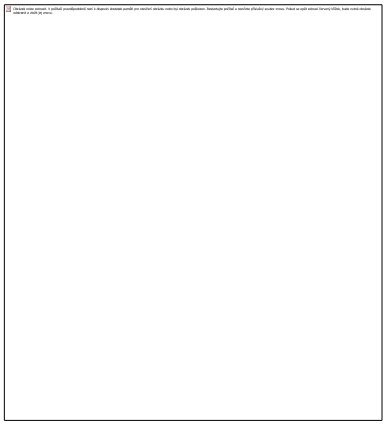
Klasifikácia tried pomocou stereotypov

- Na priradenie tried do jednej zo skupín MVC sa používajú stereotypy:
  - `<<boundary>>`
  - `<<control>>`
  - `<<entity>>`

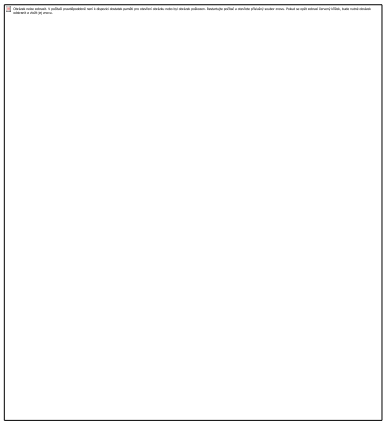
Príklad

„Ikonické“ zobrazenie

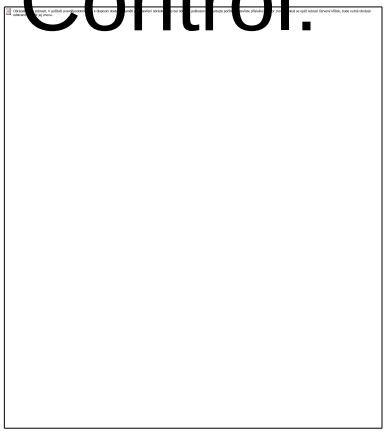
Boundary:



Entity:



Control:

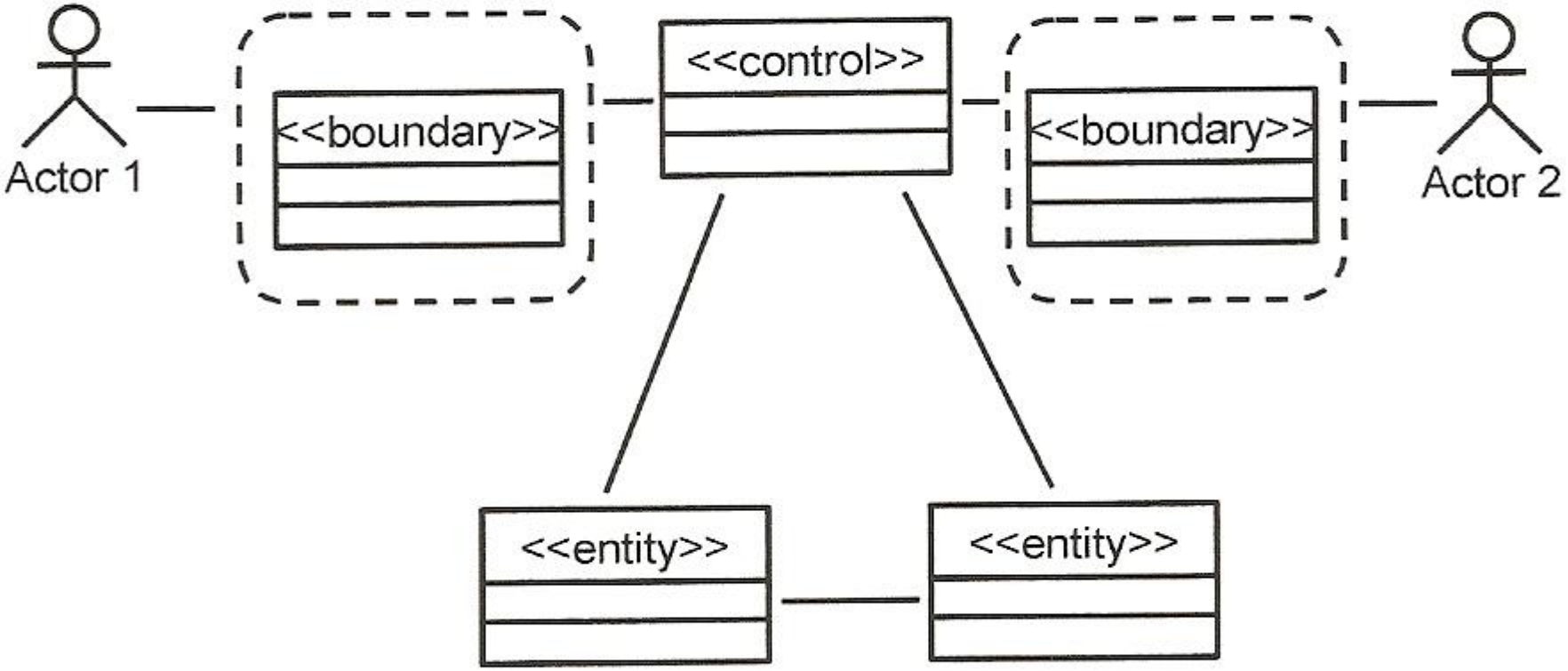


Príklad

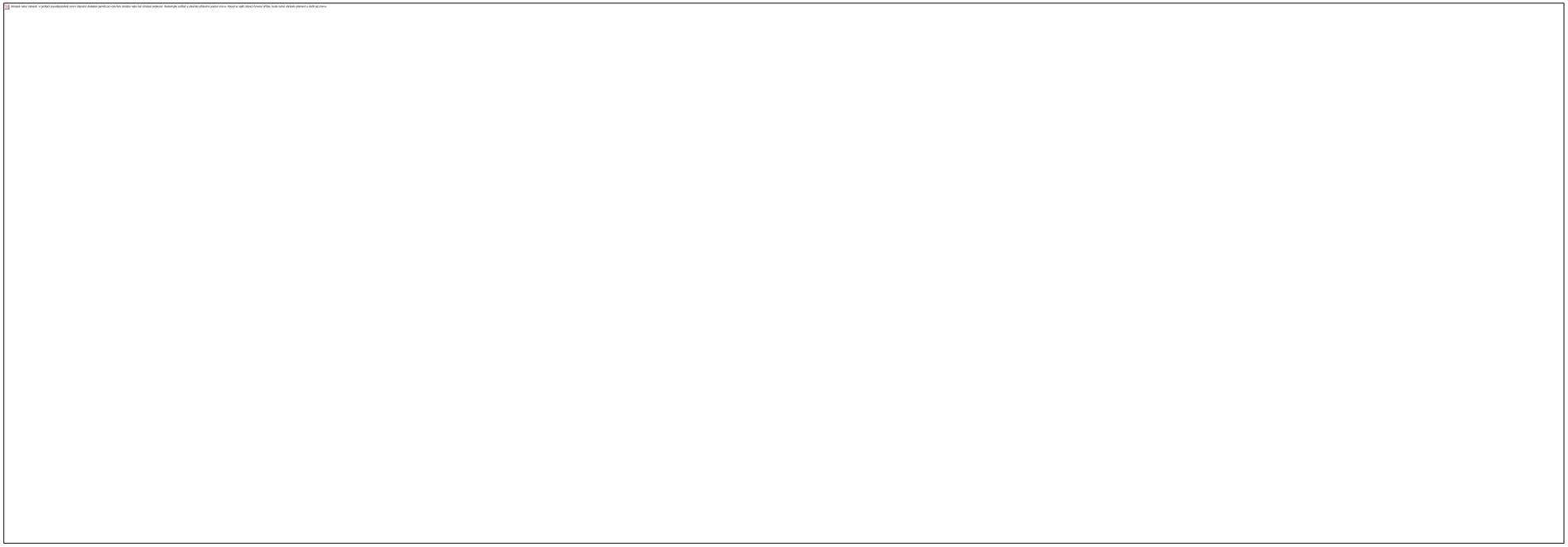
Čo sú to boundary triedy

- Boundary triedy reprezentujú rozhrania medzi systémom a jeho okolím
- Typické boundary triedy:
  - Užívateľské rozhranie
  - Systémové interfacy (rozhrania k iným systémom)
  - Interfacy s externými zariadeniami
- Vo fáze analýzy sa vytvorí jedna boundary trieda pre každú dvojicu actor/prípad užitia

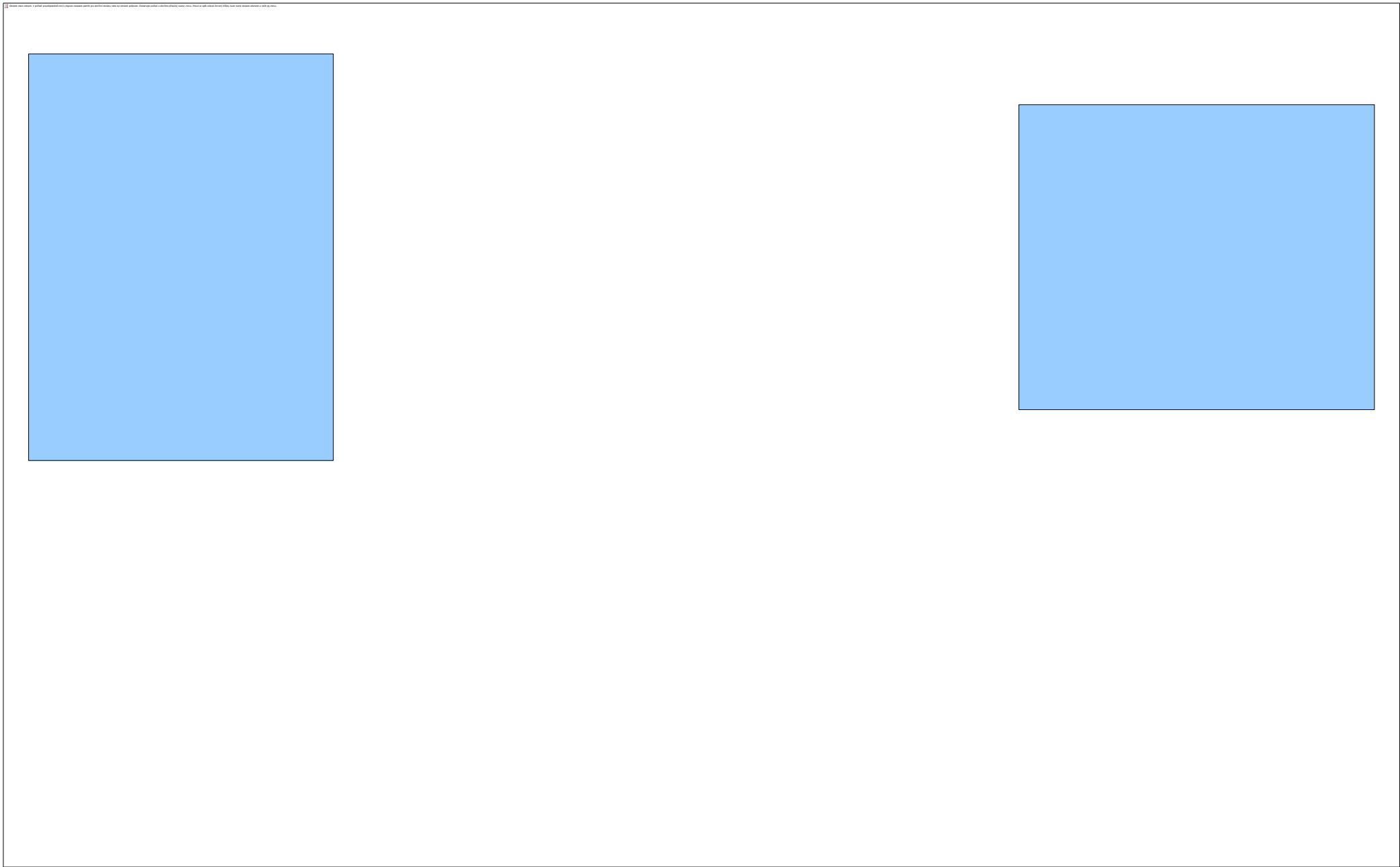
Boundary triedy



Boundary triedy: Príklad



Príklad: model tried



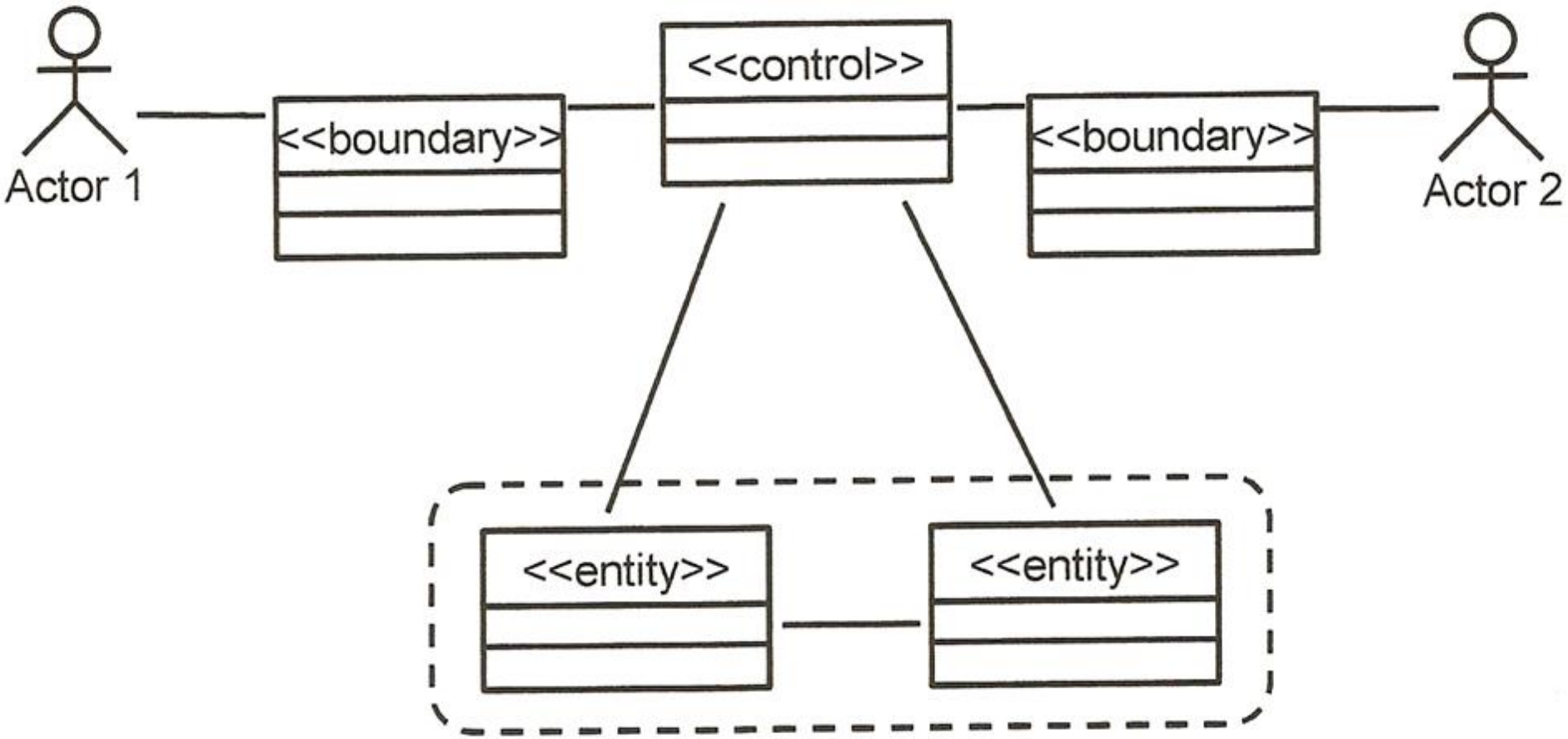
Boundary triedy: ako ich používať

- Užívateľské rozhranie:
  - Uveďte informácie, ktoré sú prezentované užívateľovi
  - Nezachádzajte do prílišných detailov
- Systémové rozhrania a rozhrania so zariadeniami
  - Uveďte, aké protokoly musia byť definované
  - Nezachádzajte do implementačných detailov týchto protokolov

Triedy typu „entity“

- Triedy typu entity predstavujú kľúčové abstrakcie (key abstractions) systému
- Predstavujú koncepty a objekty (=dáta), ktoré systém spracováva
- Zobrazujú dátový model systému
- Typické pre tieto triedy je, že nás budú zaujímať **množiny** ich objektov
- Objekty týchto tried budú často perzistentné (budeme ich ukladať do databázy)
- Ukladajú a spracovávajú dáta

Triedy typu „entity“

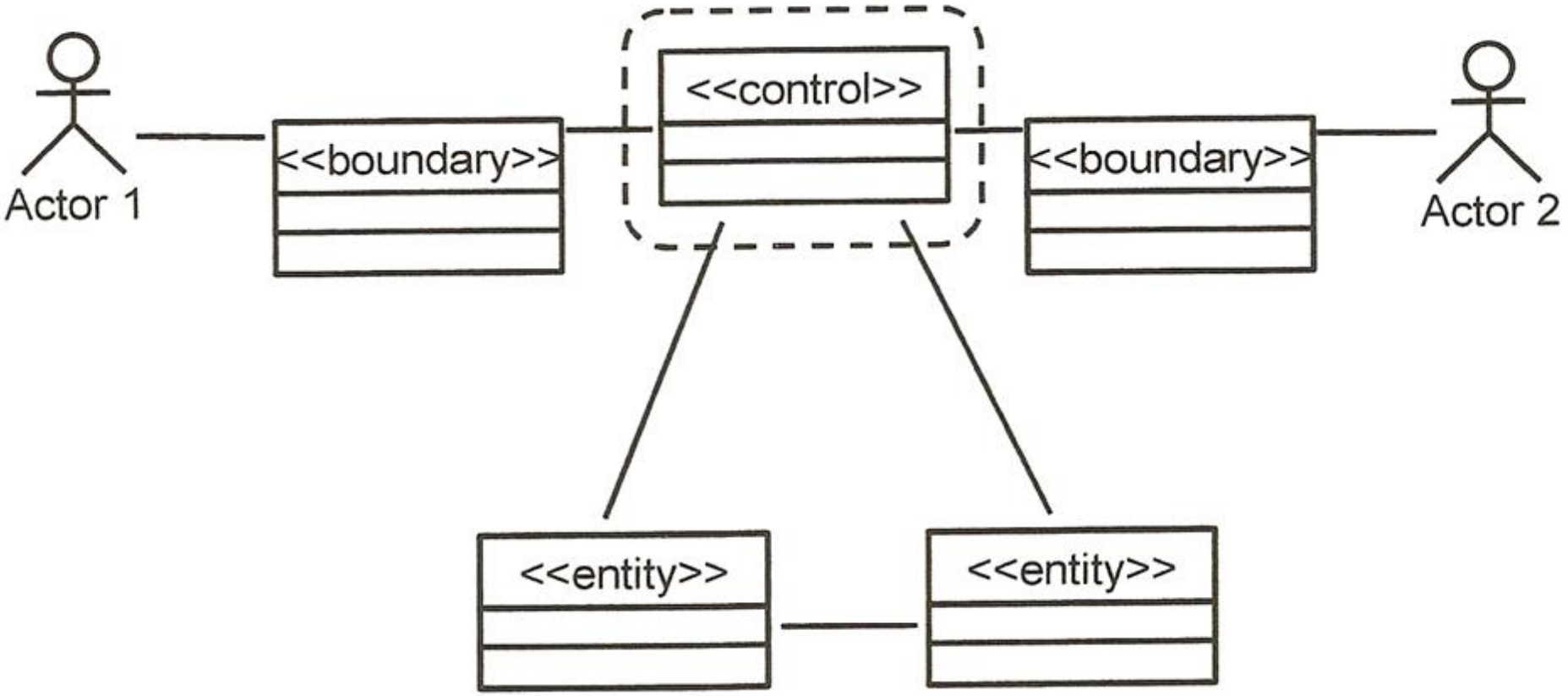


Triedy typu entity: príklad

Triedy typu „control“

- Koordinujú správanie a funkcionalitu daného prípadu užitia
- Obsahujú aplikačnú logiku a definujú jej transakcie
- Oddeľujú logiku užívateľského rozhrania od logiky dát
- Pre každý prípad užitia sa vytvorí jeden controler

Triedy typu „control“



Trieda typu „control“: príklad

Distribúcia správania prípadov užitia jednotlivým triedam

- Pre toky udalostí každého prípadu užitia:
  - Nájdite triedy
  - Vytvorte diagram tried (VOPC: View of participating classes)
  - Priraďte triedam zodpovednosti (=metódy)
  - Znázornite komunikáciu objektov nájdených tried pomocou sekvenčného diagramu

Určte stereotypy tried

- Boundary:
  - Správanie, ktoré zahŕňa komunikáciu s actorom
- Entity:
  - Správanie, ktoré zahŕňa dáta
- Control:
  - Správanie špecifické pre aplikačnú logiku daného prípadu užitia

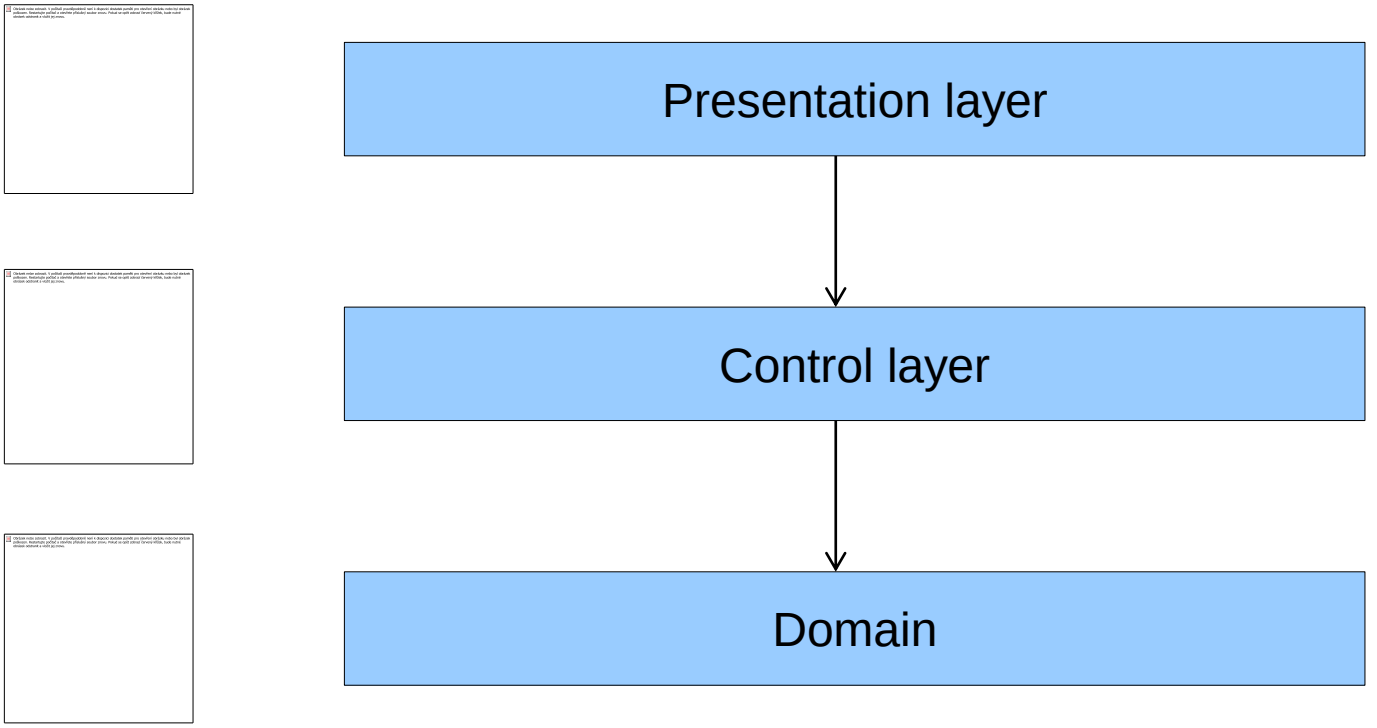
Vytvorte sekvenčný diagram

Sekvenčné diagramy: pozor na zložitosť

- Sekvenčné diagramy nerobte príliš zložité
- Pravidlo: diagram by sa mal zmestiť na jednu obrazovku
- Ak je diagram príliš veľký, rozdeľte ho na menšie

Rozdelenie sekvenčného diagramu: príklad

MVC a architektúra aplikácie



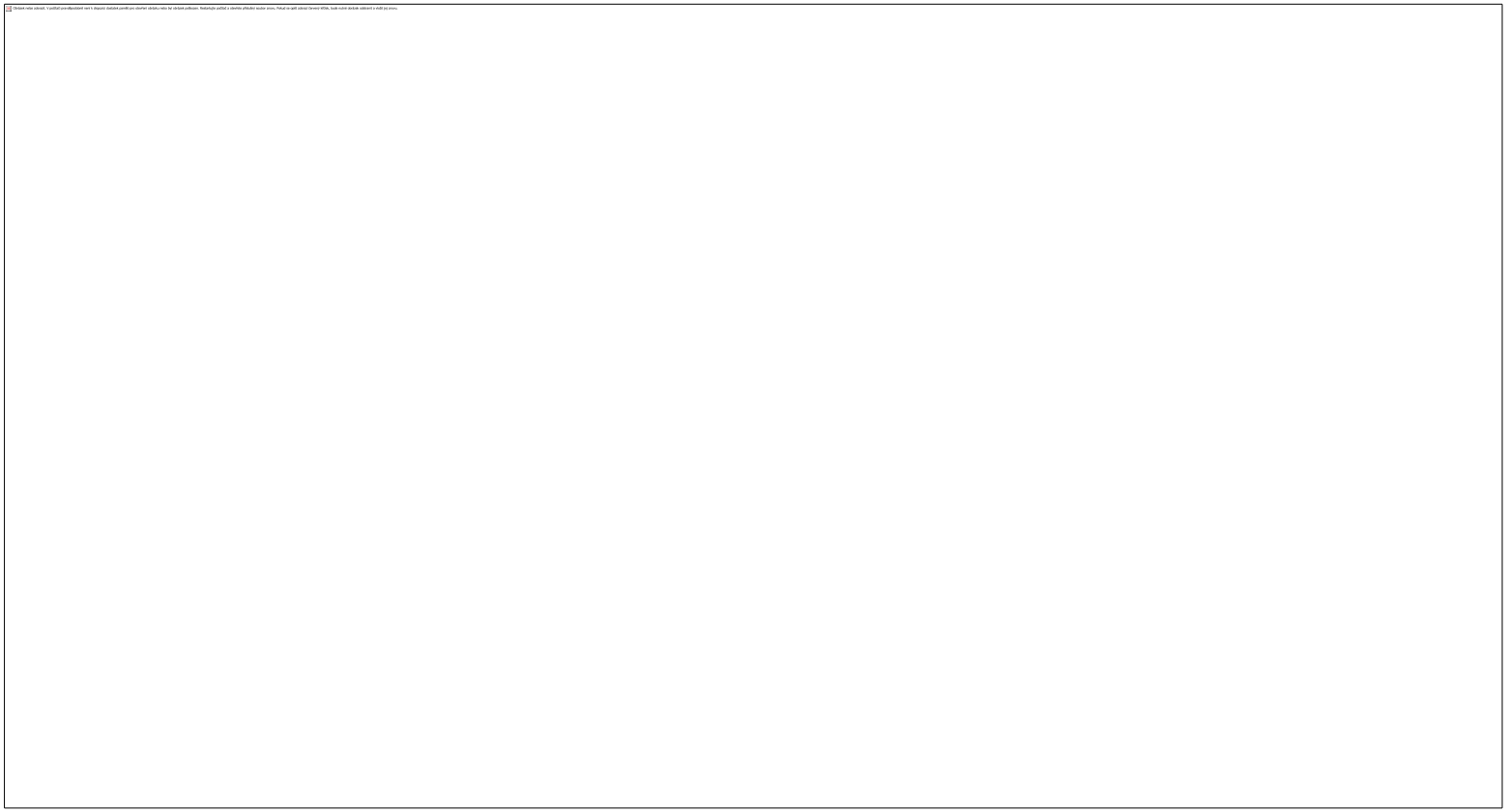
Dizajn pomocou rozhraní

Rozhranie (UML): rozhranie je kolekcia signatúr operácií a definícií atribútov, ktorá definuje množinu správaní. Rozhrania bývajú implementované (realizované), triedami a komponentami.

Realizácia rozhrania

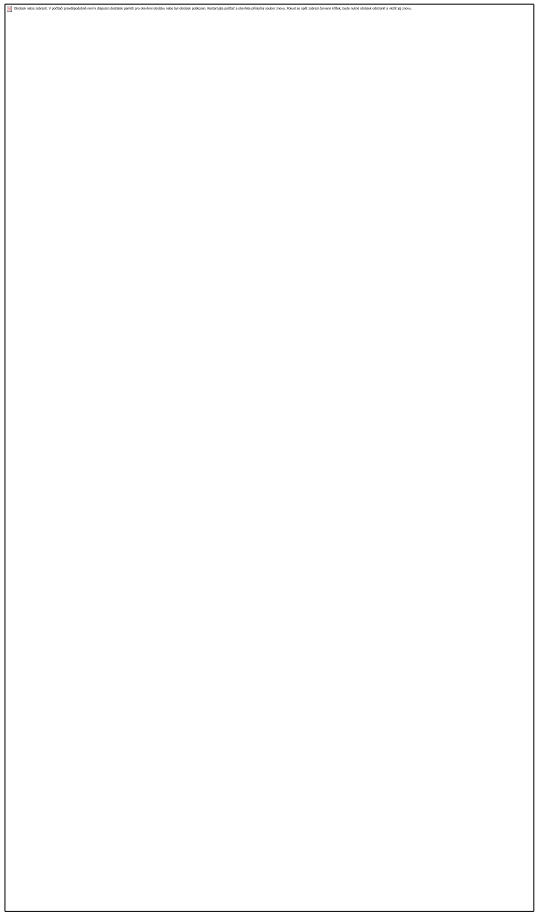
- Triedy realizujú rozhranie, ak implementujú všetky operácie a atribúty definované rozhraním
- Každá trieda môže realizovať ľubovoľné množstvo rozhraní
- Každé rozhranie môže byť implementované rôznymi triedami

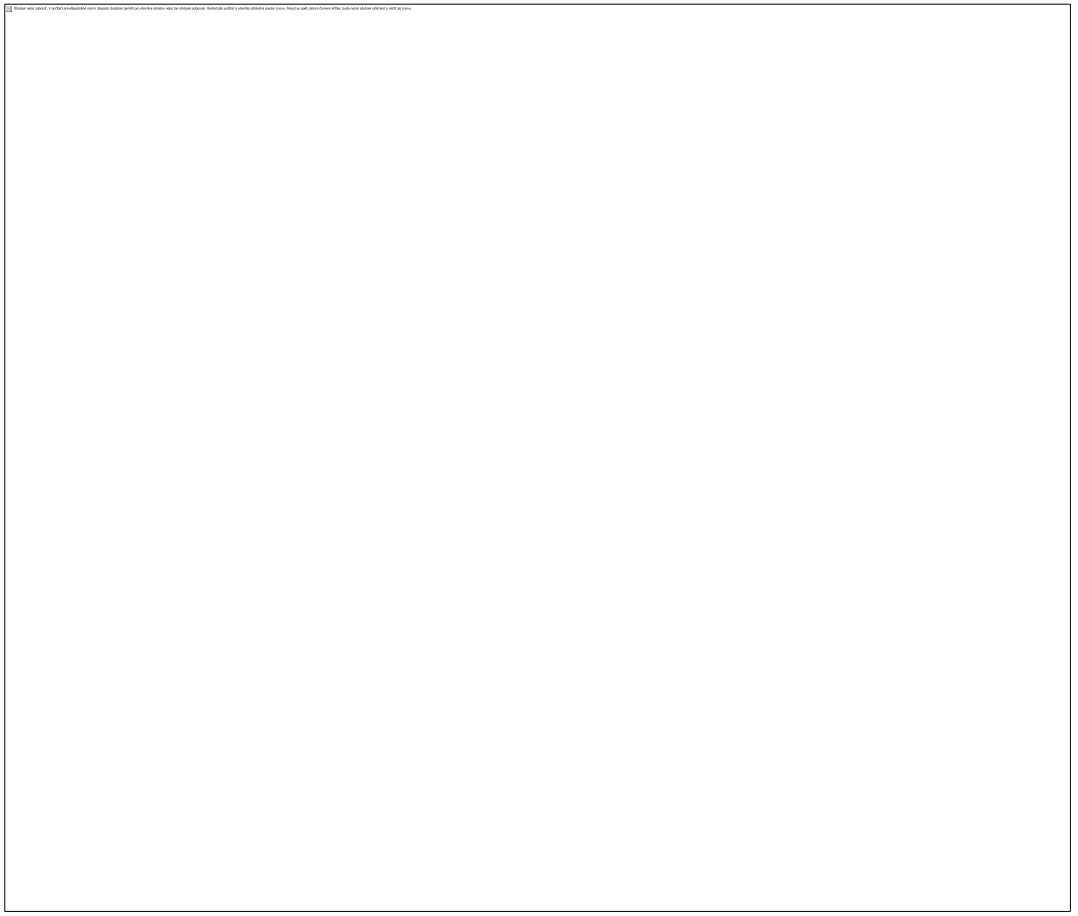
Interface



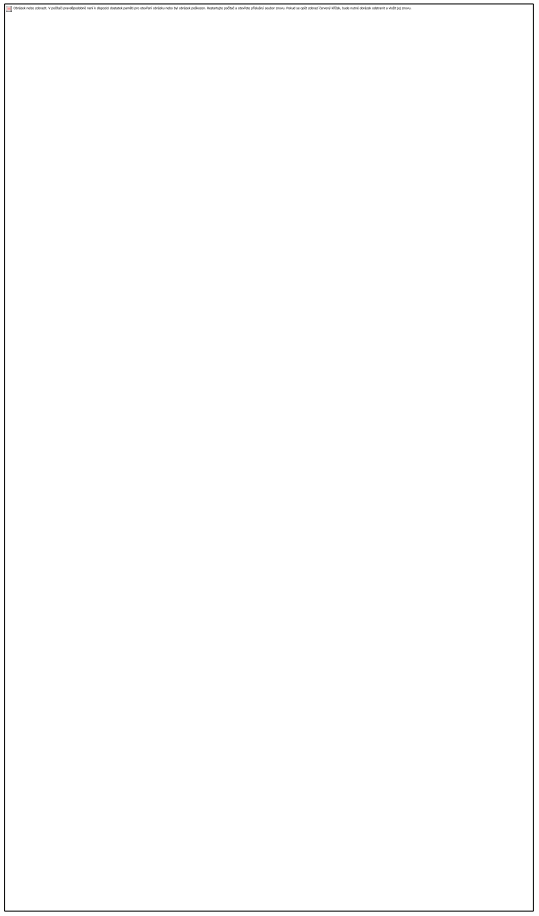
Použitie rozhraní

- Rozhrania sa používajú na uvoľnenie väzieb medzi triedami (znižujú zaťaženie triedy)

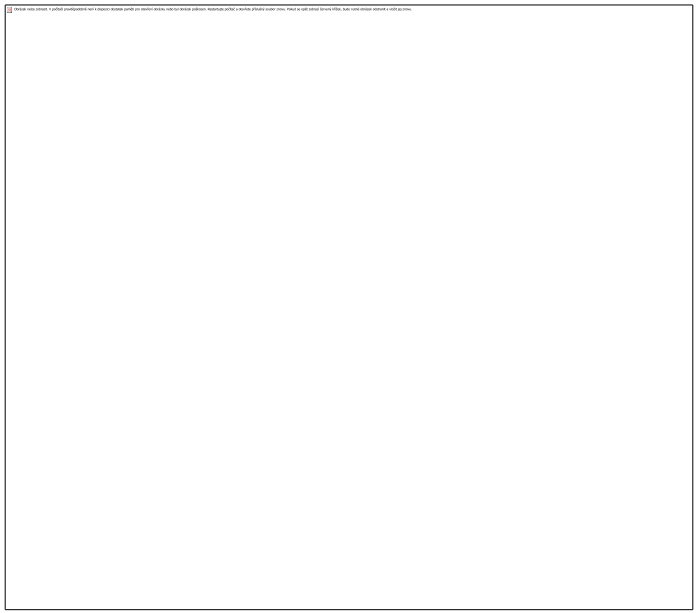




Situácia 1:



- Class2 musí mať neustále k dispozícii Class1.
- Class1 „zaťažuje“ Class2
- Väzba existuje neustále, bez nej sa nedá program skompilovať
- Každý objekt triedy Class2 musí už pri svojom vytvorení byť previazaný s objektom triedy Class1



Situácia 2

- Class2 obsahuje rozranie (abstrakciu). Nie je previazaná so žiadnou konkrétnou triedou.
- Previazanie sa môže uskutočniť neskôr, až vtedy keď ho potrebujeme (late binding).
- Na Class2 sa môžu naviazať objekty rôznych tried (stačí ka tieto triedy realizujú rozhranie).
- Takto sa dá tiež realizovať polymorfizmus (alternatíva dedičnosti)

Rozhranie a loose coupling

- Pomocou rozhraní sa realizuje princíp „loose coupling“ voľné viazanie.
- Triedy nie sú naviazané na konkrétne triedy ale na rozhrania
- Väzby na konkrétne objekty sa vytvárajú čo najneskôr
- Zvýšenie znovupoužiteľnosti
- Zjednodušeni údržby

Rozhrania v návrhu

- Rozhrania budeme používať na oddelenie tried v jednotlivých úrovniach architektúry. Predovšetkým medzi užívateľským rozhraním a triedami typu control
  - Boundary triedy a triedy typu control previažeme pomocou rozhrania

Použitie rozhrania: príklad

Rozhrania a framework Spring

- Previazanie tried pomocou rozhraní sa niekedy realizuje v „absolútnej forme“: Jediný spôsob previazania tried sú rozhrania
- Framework Spring podporuje túto myšlienku:
  - V kóde aplikácie sú triedy previazané jedine cez rozhrania
  - Konkrétne väzby sa definujú pomocou XML konfiguračného súboru
  - Framework Spring zaistí počas behu aplikácie konkrétne väzby. Programátor sa nemusí o nič starať

Unified Process: Fáza návrhu

Návrh: ciele

- Fungujúca aplikácia spĺňajúca požiadavky užívateľa
- Aplikácia musí byť postavená tak, aby ju „ľahko“ bolo možné prispôsobiť novým (zmeneným) požiadavkám
- Zmeny v aplikácii musia mať predvídateľné dôsledky
- Chyby a defekty musia byť „ľahko a rýchlo“ analyzovateľné a odstrániteľné

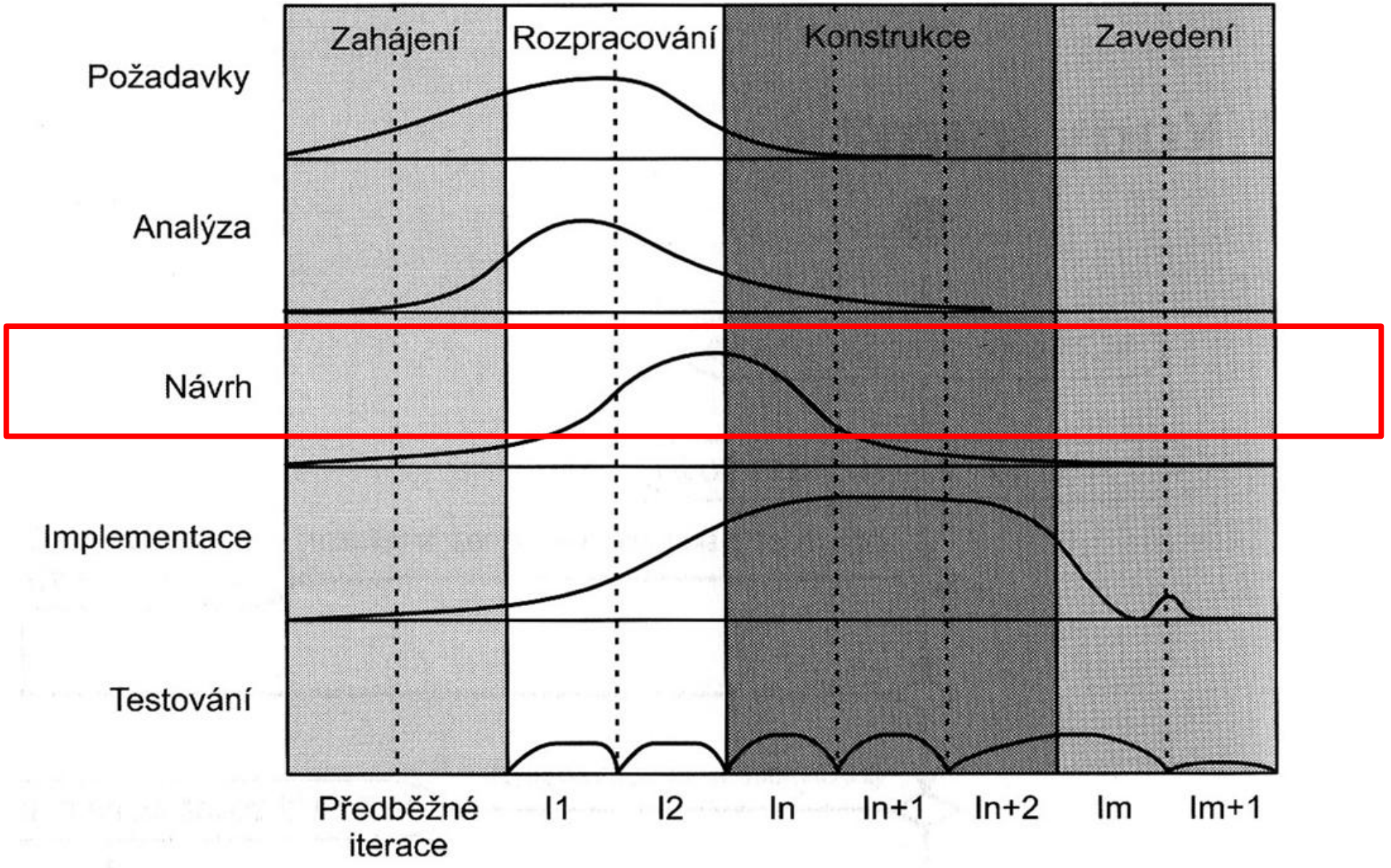
Projekt: ciele

- Vedľa aplikácie spĺňajúcej vyššie uvedené kritériá musí projekt vytvoriť:
  - Užívateľskú a administrátorskú dokumentáciu
  - Proces pre registráciu a administráciu problémov
  - Proces pre registráciu a administráciu zmien
  - Proces pre zálohovanie a obnovu dát
  - Proces pre rekonštrukciu systému v prípade fatálneho zrútenia

Návrh: ako to vyzerá v praxi

- Návrh je problém:
  - zložitosť
  - málo skúseností
  - tlak zákazníkov a managementu

Návrh v kontexte UP



Analýza a dizajn

Systémová analýza – množina aktivít s účelom dekomponovať systém na jeho časti, študovať ich funkcie, interakcie a kvalitu**.**

Systémový dizajn – množina aktivít komplementárna k systémovej analýze s účelom znova poskladať kompletný – a dúfajme že vylepšený - systém.

Analýza podľa UP

- Množina aktivít s účelom vytvoriť logické modely systému, ktorý bude spĺňať požiadavky zákazníka
- Vychádzajúc z popisu užívateľských požiadaviek (use-case model) sa v analýze vytvoria modely zobrazujúce:
  - Štruktúru toho, čo budeme automatizovať (triedy objektov a vzťahy medzi nimi)
  - Prípadne interakcie najdôležitejších (typov) objektov

Návrh podľa UP

- Cieľom návrhu je na základe analytických modelov:
  - Vytvoriť logické modely **štruktúry** systému (definovať jednotlivé prvky a rozhrania)
  - Vytvoriť presné **špecifikácie** jednotlivých **prvkov** systému (ako budú jednotlivé prvky implementované)
  - Rozhodnúť, ktoré **existujúce riešenia** (knižnice, mechanizmy perzistencie a.p.) budú použité pri tvorbe nového systému

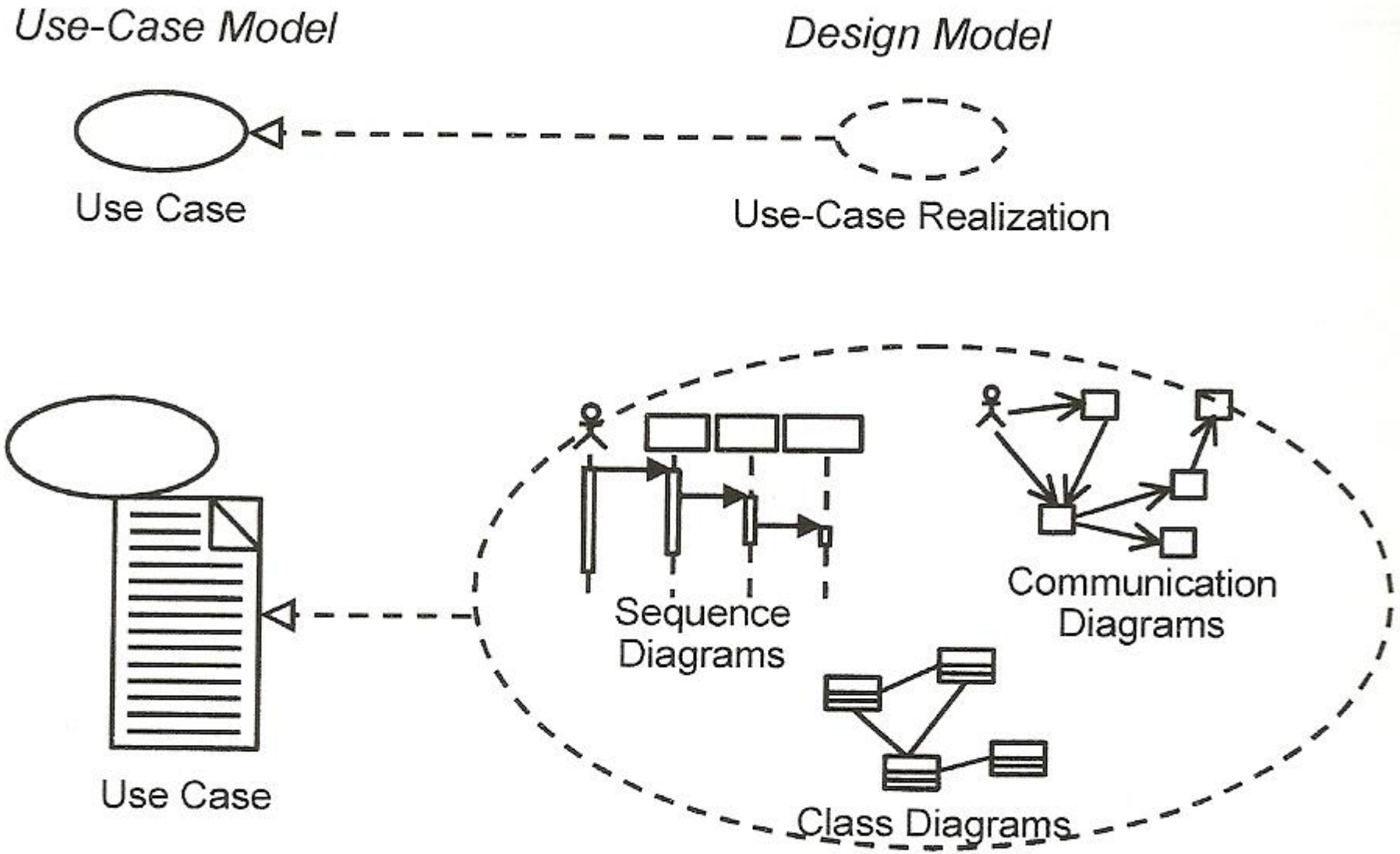
Artefakty vytvorené disciplínami Požiadavky a Analýza

- Prípady užitia a ich špecifikácie
- Model prípadov užitia (actors, use cases)
- Analytické realizácie prípadov užitia
  - Analytické triedy (class diagram)
  - Analytické balíčky
  - Diagramy interakcií

Realizácie prípadov užitia

- Množiny tried, ktoré umožnia realizáciu daného prípadu užitia
- Na úrovni analýzy obsahujú analytické triedy
- Na úrovni dizajnu sa pridajú ďalšie triedy potrebné na realizáciu

Realizácia prípadu užitia II.



Realizácia prípadu užitia: príklad

Obrázek nelze zobrazit. V počítači pravděpodobně není k dispozici dostatek paměti pro otevření obrázku nebo byl obrázek poškozen. Restartujte počítač a otevřete příslušný soubor znovu. Pokud se opět zobrazí červený křížek, bude nutné obrázek odstranit a vložit jej znovu.

Analytické balíčky

- Dôležitý pojem z oblasti analýzy. Balíčky sú však dôležité aj pre návrh
- Balíček je kontajner. Združuje prvky modelu. Podobá sa to na adresár, ktorý združuje súbory
- Ako zoskupovať prvky do balíčkov?

–Logické zoskupovanie sémanticky súvisiacich prvkov

–Podľa prípadu užitia, druhu modelu, architektonickej úrovne,...

- V etape návrhu budú balíčky predstavovať jednotky súbežnej práce

Analytické balíčky II.

- Každý prvok modelu je vlastnený jedným balíčkom (tak ako jeden súbor patrí do jedného adresára)
- Balíčky vytvárajú hierarchické štruktúry

Príklad štruktúry analytického modelu

- Príklad 1: EA model Internetový obchod
- Výhody tohto modelu:

–Jednoduchá a prehľadná štruktúra

–Štruktúra vhodná pre komunikáciu so zákazníkom

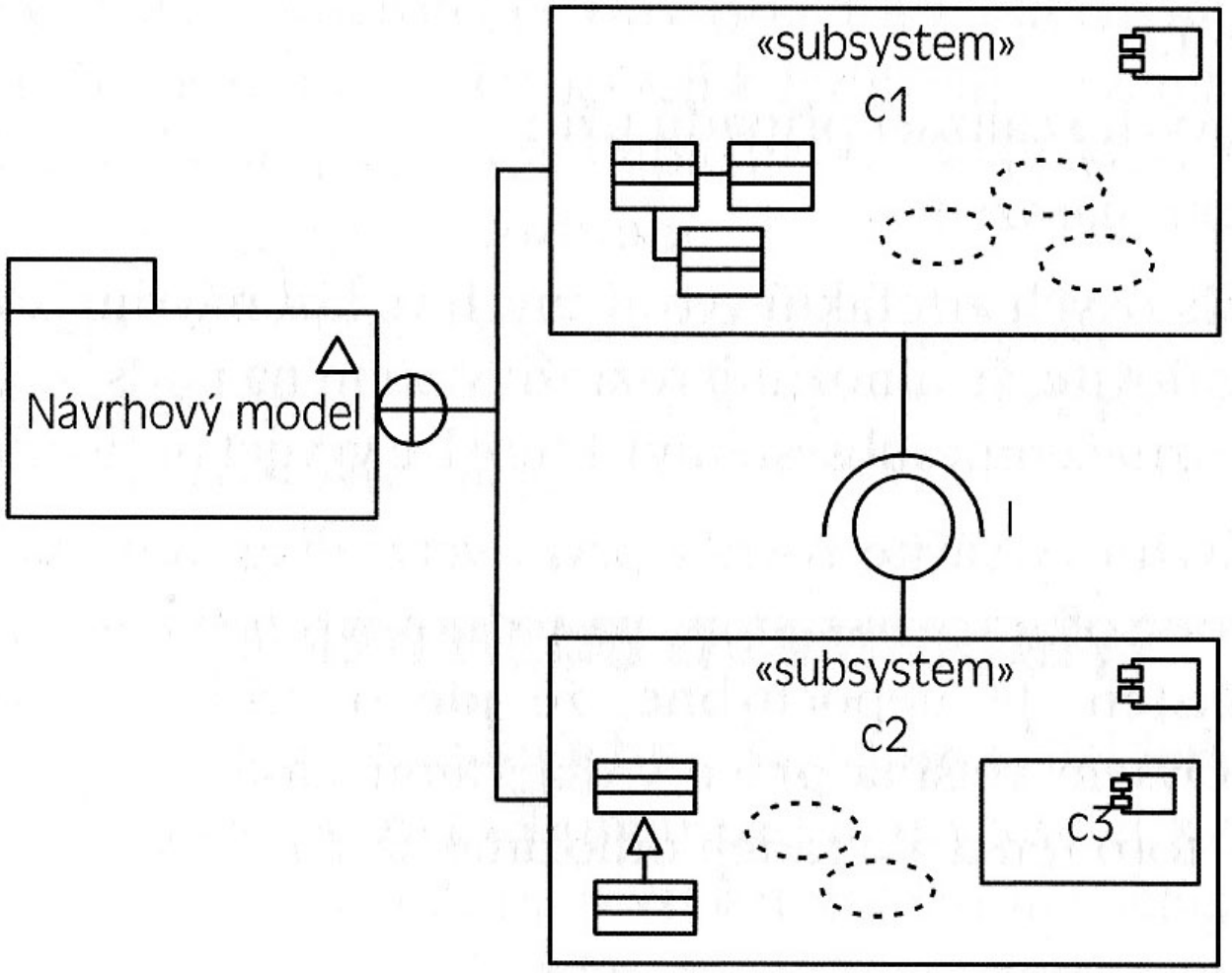
- Nevýhody

–Štruktúra neposkytuje žiadne možnosti na dopĺňanie ďalších údajov a rozširovanie modelu

Štruktúra vhodnejšia pre návrh

- Organizácia podľa prípadov užitia

Artefakty návrhu - metamodel



Subsystémy

- Balíčky zahrňujúce prvky, ktoré realizujú nejakú časť systému
- Typický príklad: subsystém pre komunikáciu s bankou
- V podstate každá realizácia prípadu užitia sa dá považovať sa subsystém

Komponenty

- Balíčky obsahujúce prvky, ktoré k sebe patria fyzicky (napríklad triedy zorganizované v jednej knižnici dll)
-

Artefakty návrhu

- Posun od analytických balíčkov k návrhovým podsystémom
- Veľký dôraz sa kladie na rozhrania
- Návrh štruktúry systému: návrh podsystémov a ich rozhraní
- Podsystémom budú často v konečnej fáze zodpovedať artefakty vytvárané v implementácii: softwarové komponenty

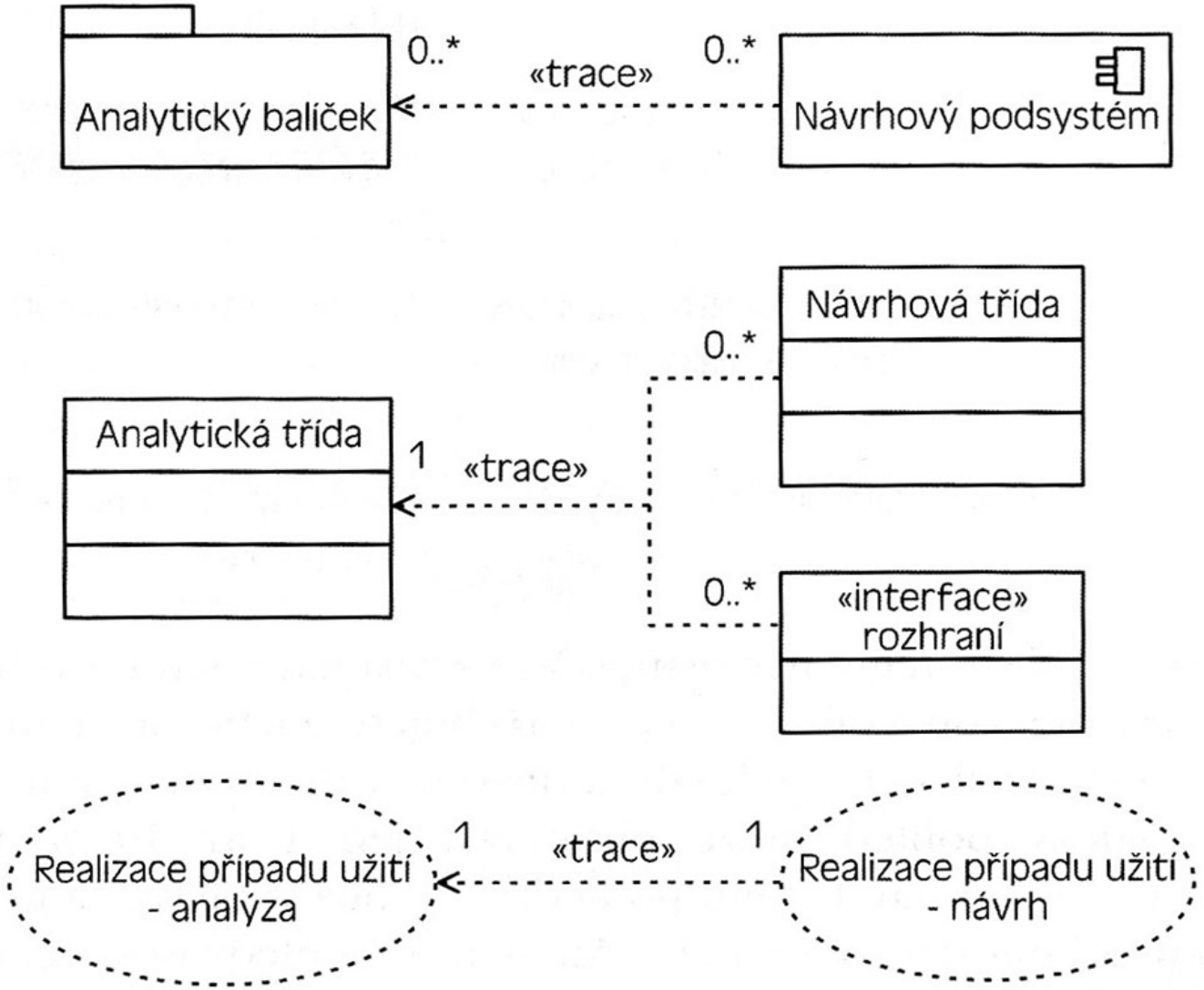
Artefakty návrhu: prehľad

- Počas návrhu sú vytvárané nasledujúce artefakty:
  - Návrhové podsystémy (design subsystems)
  - Návrhové triedy (design classes)
  - Rozhrania (interfaces)
  - Návrhové realizácie prípadov užitia (design use-case realizations)
  - Diagramy nasadenia (Deployment diagrams)

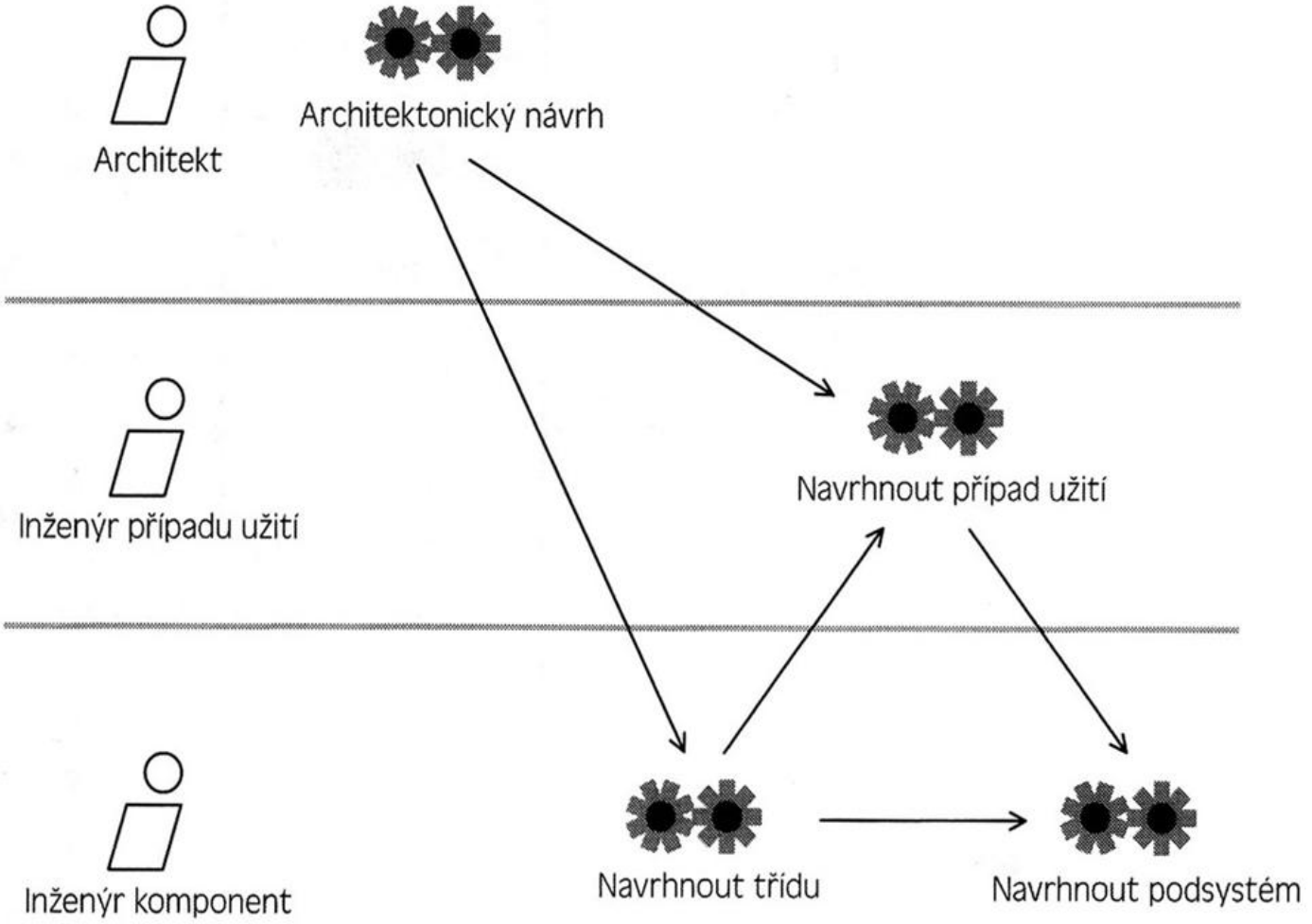
Vzťah analytického a návrhového modelu

- Návrhový model môžeme považovať za ďalšie rozpracovanie analytického modelu
- Mnohé artefakty zodpovedajú artefaktom z analytického modelu, ich špecifikácie sú však úplnejšie a presnejšie
- Artefakty návrhového modelu obsahujú technické a implementačné detaily

Artefakty analytického modelu vs. artefakty návrhového modelu



Detail návrhu



Artefakty návrhu: postup

- Pre každý prípad užitia sa vytvorí realizácia prípadu užitia
- Pre každú realizáciu prípadu užitia sa vytvorí model tried obsahujúci všetky triedy realizujúce prípad užitia (nie iba <<entity>> triedy)
- Pre každú realizáciu sa vytvorí sekvenčný diagram znázorňujúci komunikáciu objektov pri realizácii

Sekvenčný postup?

- Metodiky často vyžadujú sekvenciu:
  Analýza➔Návrh➔Realizácia
- V praxi to často vyzerá takto:
  Analýza➔Realizácia➔Návrh➔Realizácia
- Je to chyba? Nie. Je ťažké navrhnúť niečo hneď na prvý krát. Keď to skúsime naprogramovať, mnohé veci sa nám vyjasnia

Od analytických tried k návrhovým

- Vo fáze návrhu dochádza k ďalšiemu upresneniu špecifikácií tried
- Jedna analytická trieda môže byť prevedená do viacerých návrhových tried
- Pri dopĺňaní špecifikácie analytickej triedy môže dôjsť k tomu, že takto vytvorená návrhová trieda je príliš veľká, vtedy ju treba rozdeliť na niekoľko tried
- Cieľom je vytvoriť „dobrý“ objektovo-orientovaný model.

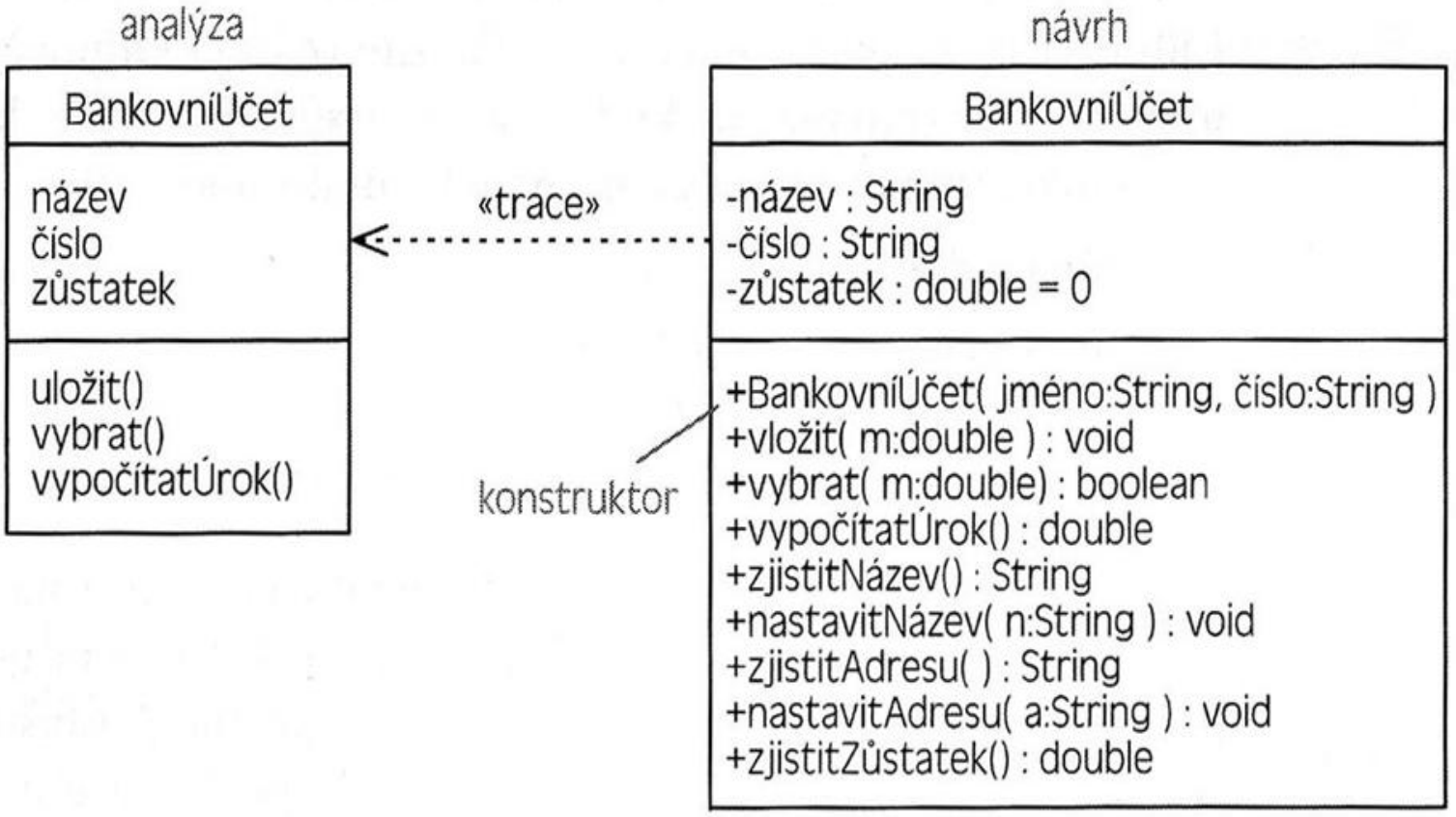
Splývanie analýzy a dizajnu

- V praxi sa málokedy striktne dodržiava oddelenie analýzy a dizajnu
- Tieto dve disciplíny často vykonávajú tí istí ľudia. Aktivity analýzy plynule prechádzajú do dizajnu
- Analýza je spôsob myslenia a je dobre nepreskakovať ju úplne

Návrhová trieda

- Návrhová trieda obsahuje:
  - Kompletnú sadu atribútov a ich detailné špecifikácie (názov, typ, viditeľnosť)
  - Zodpovednosti sú nahradené metódami.
  - Metódy majú presne definované signatúry
  - Presne špecifikované asociácie

Anatómia návrhovej triedy



Návrhové relácie

- Mnohé analytické relácie nie je možné implementovať priamo a je nutné ich vo fáze dizajnu spresniť
- Žiaden OO programovací jazyk napríklad neumožňuje priamo implementovať:
  - Dvojsmerné vzťahy
  - Asociácie s násobnosťou 1..n (m..n)
  - Asociačné triedy

Požiadavky na dizajnové vzťahy

- Všetky dizajnové vzťahy musia:
  - Byť jednosmerné (MUSIA mať smer!)
  - Mať definovanú násobnosť na obidvoch koncoch
- Navyše by dizajnové vzťahy mali aspoň na jednom konci obsahovať názov role

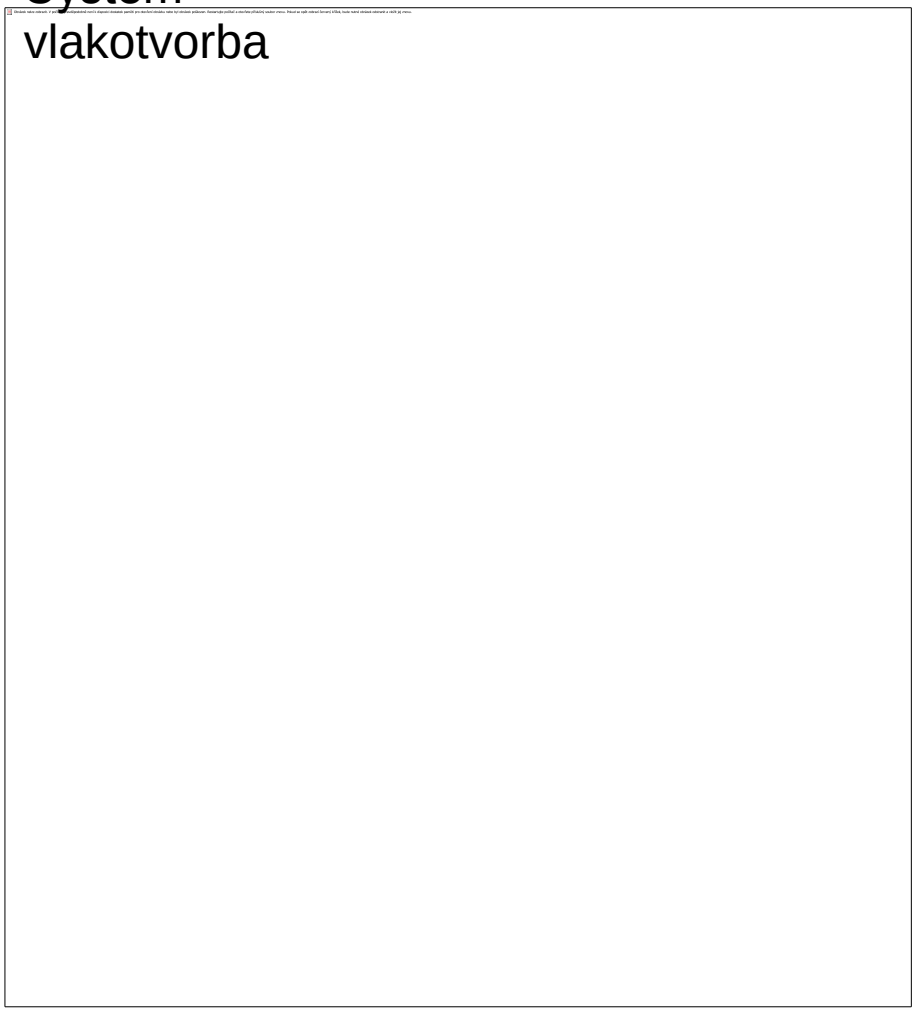
Agregácia a kompozícia

- Dva spôsoby modelovania vzťahov typu celok/časť
- Rozdiel je v tom, aký pevný je vzťah medzi celkom a jeho časťami:
- Agregácia je voľný vzťah
- Kompozícia je silný vzťah

Agregácia: príklad

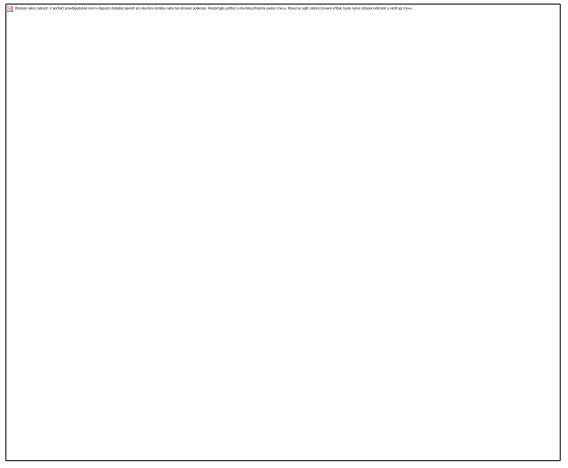
Agregácia: iný príklad

Systém



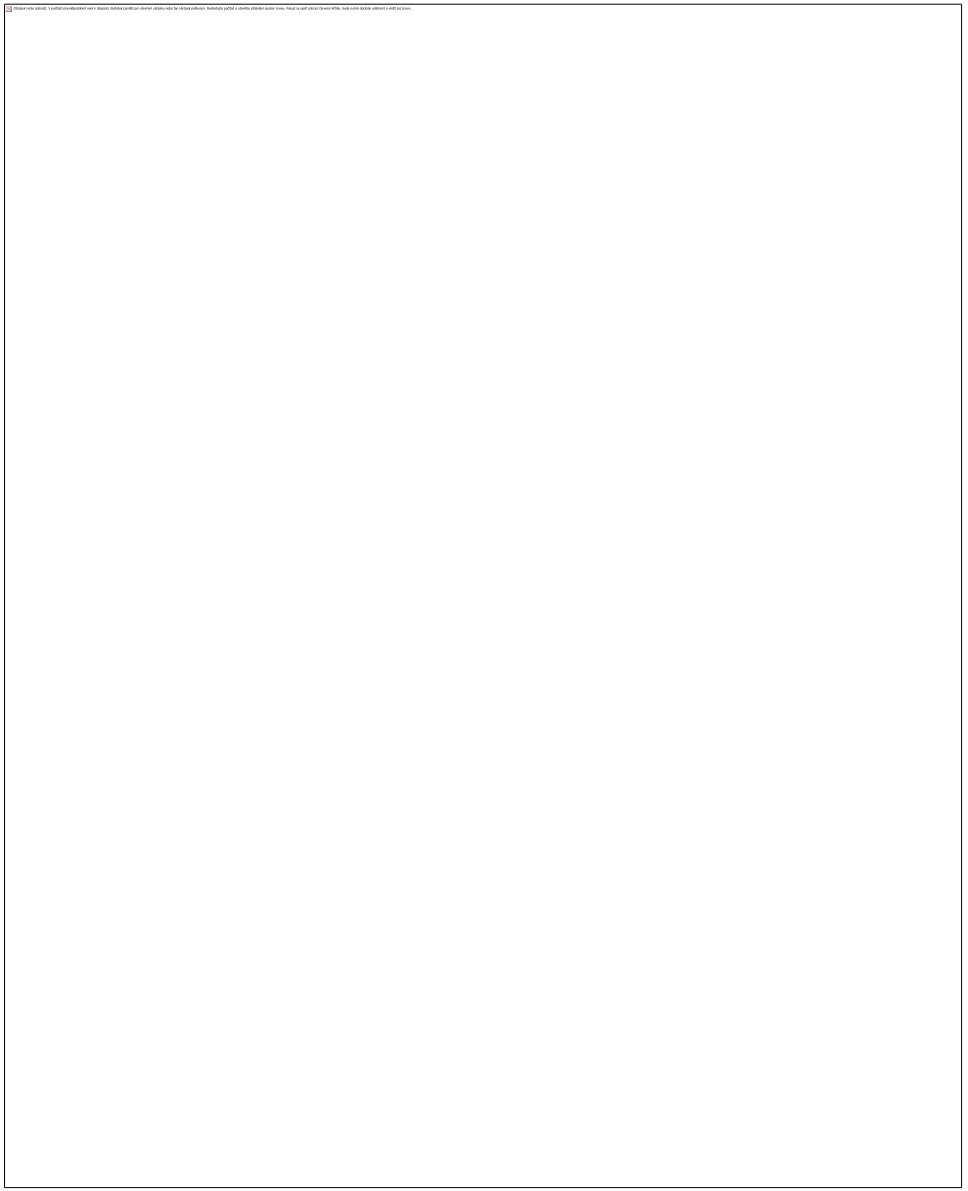
Sémantika agregácie

- Časti môžu existovať nezávisle na celku
- Časť môže byť zdieľaná viacerými celkami
- Objekt nemôže byť súčasťou seba samého



Takýto model je v poriadku. Objekt danej triedy sa môže skladať z iných objektov tej istej triedy (nie však zo seba samého)

Kompozícia: príklady



Sémantika kompozície

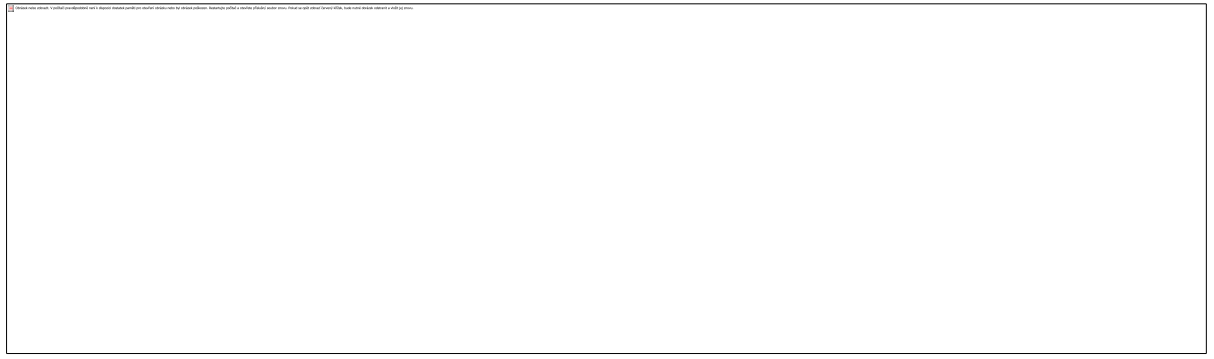
- Silnejší vzťah než agregácia
- Časti nemôžu existovať sami o sebe mimo celku
- Časť patrí vždy iba k jednému celku
- Celok nesie výhradnú zodpovednosť za svoje časti, za ich vznik a zánik
- Ak zrušíme celok, zrušíme s ním automaticky aj všetky jeho časti
- Typickým príkladom kompozície sú atribúty objektu

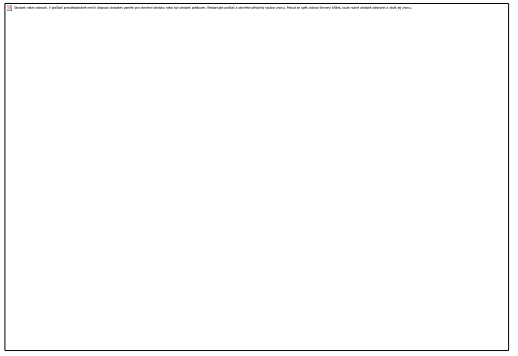
Ako upresniť analytické relácie

- Rozhodnite sa či použiť agregáciu alebo kompozíciu
- Pridajte násobnosti a názvy rolí
- Pozor! Kompozíciu môžete použiť iba vtedy, ak násobnosť na strane celku je rovná 1
- Smer vzťahu je vždy od celku k jeho častiam
- Násobnosti vyššie než 1 budeme upresňovať neskôr

Asociácie typu 1:1

- Asociácia typu 1:1 predstavuje takmer vždy kompozíciu
- Triedy s touto asociáciou je možné (nie nutné) zlúčiť do jednej triedy (jedna trieda sa stane atribútom inej triedy)





Asociácie typu 1:N

- V asociáciách typu 1:N máme na strane N kolekciu objektov
- Väčšina OO jazykov obsahuje vstavanú podporu kolekcií rôznych druhov
- Rôzne druhy kolekcií sú vhodné pre rôzne typy prístupov
- Voľba správneho druhu kolekcie je jedným z kľúčov dobrého OO návrhu

Kolekcie

- Trieda kolekcia je trida, ktorej inštancie majú za účel spravovať kolekcie iných objektov
- Wikipedia: a collection is grouping of some variable data items (possibly zero) that have some shared significance to the problem being solved and need to be operated upon in some controlled fashion

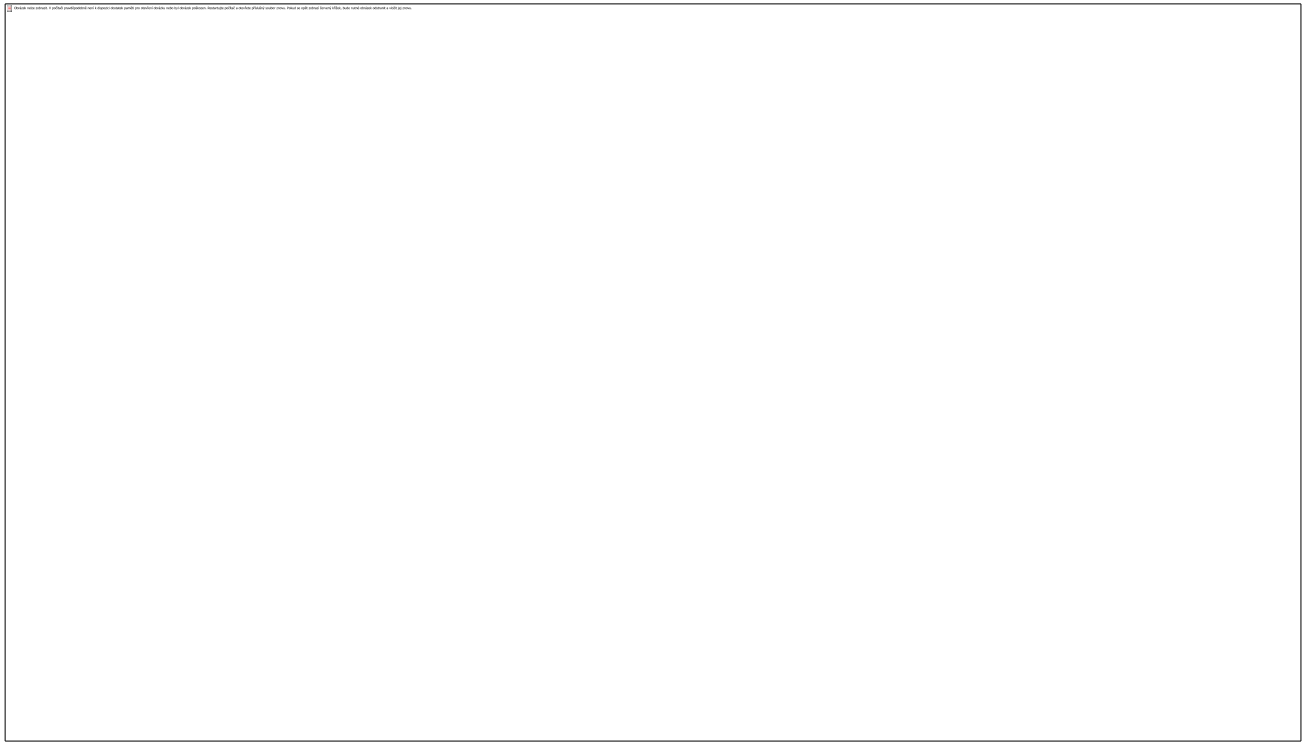
Operácie s kolekciami

- Všetky triedy kolekcií obsahujú metódy pre
  - Pridávanie objektov do kolekcie
  - Odstraňovanie objektov z kolekcie
  - Získavanie odkazu na objekt uložený v kolekcii
  - Prechádzanie kolekcie

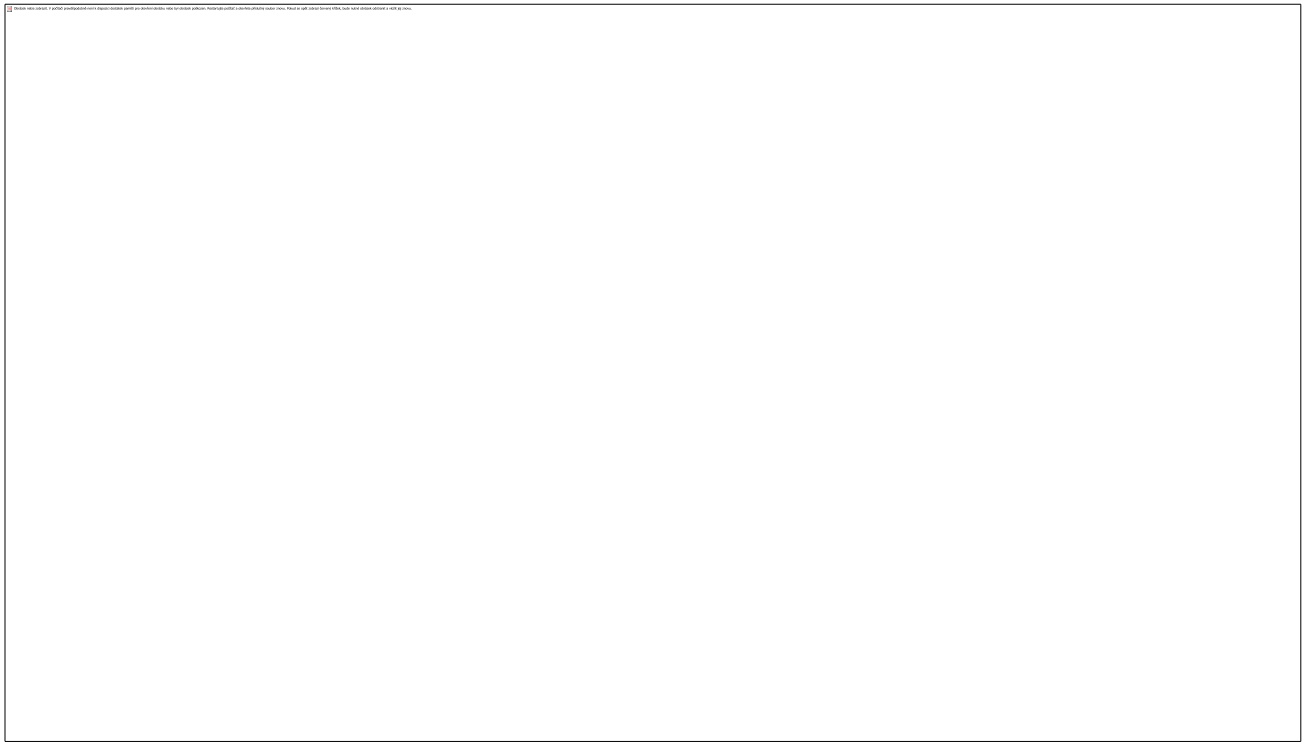
Dôležitý rozdiel medzi OO a relačným modelovaním

- Pri objektovom modelovaní sa zaoberáme jednotlivými objektami. Ak cheme hovoriť o množinách objektov, musíme použiť kolekcie
- Pri relačnom modelovaní sa zaoberme (z definície pojmu relácia) množinami objektov. Všetky operácie na relačnom modeli sú operácie s množinami objektov. Jednotlivé objekty sa v relačnom svete modelovať nedajú.

Príklad kolekcie



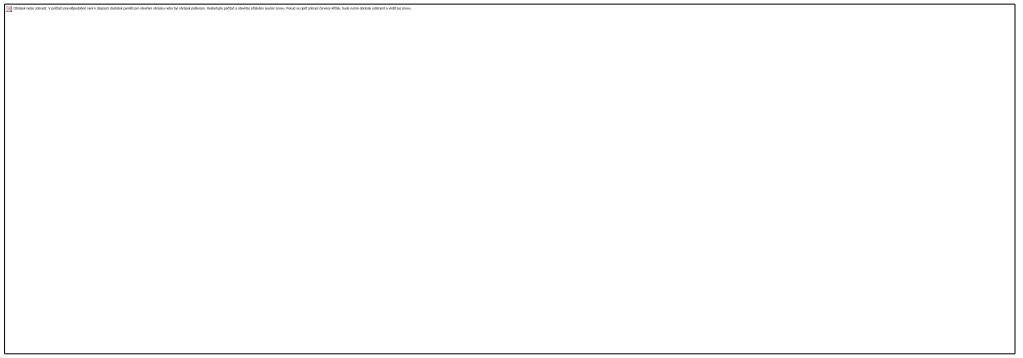
Príklad kolekcie 2



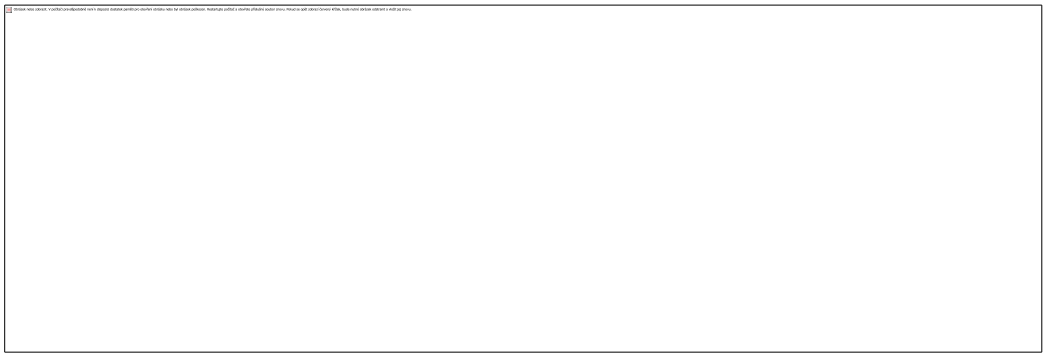
Kolekcia a trieda na strane celku majú teda vždy vzťah typu Kompozícia

Iné možnosti modelovania kolekcií

Ako stereotyp:



Ako stereotyp spolu s ďalšími implementačnými podrobnosťami:



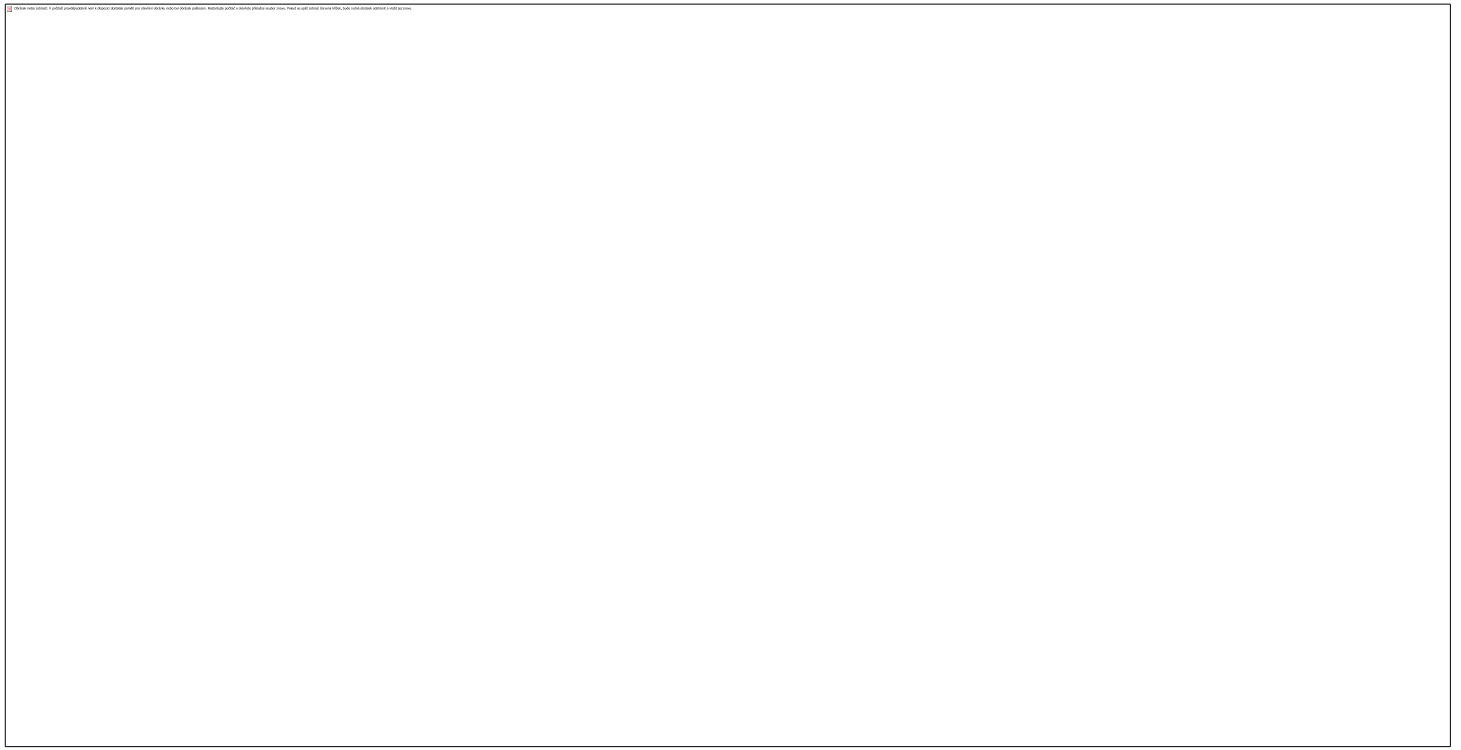
Konkretizované relácie

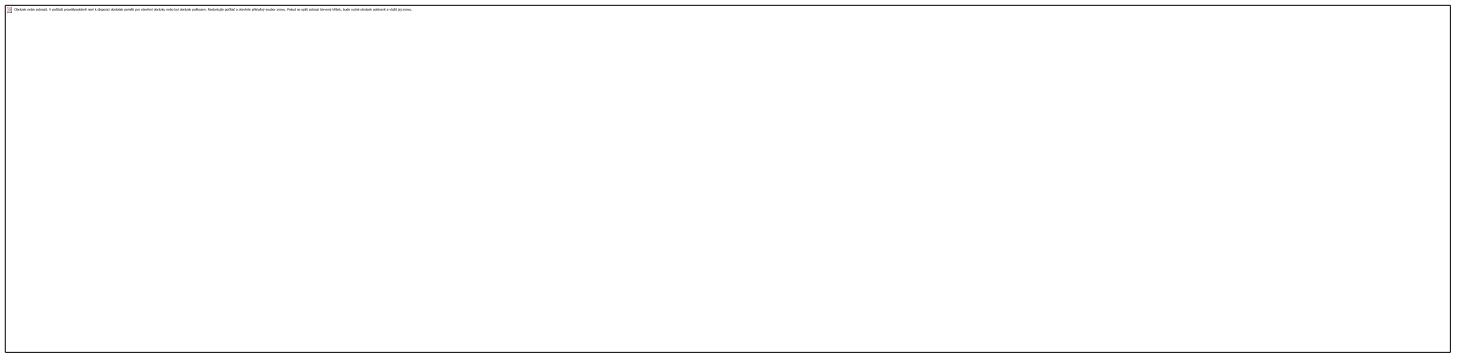
- Niektoré analytické relácie nie je možné preložiť priamo do návrhu:
  - Asociácie M:N
  - Obojsmerné asociácie
  - Asociačné triedy *(Podivný preklad v literatúre ako „triedy asociácií“)*
- Tieto analytické relácie je potrebné „konkretizovat“

Asociácie typu M:N

- Asociácie typu M:N nie sú bežne podporované žiadnym OO jazykom
- Musíme ich konkretizovať do normálnych tried, agregácií, kompozícií a závislostí
- Ak z analýzy nie je jasné, ktorá strana predstavuje celok, musíme to rozhodnúť v návrhu
- Asociácie typu M:N bývajú často obojsmerné

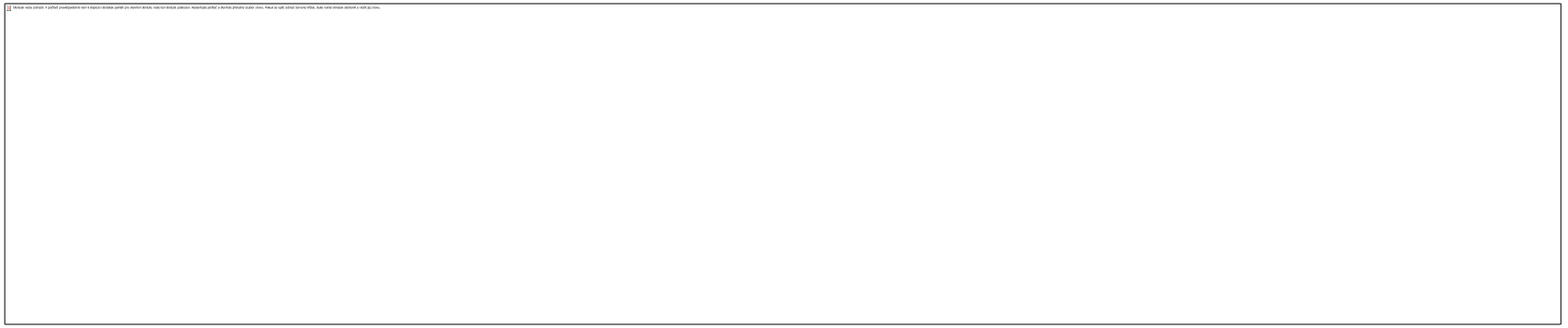
Asociácia M:N príklad





Obojsmerné asociácie

- Obojsmerné asociácie je potrebné konkretizovať do dvoch jednosmerných asociácií alebo závislostí
- Pozor na obmedzenie v súvislosti s asymetriou agregácie a kompozície: objekt nesmie byť nikdy súčasťou seba samého



Asociačná trieda

Štruktúry modelov

Literatúra

www.ibm.com

RSA model structure guidelines

- Modely softwarových systémov majú tendenciu rásť a stávať sa neprehľadnými
- Neprehľadný model = nepoužiteľný model
- Dobrá štruktúra modelov Vám umožní
  - Rýchlo sa v modeloch orientovať
  - Nájsť informácie, ktoré potrebujete

Použitie modelov

- Počas návrhu (technický výkres)
  - Predloha pre programátora
- Počas údržby (ako dokumentácia)
  - Užívateľ chce rozšíriť prípad užitia:
    - Ktoré komponenty a triedy sa podieľajú na jeho realizácii?
    - Ktoré komponenty máme už k dispozícii a môžeme ich použiť?
    - Aké rozhrania majú existujúce triedy na ktoré sa musíme napojiť?

- Chceme zmeniť kód metódy M triedy T
  - Ktoré komponenty túto metódu používajú?
    - Ak ju zmením tak ovplyvním všetky tieto komponenty
  - Na základe toho sa môžem rozhodnúť, či kód naozaj zmením alebo problém vyriešim inak
  - Ak sa rozhodnem kód zmeniť, tak musím všetky tieto komponenty otestovať
- Toto sa označuje ako „*Where used?*“ alebo inak *Traceability*

Štruktúra modelov podľa metodiky RUP

- Modely:
  - Model prípadov užitia
  - Analytický model (nepovinný)
  - Návrhový model
  - Model implementácie
  - Deployment model
- Budeme vychádzať zo zlúčeného analytického a návrhového modelu
  - Najskôr vznikne analytický model, ten sa neskôr transformuje do návrhového

Modely a diagramy

- Pozor na rozdiel medzi modelom a diagramom
- Model obsahuje definície prvkov systému
- Diagram je grafickým zobrazením časti modelu
  - A je zároveň jeho súčasťou
- Model je teda súhrnom všetkých diagramov a definícií prvkov systému
- Rôzne diagramy reprezentujú rôzne pohľady na rôzne časti systému

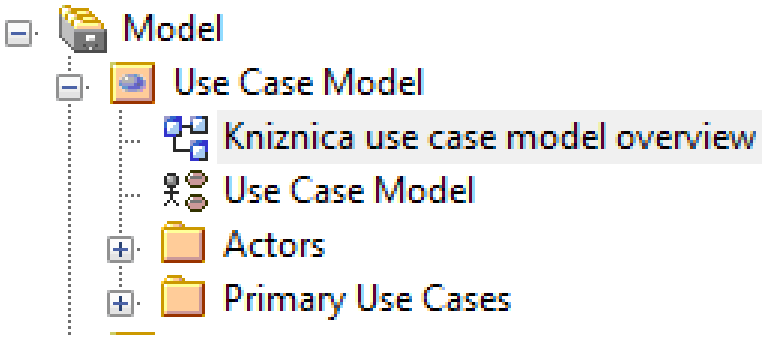
Single Point of Definition

- Dôležité pravidlo modelovania: Každý prvok systému definujte práve na jednom mieste
- Každá trieda, rozhranie, atribút, vzťah atď budú v modeli iba raz
  - Veľmi častá chyba: Trieda X je definovaná vo všetkých balíčkoch, kde ju chceme vidieť
- Prvok, ktorý sme v modeli raz zadefinovali, sa môže zobraziť na mnohých diagramoch

Use Case model

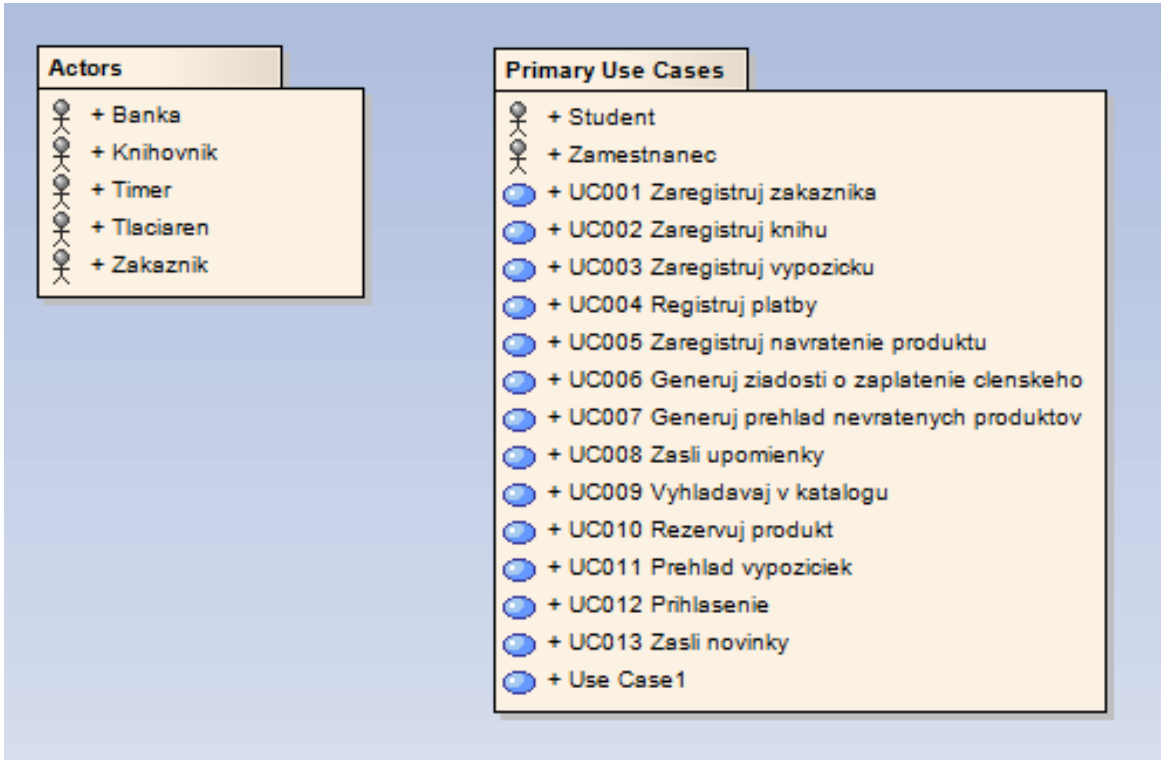
- Ten už poznáte
  - Actors
  - Use Cases
  - Use Case Diagrams

Use Case Model: príklad



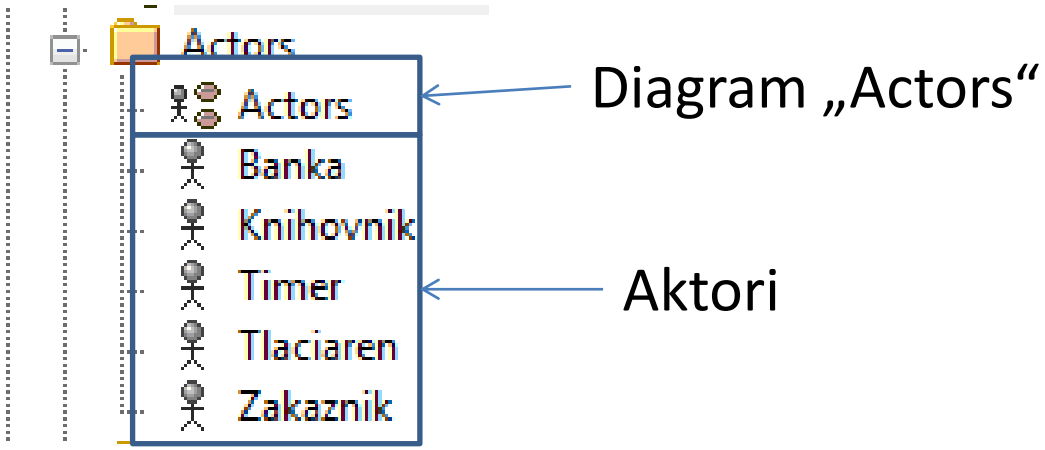
- Každý model bude obsahovať jeden diagram s názvom „overview“ alebo „prehľad“
- Tento diagram bude obsahovať odkazy na dôležité komponenty modelu
- V tomto prípade Actors a Use Cases

Use Case model overview: príklad

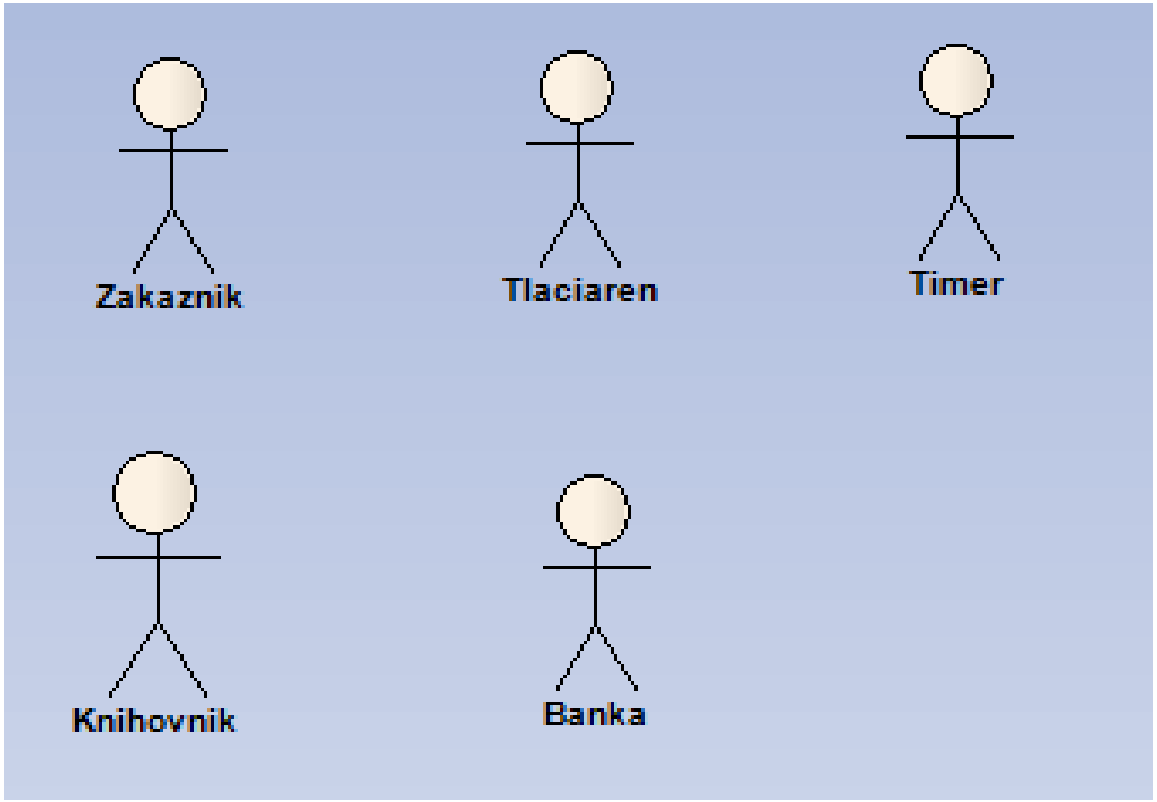


Prehľad nemusí mať iba formu balíčkov, akákoľvek iná forma je prípustná, ale musia tam byť odkazy na jednotlivé časti modelu

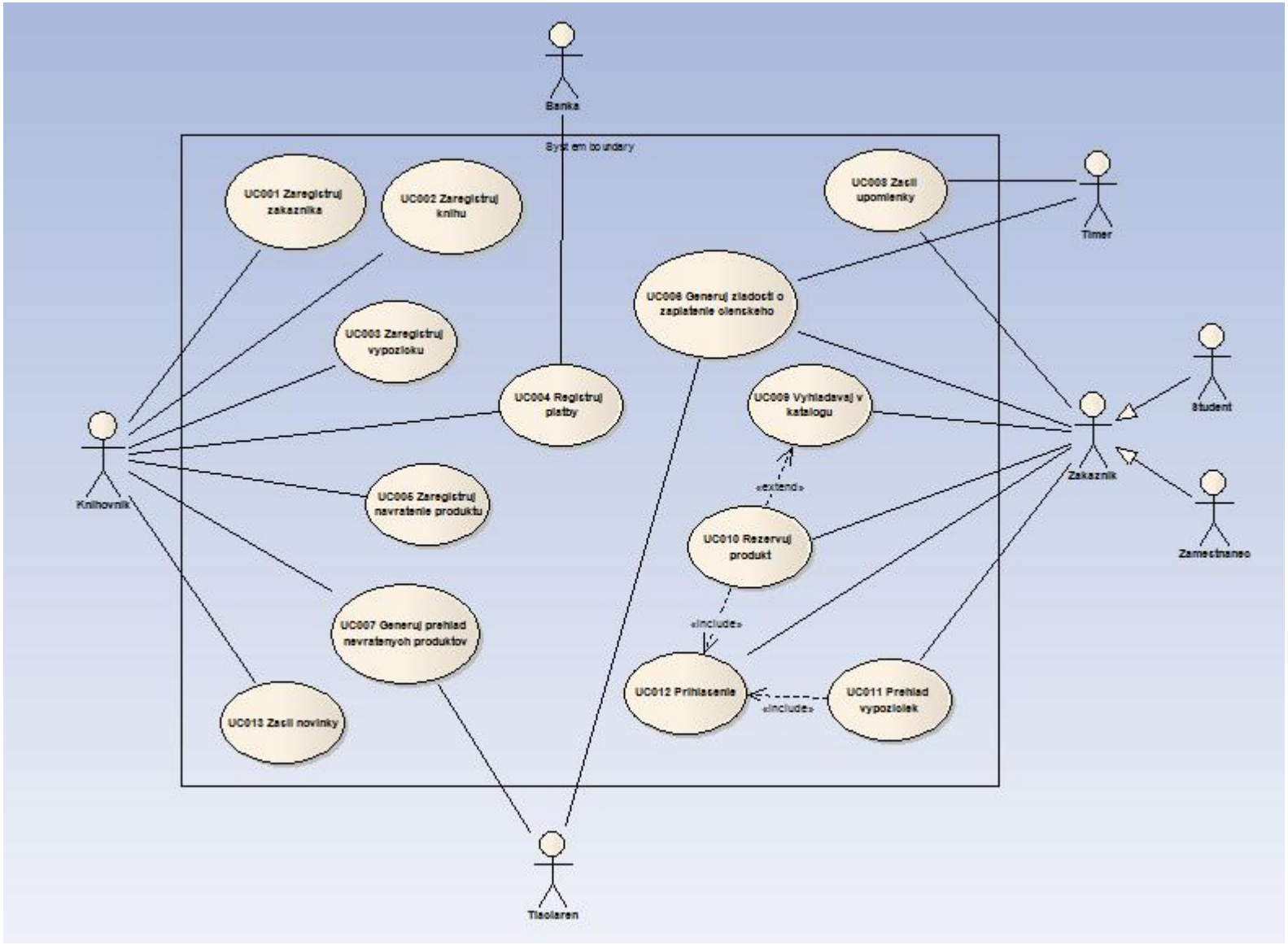
Balíček Actors



Use Case Diagram Actors



Use Case Diagram „Knižnica“



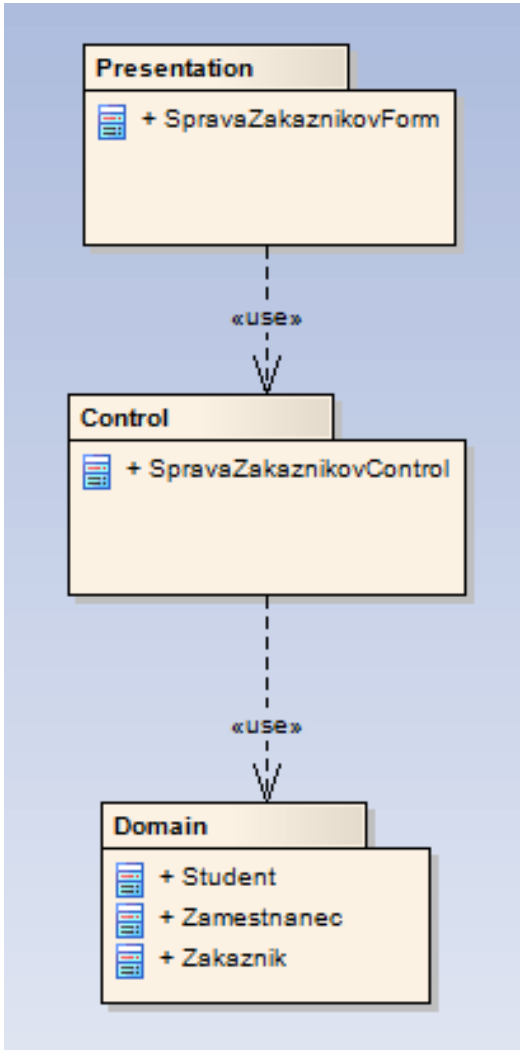
Návrhový model

- Úrovne architektúry (Architectural layers)
  - Presentation (V)
  - Control (C)
  - Domain (M)
- Design Contracts
  - Design Mechanisms
  - Design level Use Case Realizations

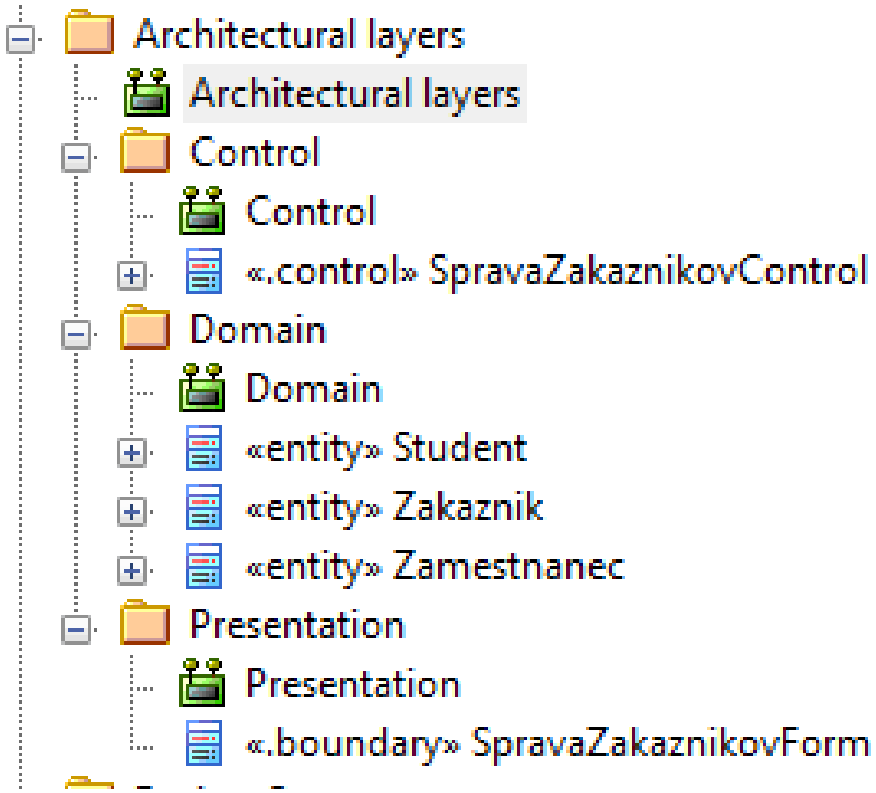
Architectural Layers

- Balíčky v tejto časti modelu obsahujú definície tried
- Definície tried sú teda zorganizované podľa úrovní architektúry
- Architectural layers sú jediným miestom, kde sú triedy definované.
  - Ak potrebujeme pridať novú triedu, tak ju zaradíme podľa úrovne architektúry

Architectural layers: príklad



Architectural layers: iný pohľad

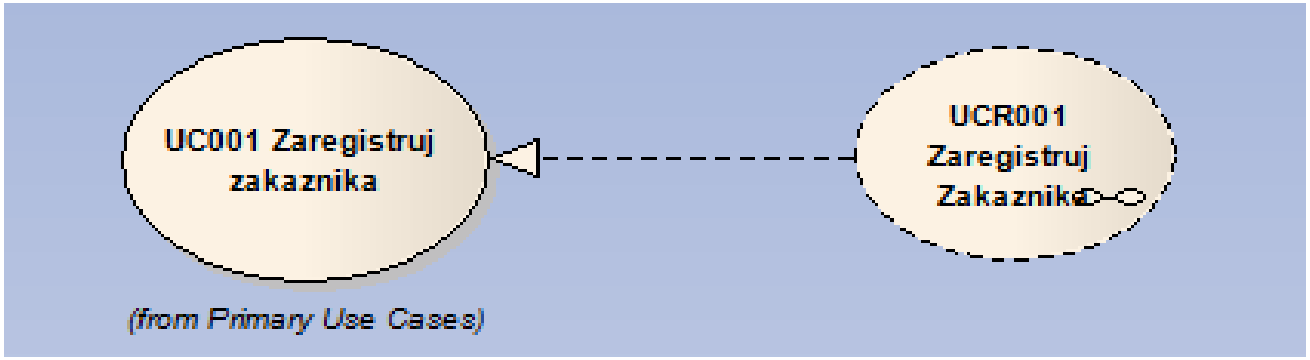


Design contracts

- Design mechanisms
  - Podobá sa to na návrhové vzory
  - Je to predpis ako štandardne realizovať niektoré záležitosti
  - Typický príklad: Perzistencia
  - Predpis, ako má programátor, ktorý chce perzistentne ukladať nejaké objekty, postupovať
- Design Level Use Case Realizations

Realizácie

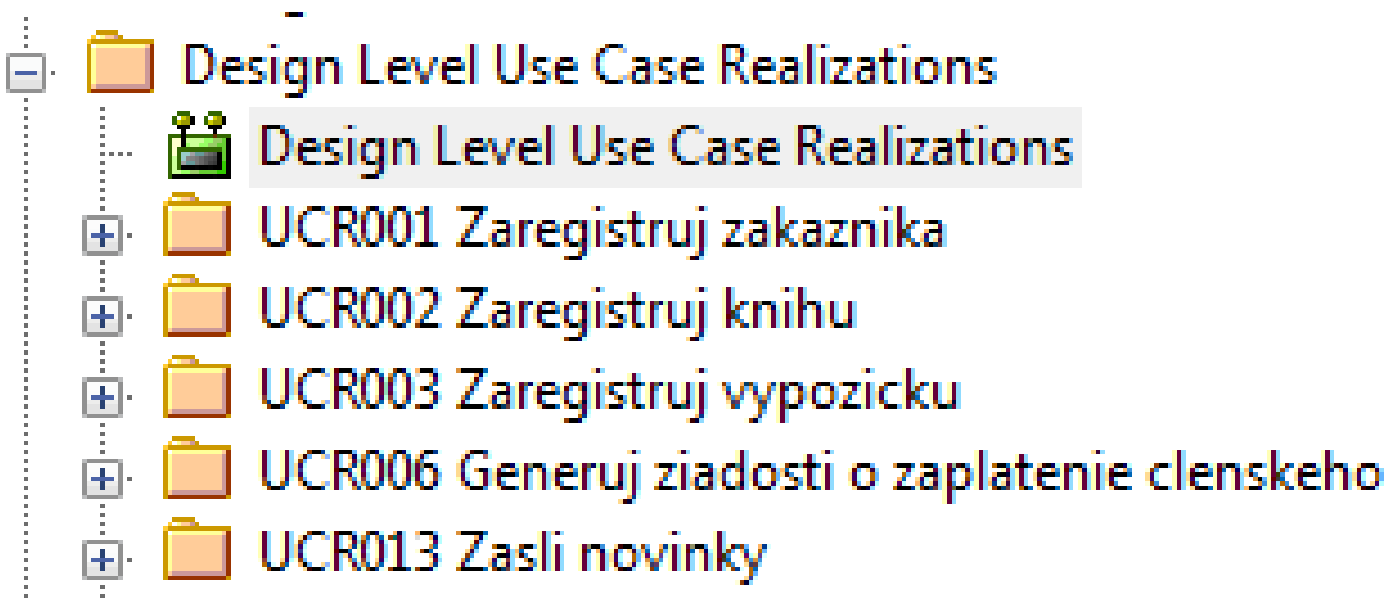
- Popis, ako je nejaký koncept realizovaný
- Najbežnejšie sú realizácie prípadov užitia a rozhraní



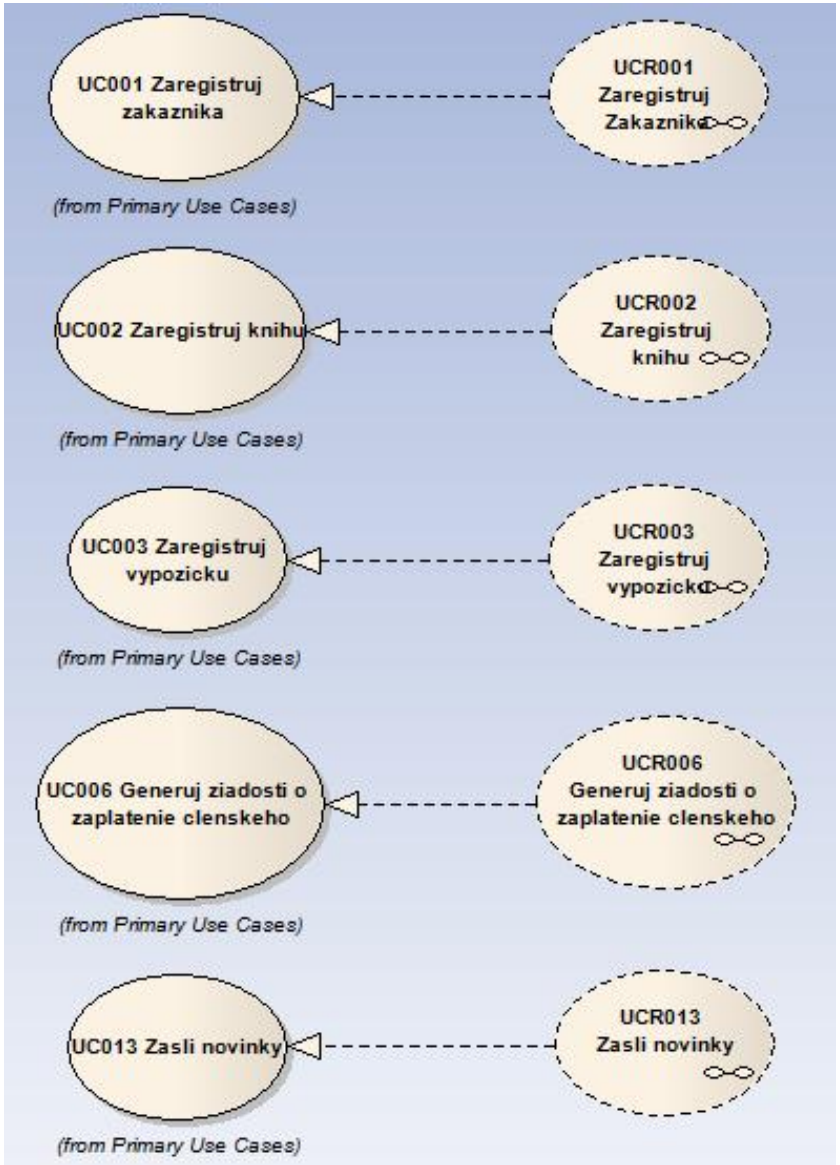
Realizácie prípadov užitia

- Pre každý prípad užitia sa vytvorí jeden balíček, v ktorom sú informácie o tom, ako je daný prípad užitia realizovaný
- Ktoré triedy sa prípadu užitia zúčastňujú
- Ako objekty pri realizácii komunikujú

Realizácie prípadov užitia: príklad



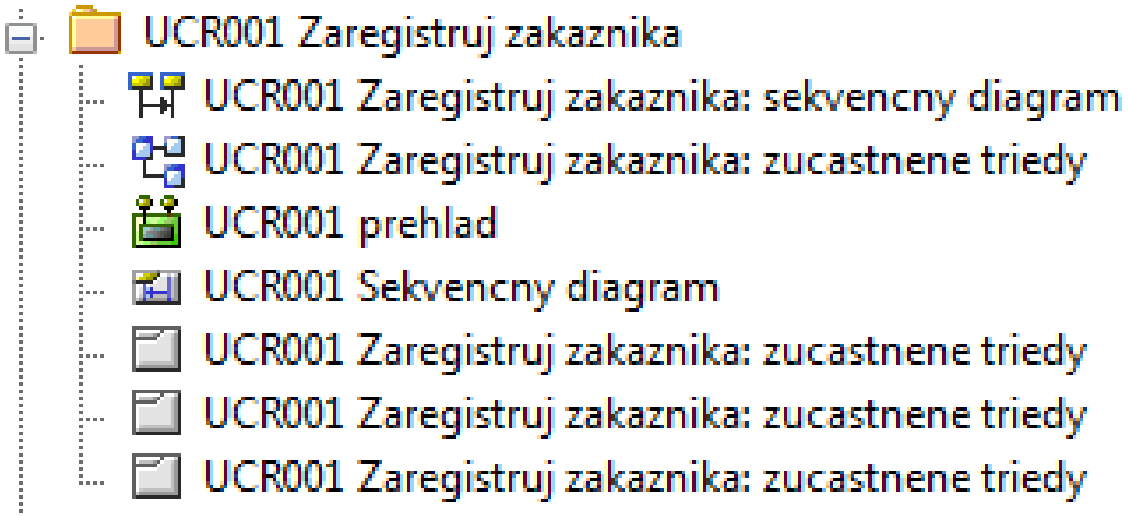
Realizácie prípadov užitia: prehľad



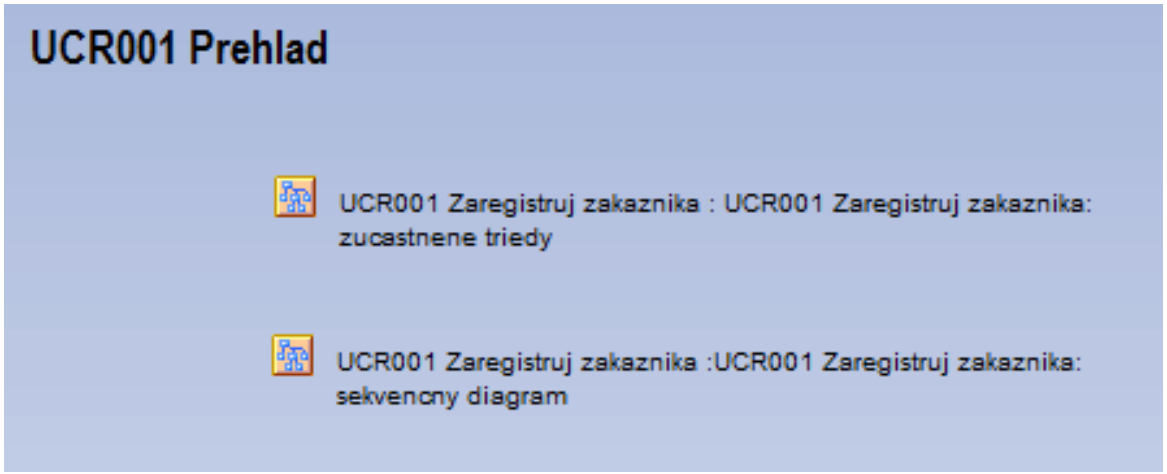
Realizácia prípadu užitia: diagramy

- Každá realizácia obsahuje minimálne
  - Diagram zúčastnených tried: VOPC (View Of Participating Classes)
  - Sekvenčný diagram
  - Prehľadový diagram (slúži ako default diagram a teda prístup do balíčka z iných diagramov)

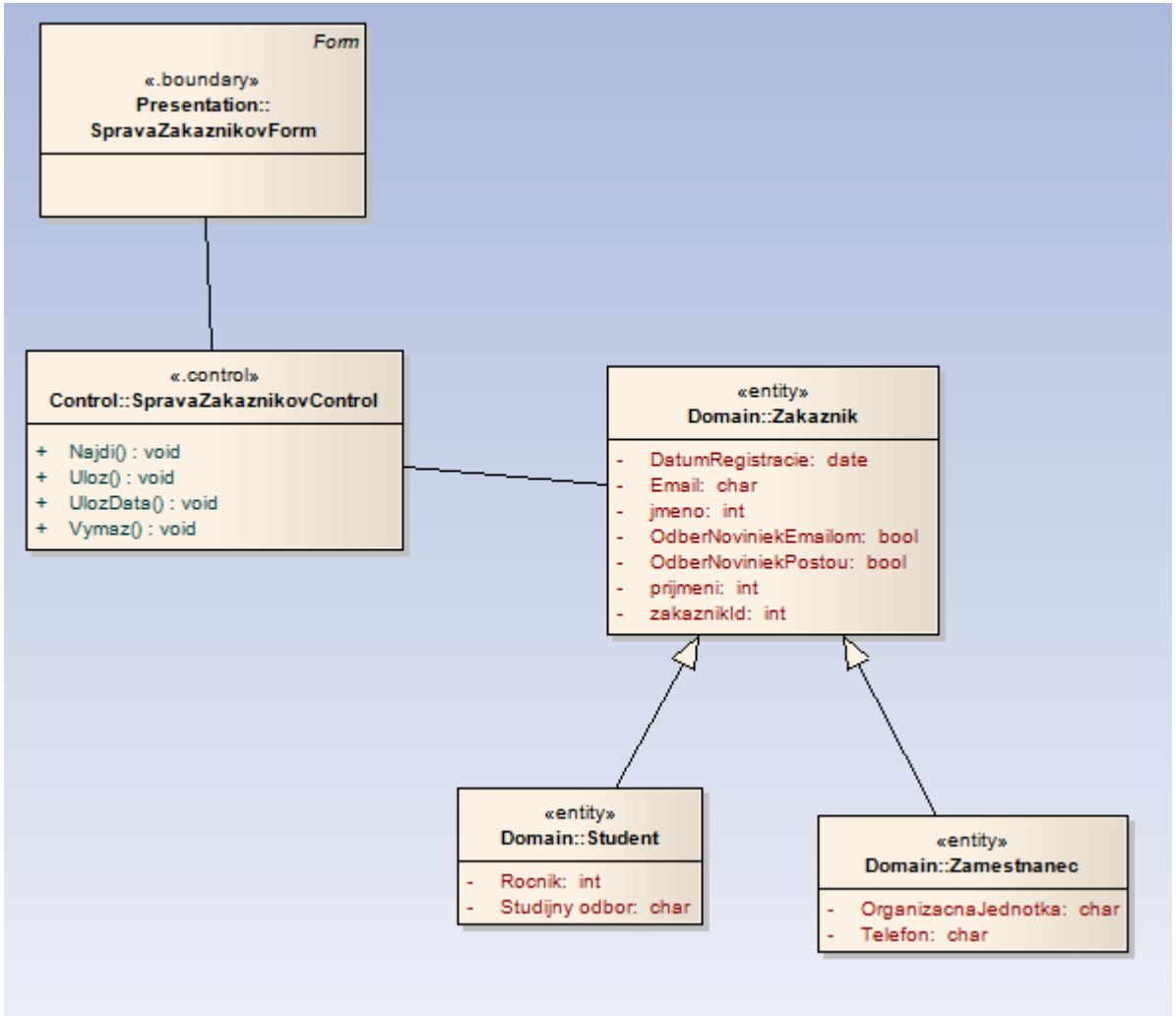
Príklad


UCR001 Zaregistruj zakaznika
UCR001 Zaregistruj zakaznika: sekvencny diagram
UCR001 Zaregistruj zakaznika: zucastnene triedy
UCR001 prehlad
UCR001 Sekvencny diagram
UCR001 Zaregistruj zakaznika: zucastnene triedy
UCR001 Zaregistruj zakaznika: zucastnene triedy
UCR001 Zaregistruj zakaznika: zucastnene triedy

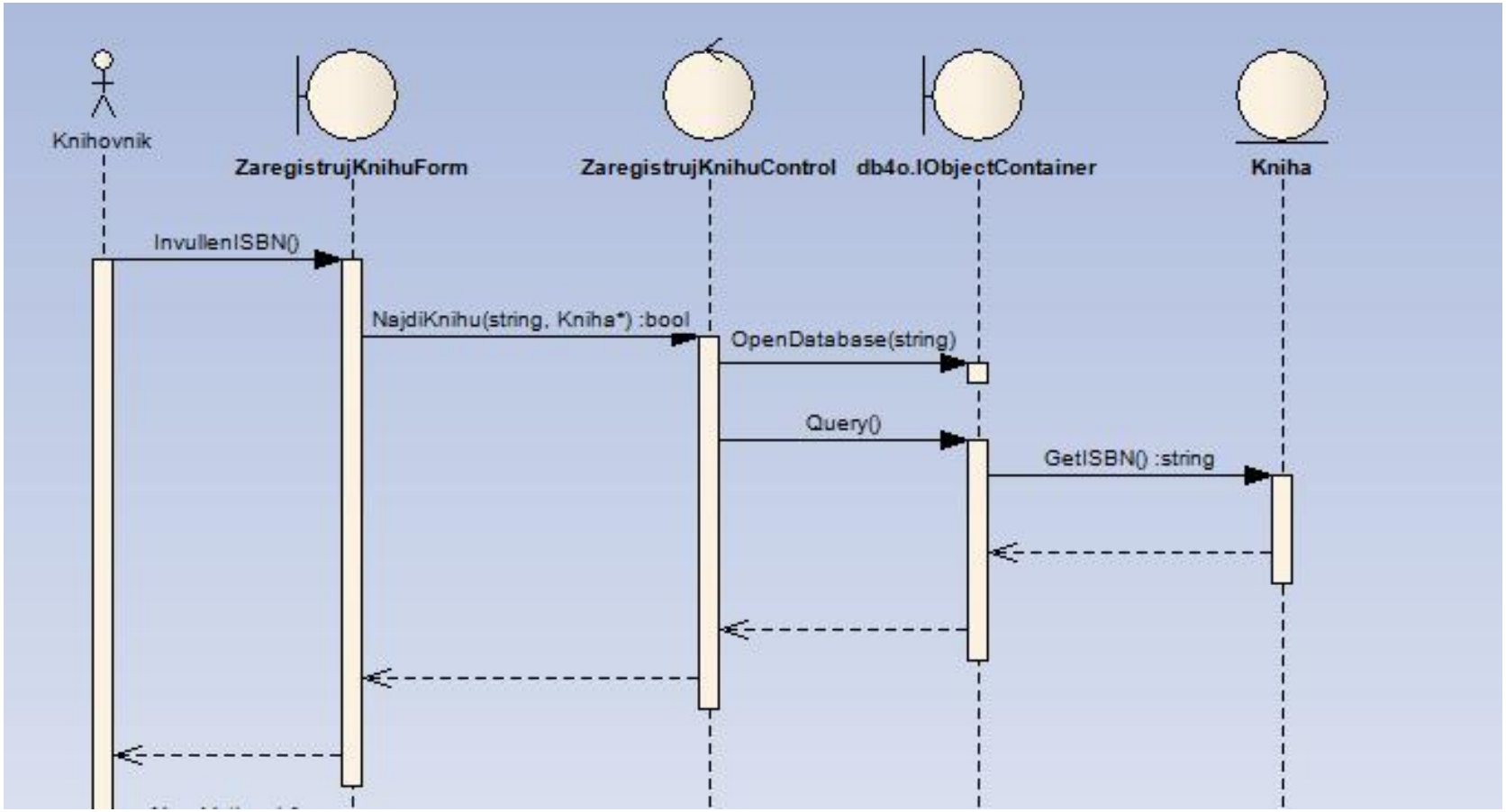
UCR: prehľad


UCR001 Prehlad
UCR001 Zaregistruj zakaznika : UCR001 Zaregistruj zakaznika: zucastnene triedy
UCR001 Zaregistruj zakaznika :UCR001 Zaregistruj zakaznika: sekvencny diagram

VOPC


Form
«.boundary»
Presentation::
SpravaZakaznikovForm
«.control»
Control::SpravaZakaznikovControl
+ Najdi() : void
+ Uloz() : void
+ UlozData() : void
+ Vymaz() : void
«entity»
Domain::Zakaznik
- DatumRegistracie: date
- Email: char
- jmeno: int
- OdberNoviniekEmailom: bool
- OdberNoviniekPostou: bool
- prijmeni: int
- zakaznikId: int
«entity»
Domain::Student
- Rocnik: int
- Studijny odbor: char
«entity»
Domain::Zamestnanec
- OrganizacnaJednotka: char
- Telefon: char

Sekvenčný diagram

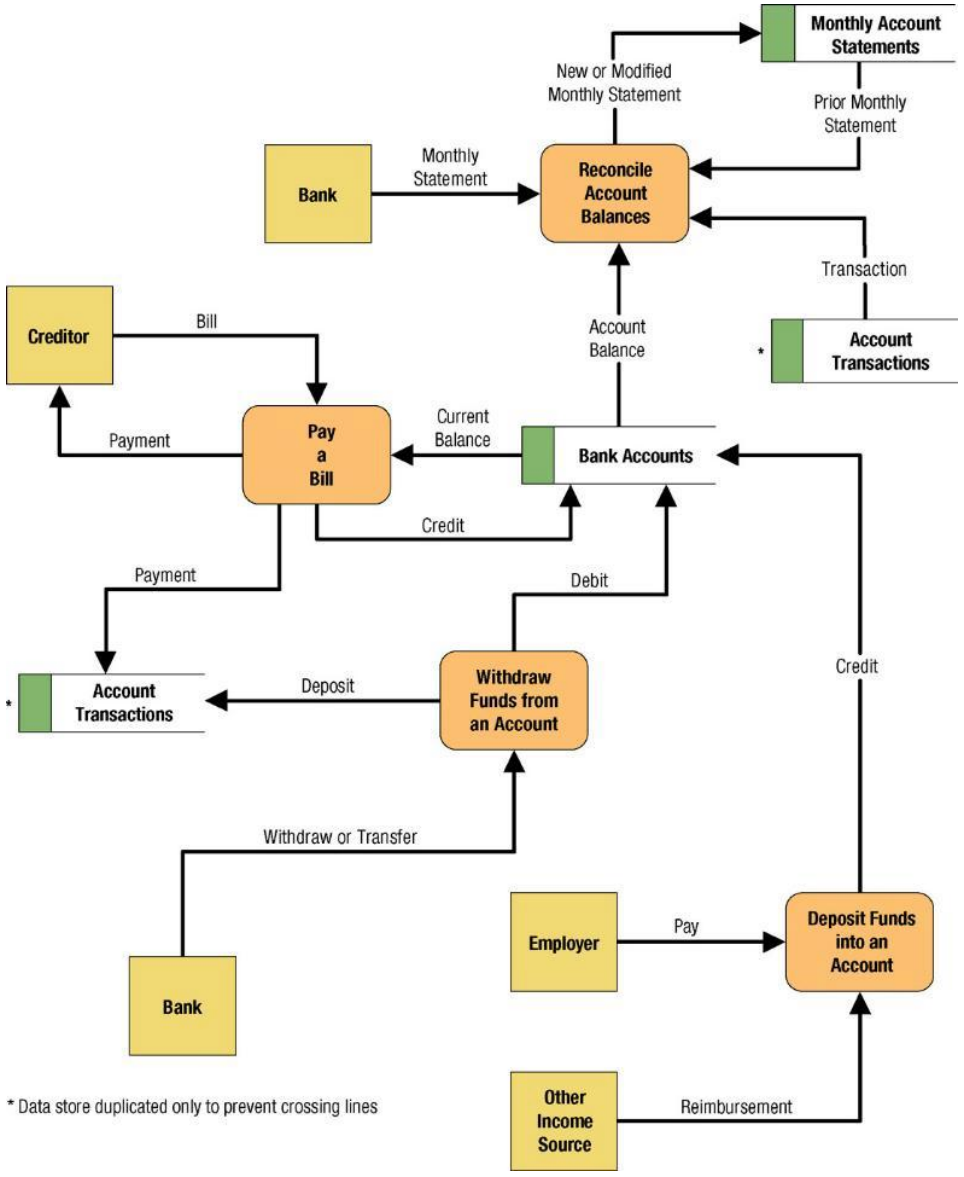


Diagramy dátových tokov (Data Flow Diagrams)

DFD: technika procesného modelovania

- **Modelovanie procesov :** technika používaná na organizovanie a dokumentovanie procesov daného systému.
  - Tok dát cez jednotlivé procesy
  - Logika
  - Predpisy a pravidlá
  - Procedúry
- DFD**:** procesný model, ktorý zobrazuje tok dát cez systém a spracovanie vykonávané systémom.
- DFD sa stali populárnym nástrojom pre business process redesign.

Príklad DFD. Viete ho prečítať?



Rozdiely medzi DFD a vývojovým diagramom (flowchart)

- Procesy na DFD sa môžu vykonávať paralelne (v tom istom čase)
  - Procesy na vývojovom diagrame sa vykonávajú po jednom
- DFD zobrazujú tok dát cez systém
  - Vývojové diagramy zobrazujú tok riadenie (sequence and transfer of control)
- Procesy na DFD môžu prebiehať v značne rozličných časoch (denne, týždenne, na požiadanie)
  - Procesy na vývojových diagramoch sú súčasťou jedného programu s konzistentným časovaním

Pojmy DFD

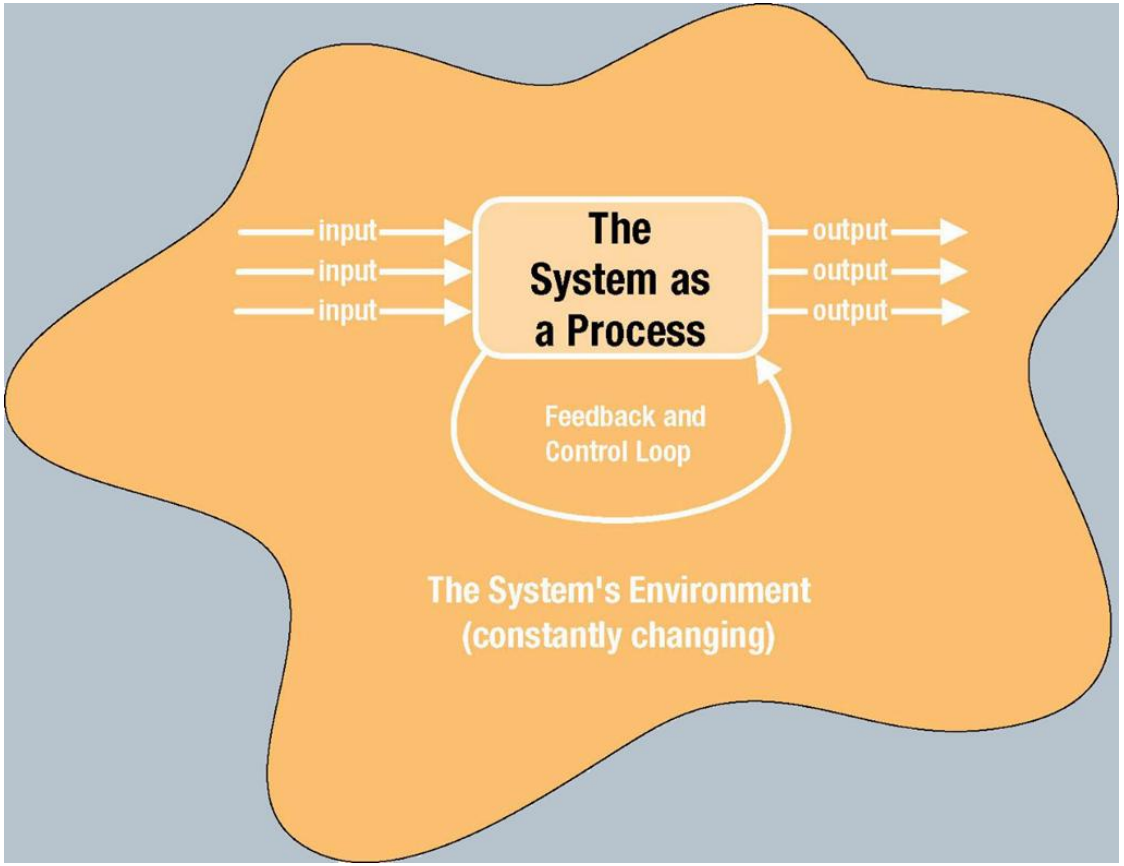
- DFD používajú nasledujúce pojmy *(inak povedané abstraktný jazyk DFD obsahuje nasledujúce objekty)*:
  - Proces
  - Dátový tok
  - Kontrolný tok
  - Data Store
  - Externá entita

Notácie

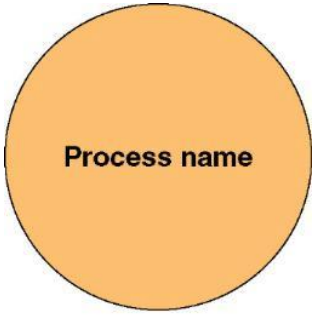
- Existuje niekoľko rôznych implementácií (notácií) jazyka DFD
- K najznámejším z nich patria:
  - Gane & Sarson
  - Yourdon / De Marco
- Existujú aj ďalšie, pričom niektoré formy sa v rôznych notáciách opakujú ale znamenajú niečo iné
- Je jedno, ktorú budete používať, ale používajte len jednu a nemiešajte ich dohromady

Procesy

Proces – aktivita vykonávaná systémom ako reakcia na vstupný tok dát alebo vstupnú podmienku za účelom zmeny stavu vstupujúceho objektu



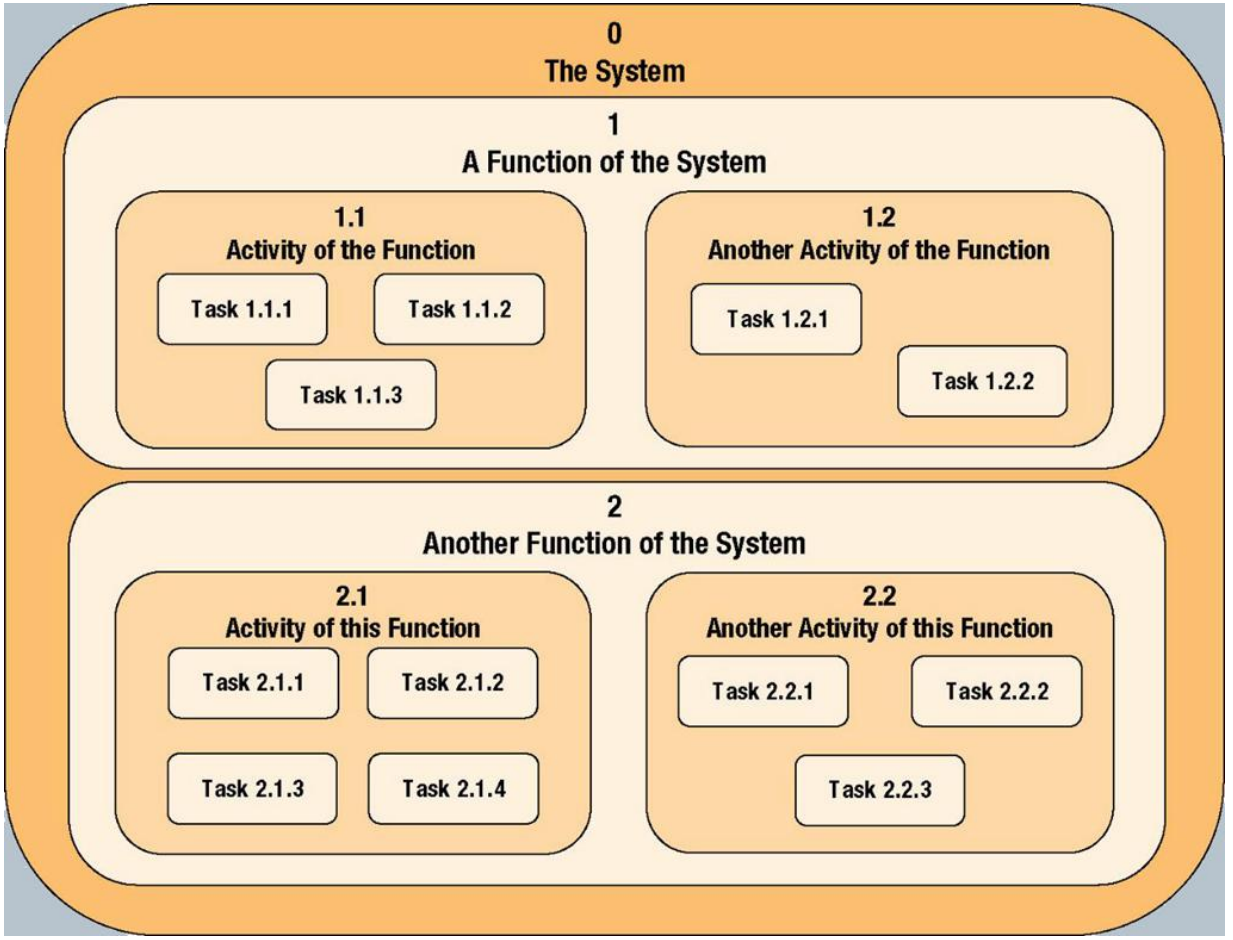
Gane & Sarson shape; used throughout this book



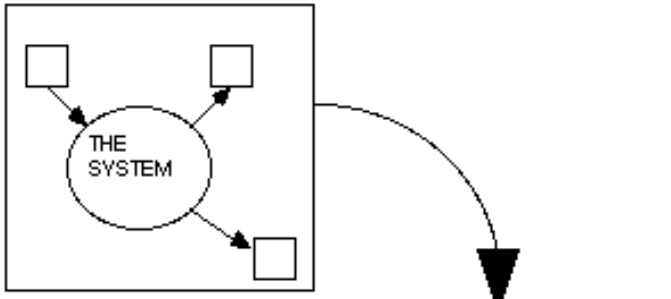
DeMarco/Yourdon shape

Dekompozícia procesov

Dekompozícia – rozdelenie systému na subsystémy. Každá úroveň abstrakcie ukazuje viac alebo menej detailov.



Viacúrovňový DFD



CONTEXT
DIAGRAM

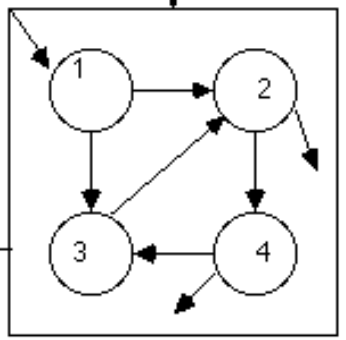


FIGURE 0

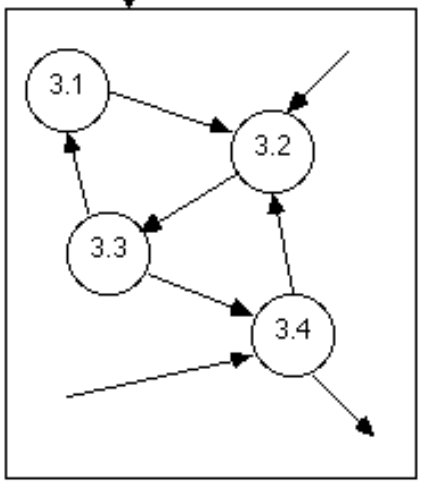
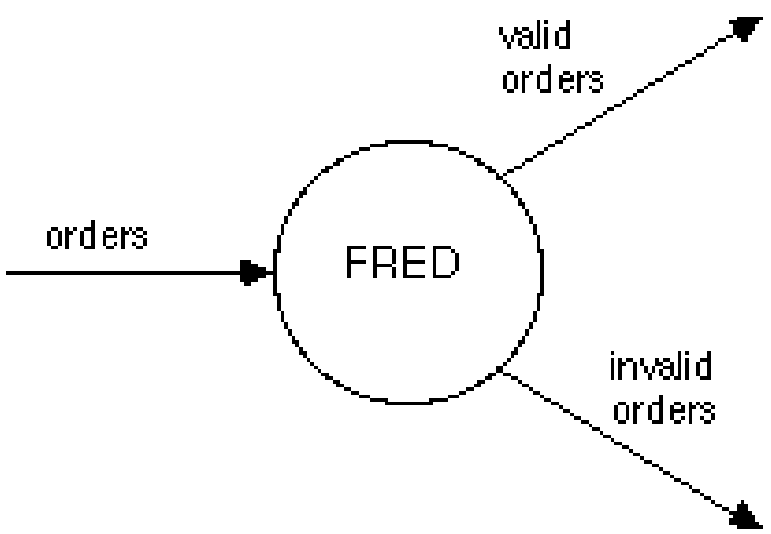


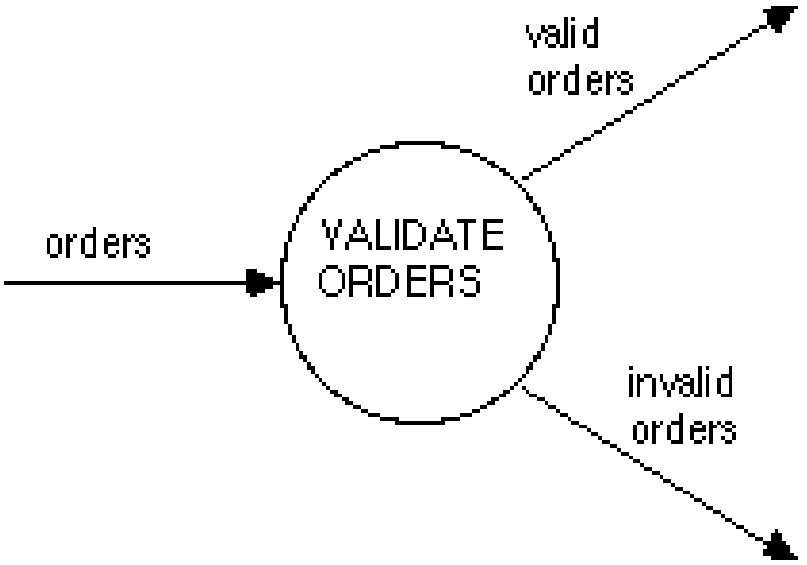
FIGURE 3

Pomenovanie procesov

Nesprávne



Správne



Ďalšie príklady názvov procesov

SPRÁVNE

- CALCULATE MISSILE TRAJECTORY
- PRODUCE INVENTORY REPORT
- VALIDATE PHONE NUMBER
- ASSIGN STUDENTS TO CLASS

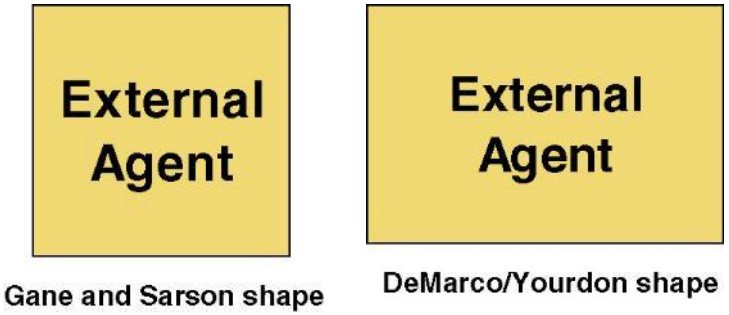
NESPRÁVNE

- DO STUFF
- MISCELLANEOUS FUNCTIONS
- NON-MISCELLANEOUS FUNCTIONS
- HANDLE INPUT
- TAKE CARE OF CUSTOMERS
- PROCESS DATA
- GENERAL EDIT

Externé entity (agenti)

Externá entita – osoba, organizačná jednotka alebo systém, ktorý je v interakcii s daným systémom

- Externé entity definujú hranice modelovaného systému.
- So zmenou rozsahu projektu sa externé entity môžu stať procesmi a naopak.
- Externé entity sú takmer vždy:
  - Pracoviská a oddelenia .
  - Externé organizácie
  - Iné informačné systémy
  - Koncoví užívatelia a manažéri
- Pomenovanie: podstatné meno v jednotnom čísle



Data stores

Data store – dáta uložené pre neskoršie použitie (perzistentné dáta)

- Často implementované ako súbory alebo databázy
- Data store predstavuje dáta v kľude, zatiaľ čo dátový tok predstavuje dáta v pohybe
- Väčšinou sú to
  - Osoby
  - Miesta
  - Objekty
  - Udalosti (informácie o nich)
  - Koncepty
- Pomenovanie podstatnými menami v množnom čísle

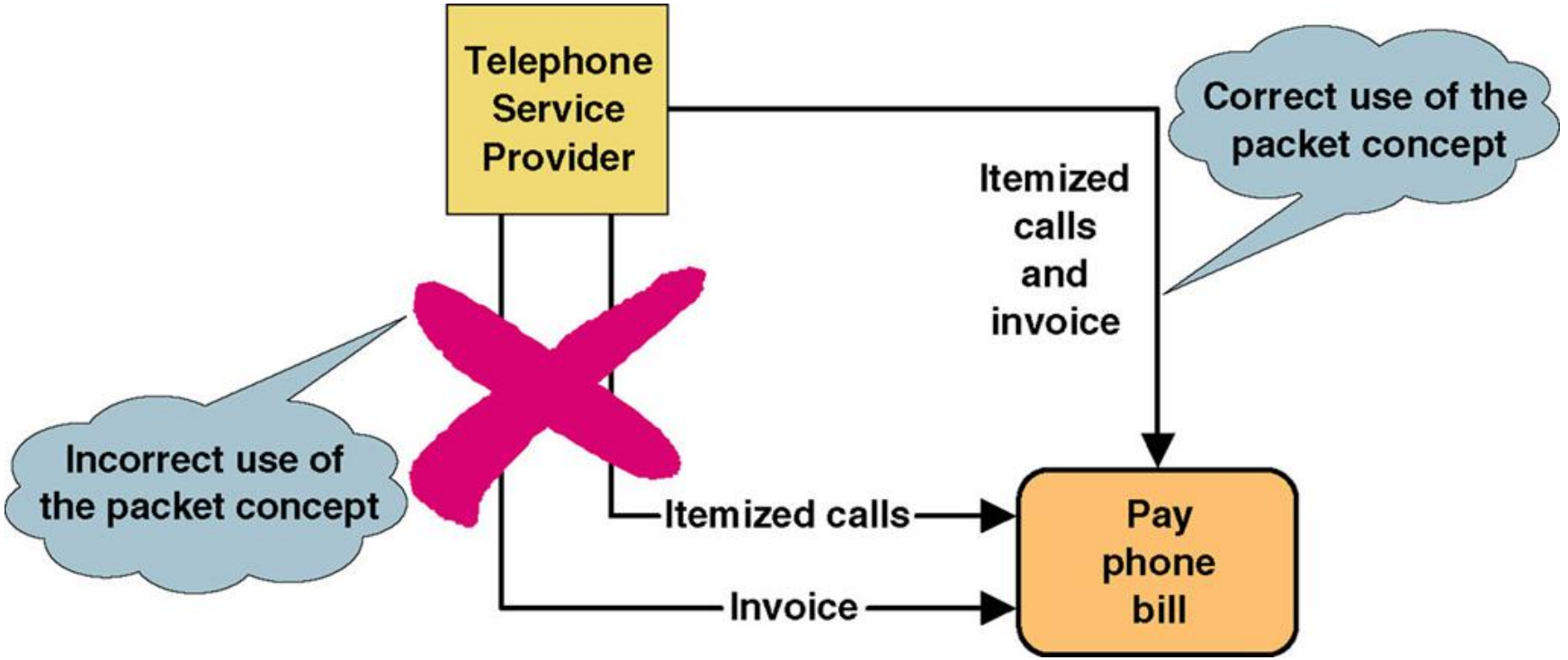
Gane and Sarson shape

DeMarco/Yourdon shape

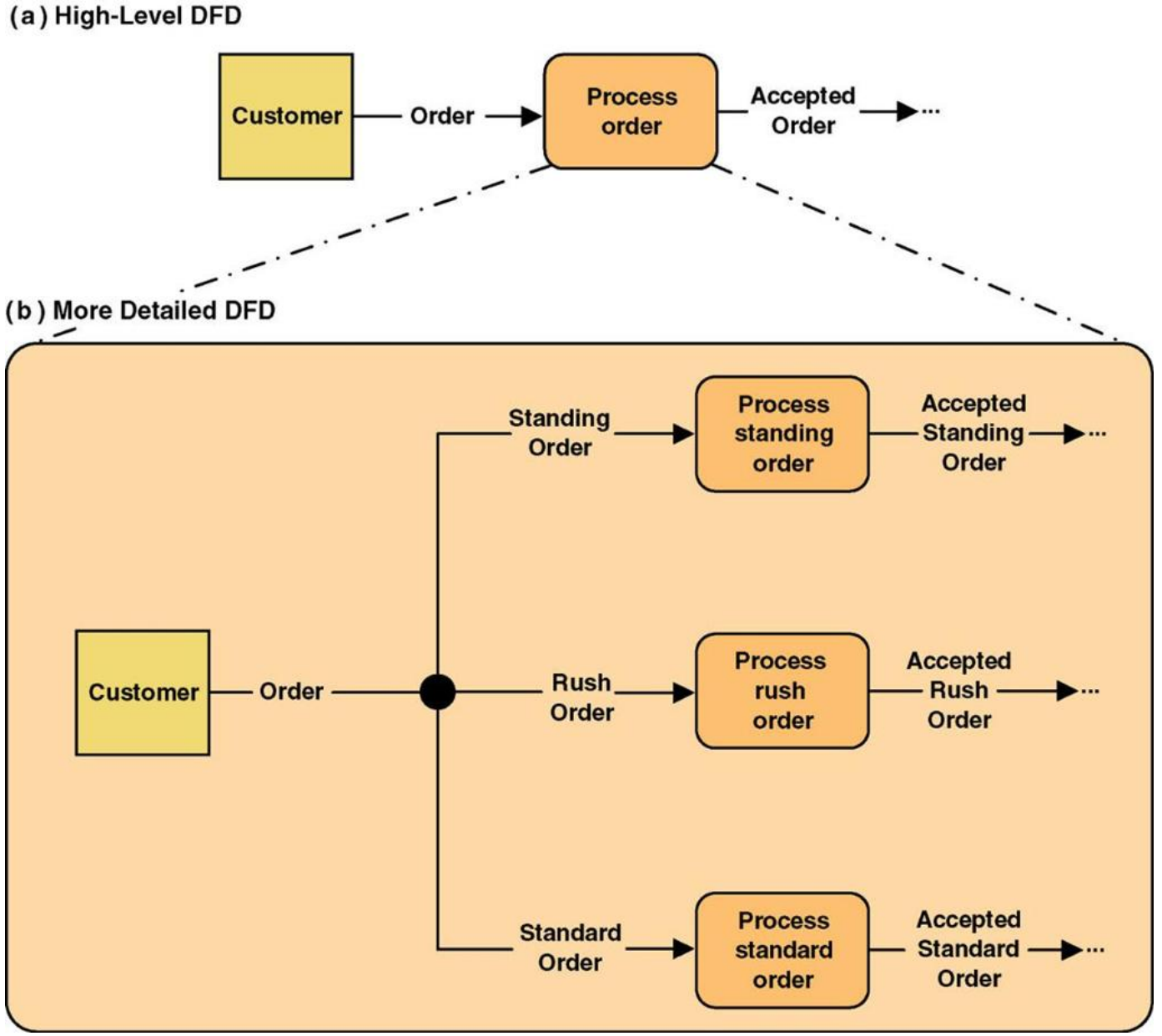
Dátové a riadiace toky

- **Dátový tok** – Dáta, ktoré sú vstupom alebo výstupom daného procesu.
  - Dátový tok môže tiež reprezentovať vytvorenie, čítanie, vymazanie alebo zmenu dát v súbore alebo databáze (data store)
- **Zložený dátový tok – pozostáva z viacerých dátových tokov**
- **Riadiaci tok (control flow) – udalosť, ktorá spúšťa nejaký proces**
  - Na DFD sa často nevyskytuje

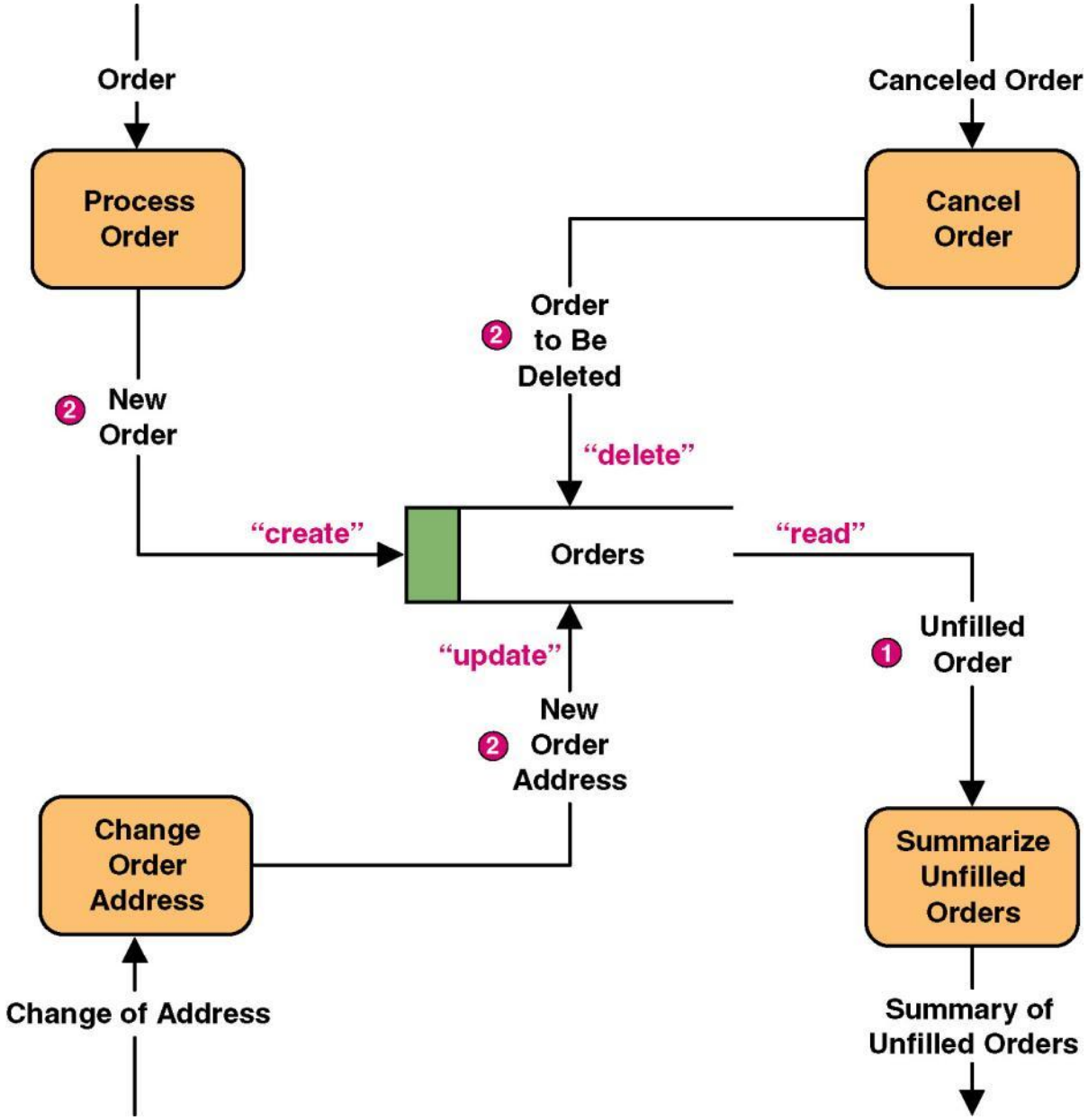
Pakety dátových tokov



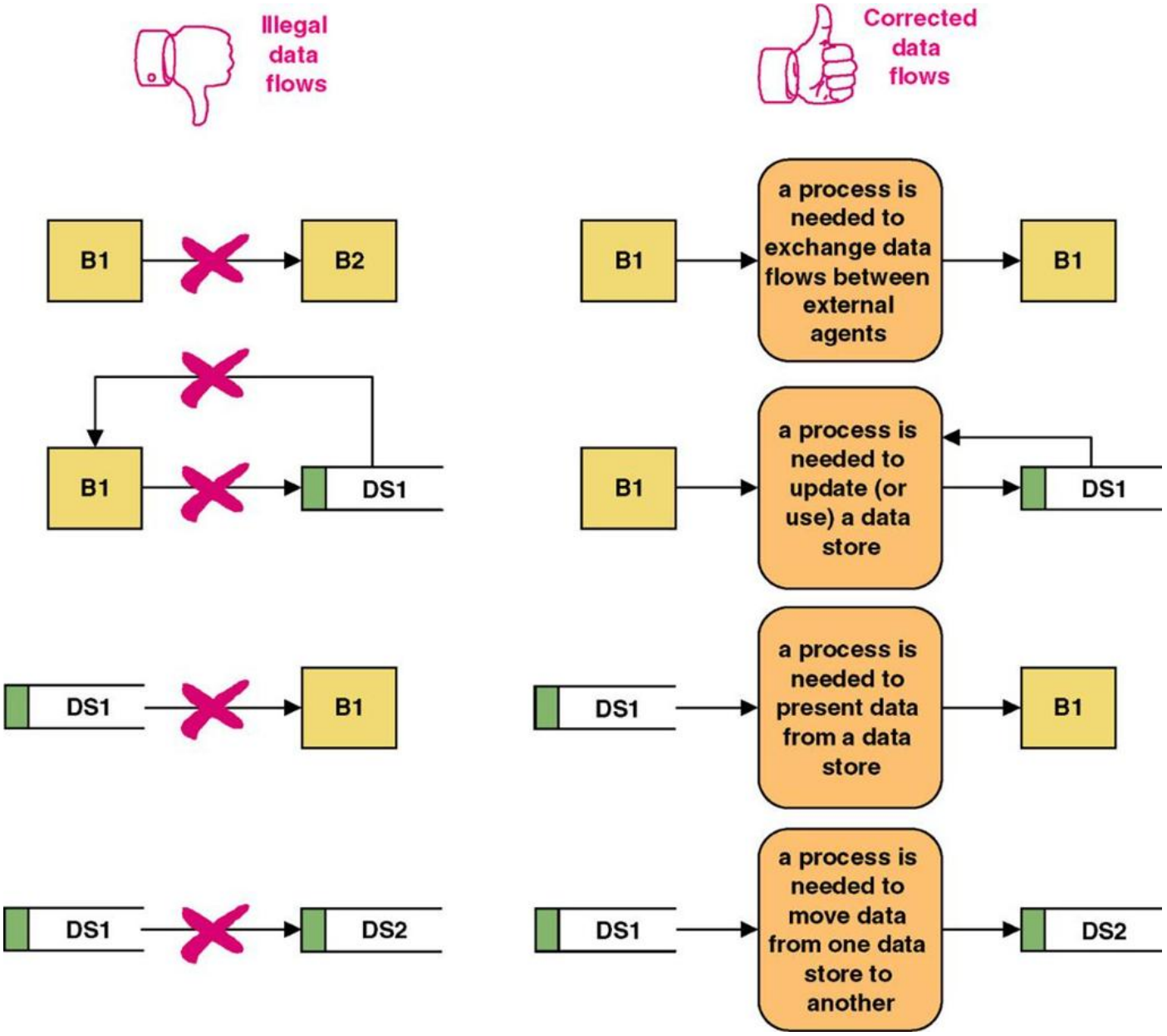
Elementárne a zložené dátové toky



Dátové toky medzi procesmi a data stormi



Ilegálne dátové toky



Štruktúra dátového toku

Atribút – najmenší kúsok dát týkajúci sa daného byznysu, ktorý užívateľovi dáva zmysel

Dátová štruktúra – špecifické usporiadanie dátových atribútov ktoré definuje jeden výskyt dátového toku (inštanciu)

- Dátové toky bývajú definované pomocou nasledujúcich štruktúr
  - Sekvencia alebo skupina dátových atribútov
  - Selekcia atribútov z danej množiny
  - Opakovanie jedného alebo viacerých atribútov

Príklad štruktúry dátového toku

DATA STRUCTURE

```
ORDER=
      ORDER NUMBER +
      ORDER DATE+
      [ PERSONAL CUSTOMER NUMBER,
        CORPORATE ACCOUNT NUMBER]+
      SHIPPING ADDRESS=ADDRESS+
      (BILLING ADDRESS=ADDRESS)+
      1 {PRODUCT NUMBER+
          PRODUCT DESCRIPTION+
          QUANTITY ORDERED+
          PRODUCT PRICE+
          PRODUCT PRICE SOURCE+
          EXTENDED PRICE } N+
      SUM OF EXTENDED PRICES+
      PREPAID AMOUNT+
      (CREDIT CARD NUMBER+EXPIRATION DATE)
      (QUOTE NUMBER)

ADDRESS=
      (POST OFFICE BOX NUMBER)+
      STREET ADDRESS+
      CITY+
      [STATE, MUNICIPALITY]+
      (COUNTRY)+
```

ENGLISH ENTERPRETATION

An instance of ORDER consists of:
- ORDER NUMBER and
- ORDER DATE and
- Either PERSONAL CUSTOMER NUMBER
  - or CORPORATE ACCOUNT NUMBER
- and SHIPPING ADDRESS (which is equivalent
  - to ADDRESS)
- and optionally: BILLING ADDRESS (which is
  - equivalent to ADDRESS)
- and one or more instances of:
  - PRODUCT NUMBER and
  - PRODUCT DESCRIPTION and
  - QUANTITY ORDERED and
  - PRODUCT PRICE and
  - PRODUCT PRICE SOURCE and
  - EXTENDED PRICE
- and SUM OF EXTENDED PRICES and
- PREPAID AMOUNT and
- optionally: both CREDIT CARD NUMBER and
  - EXPIRATION DATE

An instance of ADDRESS consists of:
- optionally: POST OFFICE BOX NUMBER and
- STREET ADDRESS and
- CITY and
- Either STATE or MUNICIPALITY
- and optionally: COUNTRY
- and POSTAL CODE

Príklady rôznych typov štruktúr

Data Structure	Format by Example (relevant portion is **boldfaced**)	English Interpretation (relevant portion is **boldfaced**)
Sequence of Attributes - The sequence data structure indicates one or more attributes that may (or must) be included in a data flow.	WAGE AND TAX STATEMENT= **TAXPAYER IDENTIFICATION NUMBER+** **TAXPAYER NAME+** **TAXPAYER ADDRESS+** **WAGES, TIPS, AND COMPENSATION+** **FEDERAL TAX WITHHELD+…**	An instance of WAGE AND TAX STATEMENTS consists of: **TAXPAYER IDENTIFICATION NUMBER and** **TAXPAYER NAME and** **TAXPAYER ADDRESS and** **WAGES, TIPS AND COMPENSATION and** **FEDERAL TAX WITHHELD and…**
Selection of Attributes - The selection data structure allows you to show situations where different sets of attributes describe different instances of the data flow.	ORDER= **(PERSONAL CUSTOMER NUMBER,** **CORPORATE ACCOUNT NUMBER)+** ORDER DATE+…	An instance or ORDER consists of: **Either PERSONAL CUSTOMER NUMBER or** **CORPORATE ACCOUNT NUMBER;** and ORDER DATE and…

Ešte jeden príklad štruktúry

Data Structure	Format by Example (relevant portion is **boldfaced**	English Interpretation (relevant portion is **boldfaced**)
Repetition of Attributes - The repetition data structure is used to set off a data attribute or group of data attributes that may (or must) repeat themselves a specific number of time for a single instance of the data flow. The minimum number of repetitions is usually *zero* or *one*. The maximum number of repetitions may be specified as "n" meaning "many" where the actual number of instances varies for each instance of the data flow.	POLICY NUMBER+ POLICYHOLDER NAME+ POLICY HOLDER ADDRESS+ **0 {DEPENDENT NAME+** **DEPENDENT'S RELATIONSHIP} N+** **1 {EXPENSE DESCRIPTION+** **SERVICE PROVIDER+** **EXPENSE AMOUNT} N	An instance of CLAIM consists of: POLICY NUMBER and POLICYHOLDER NAME and POLICYHOLDER ADDRESS and **zero or more instance of:** **DEPENDENT NAME and** **DEPENDENT'S RELATIONSHIP and** **one or more instances of:** **EXPENSE DESCRIPTION and** **SERVICE PROVIDER and** **EXPENSE ACCOUNT**

Rozdeľovanie a spájanie dátových tokov

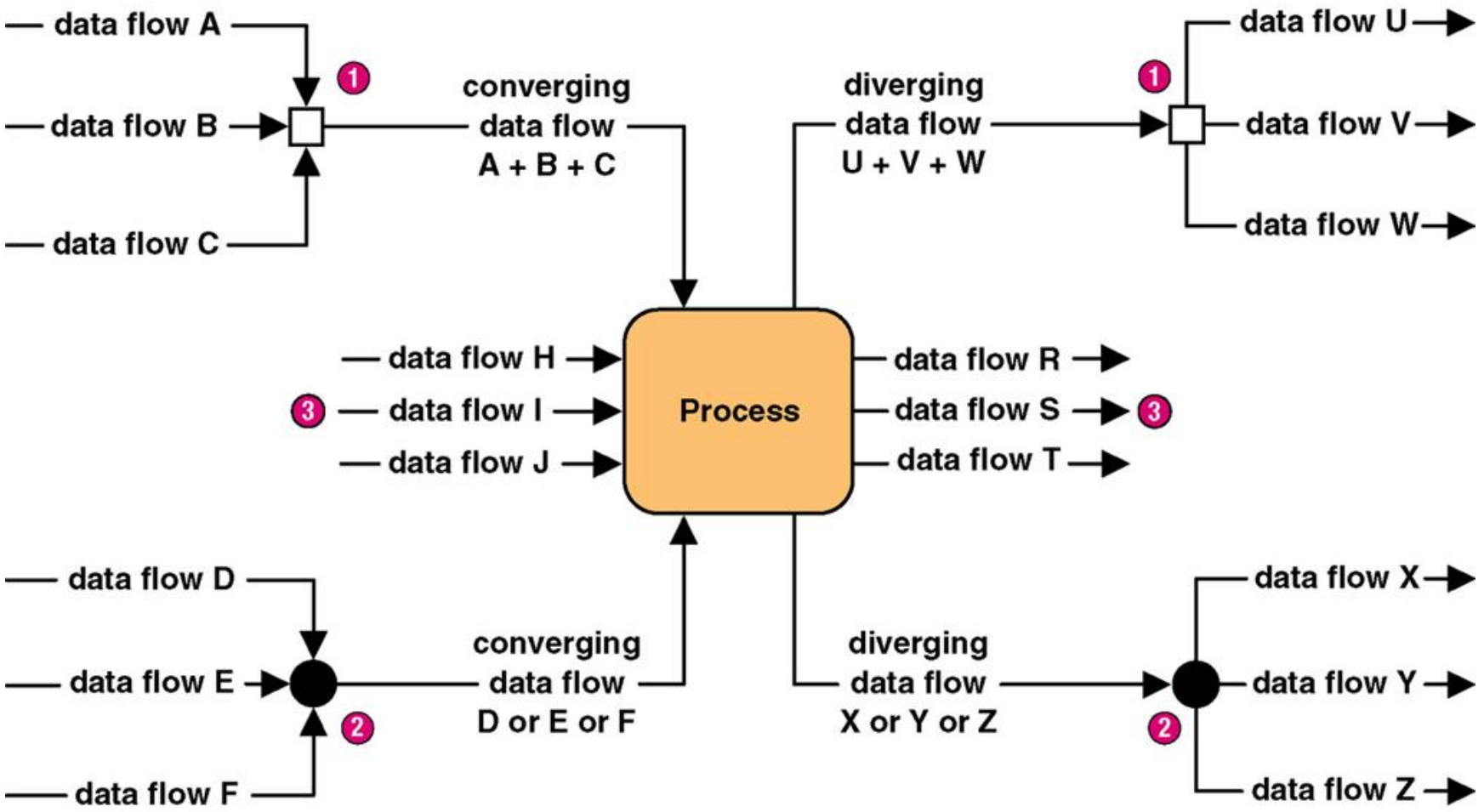
Rozdeľovanie dátových tokov – do niekoľkých menších tokov

- Naznačuje, že tok vzniká ako jeden tok, ale je rozposielaný do rozličných miest
- Hodí sa tiež na ukázanie toho, že niekoľko kópií toho istého toku putuje na rôzne miesta

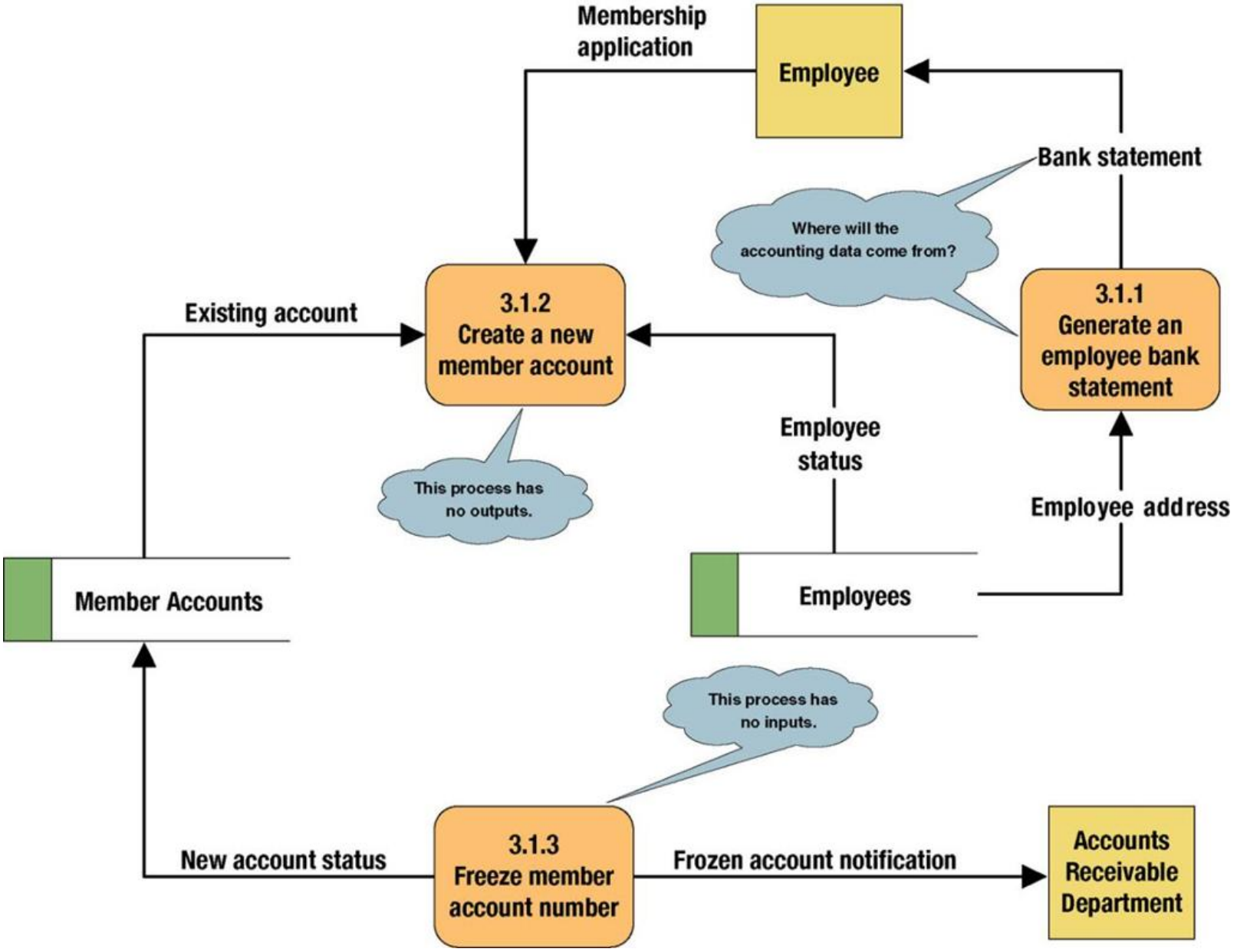
Spájanie dátových tokov – z viacerých tokov vzniká jeden paket

- Znamená, že dáta z rôznych zdrojov môžu (alebo musia) vstúpiť ako jeden balík do daného procesu na spracovanie

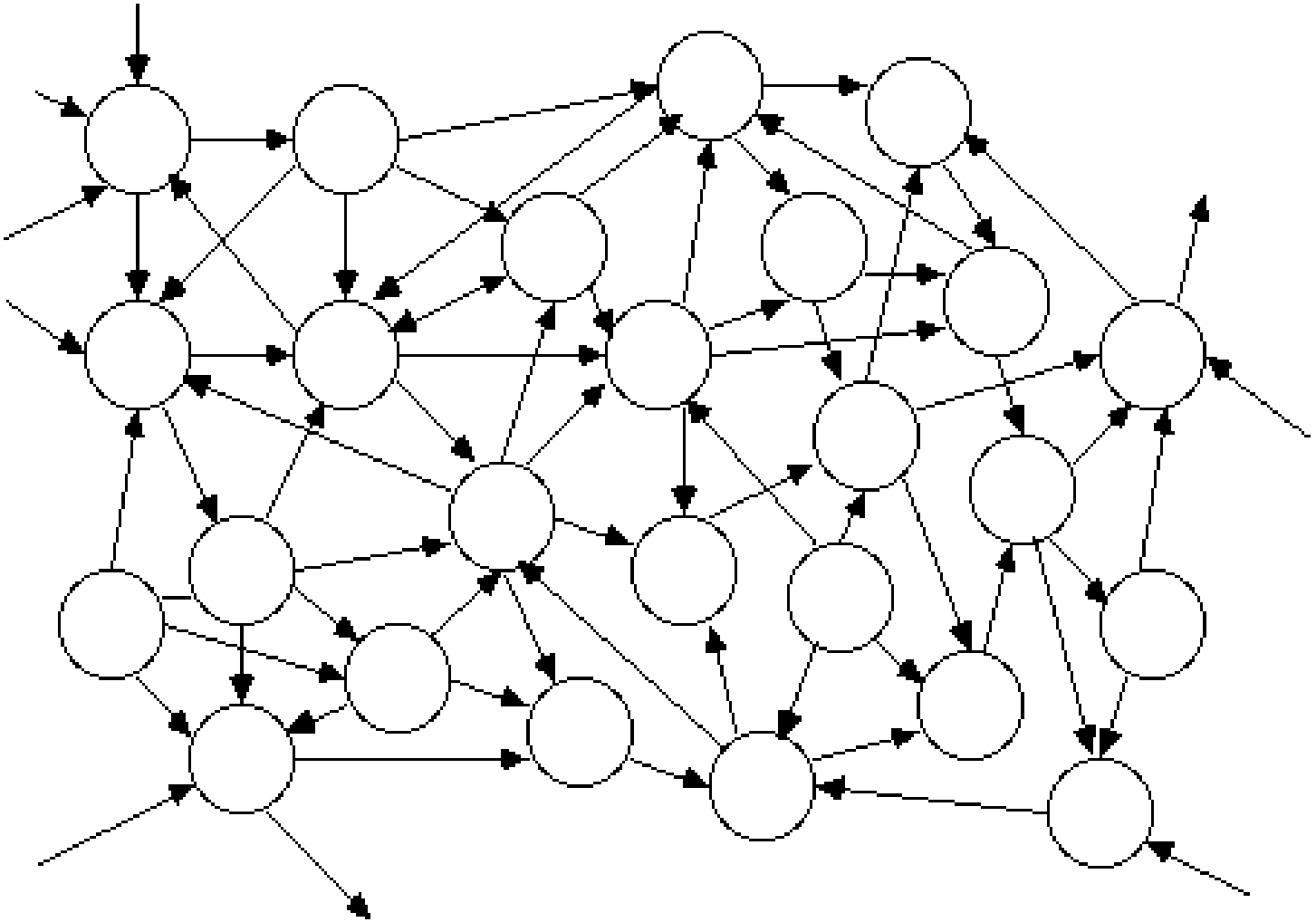
Príklad spájanie a rozdeľovania tokov



Bežné chyby pri tvorbe DFD



Iný príklad chybného DFD

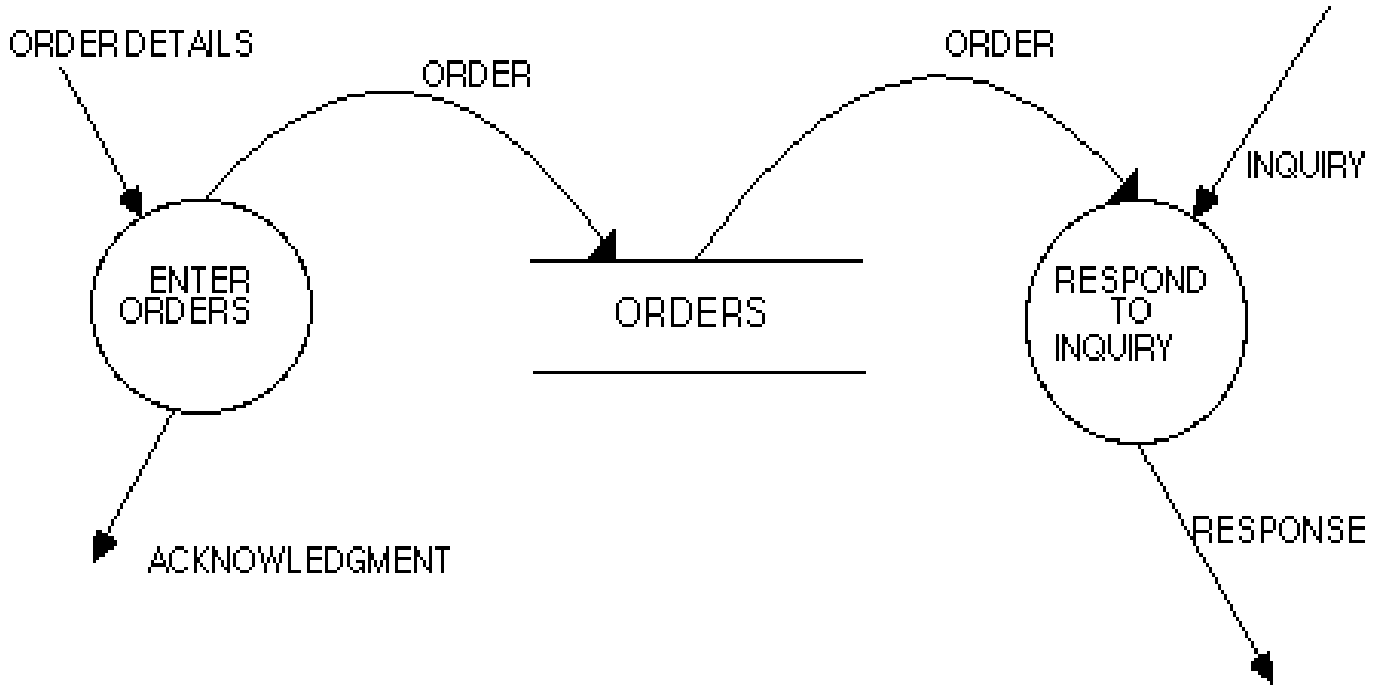


Čo DFD nezobrazujú

- Akým spôsobom vstupuje tok do procesu? Žiada o to proces? Alebo sa dáta presúvajú samy od seba?
- Viac vstupov: Musia byť všetky vstupy dostupné naraz? Alebo stačí jeden? Musí každému vstupu zodpovedať jeden výstup?
- Pri viacerých výstupoch: v akom poradí sa generujú

Komunikácia procesov cez data store

Umožňuje asynchrónne fungovanie procesov



DFD v SSADM

- SSADM: Structured System Analysis and Design Methodology
- SSADM používa dva pohľady na softwarový systém
  - Dátový pohľad (Dátové modely, ERD)
  - Funkčný pohľad (Procesy, DFD)

Dátové modely

- Dátové modely reprezentujú dáta, ktoré sú niekde uložené
  - Najčastejšie v databáze
  - Dátové modely teda zobrazujú štruktúru databázy

Funkčný pohľad

- Modeluje aké funkcie systém vykonáva
- Systém sa dá vnímať ako množina funkcií
- Funkcia je definovaná pomocou:
  - Vstupov
  - Výstupov
  - Predpisu ako sa vstup transformuje na výstup

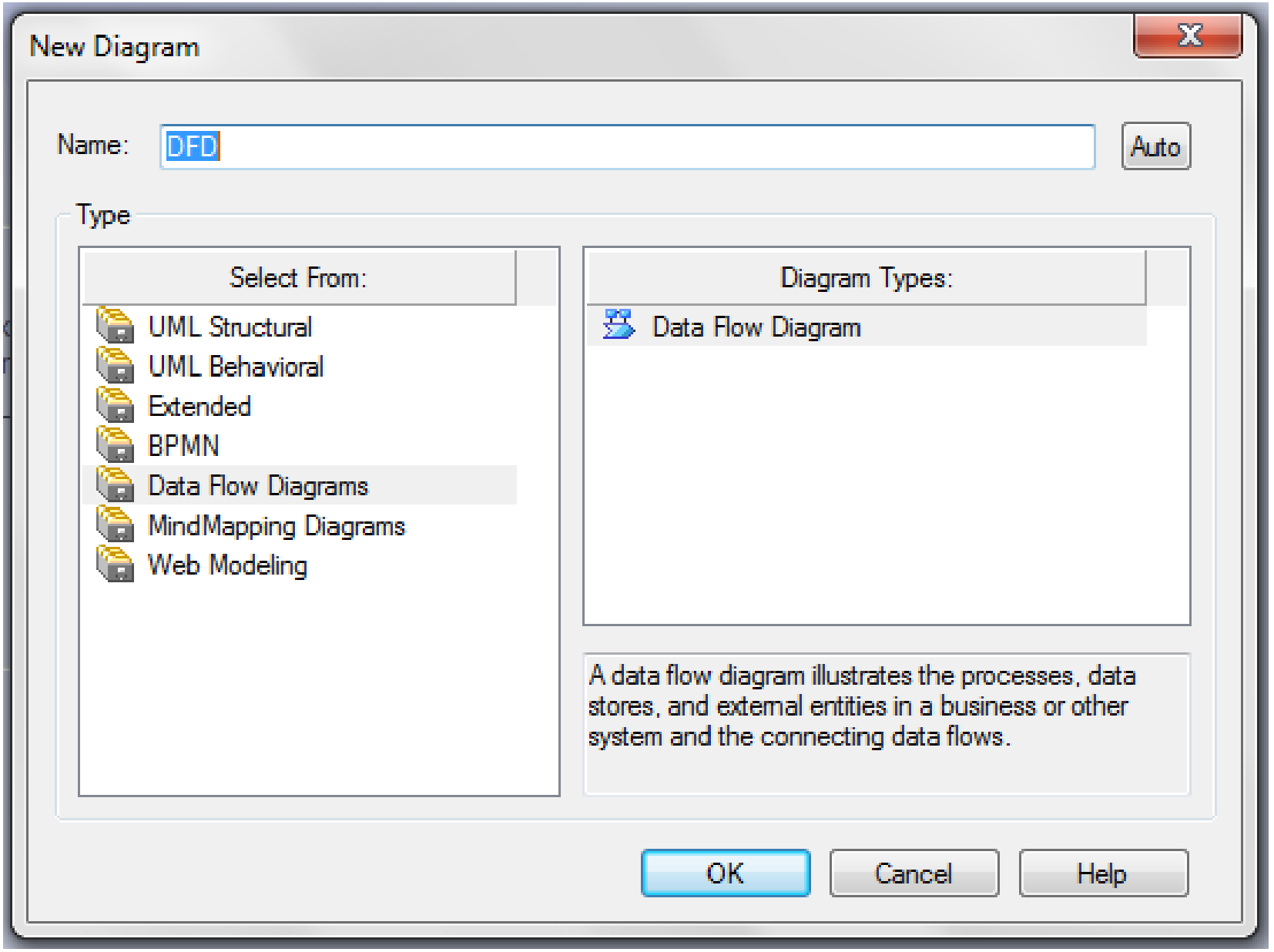
Ako je to na seba naviazané

- Vstupy a výstupy funkcií sú dáta
- Funkcie komunikujú medzi sebou (a s data stormi) pomocou vstupov a výstupov: dátové toky
- Táto vlastnosť bola veľmi dobre využívaná v CASE nástrojoch pre SSADM

Podpora DFD v EA

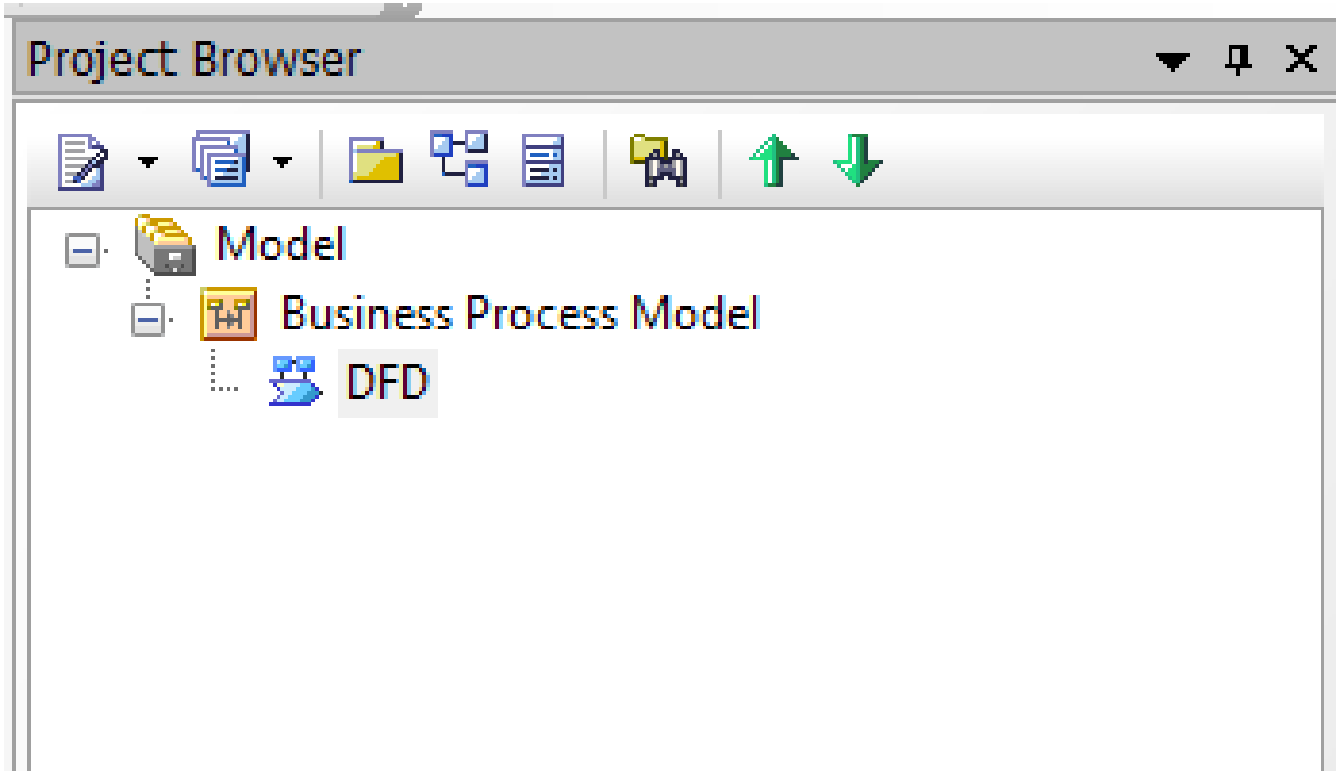
- Enterprise Architect poskytuje jednoduchú podporu pre DFD
- Používa Yourdonovu notáciu
- Neumožňuje (priamo) vytvárať hierarchické štruktúry procesov
- Neumožňuje (priamo) definovať štruktúry dátových tokov

Vytvorenie DFD

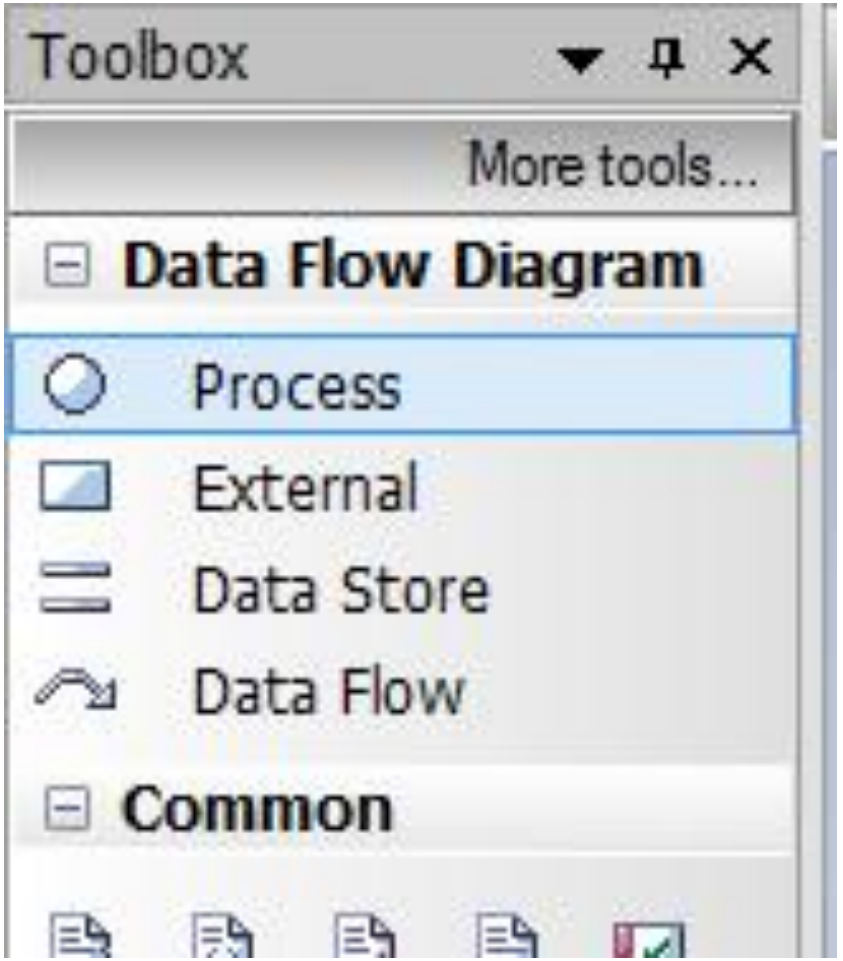


DFD: súčasť procesného modelu

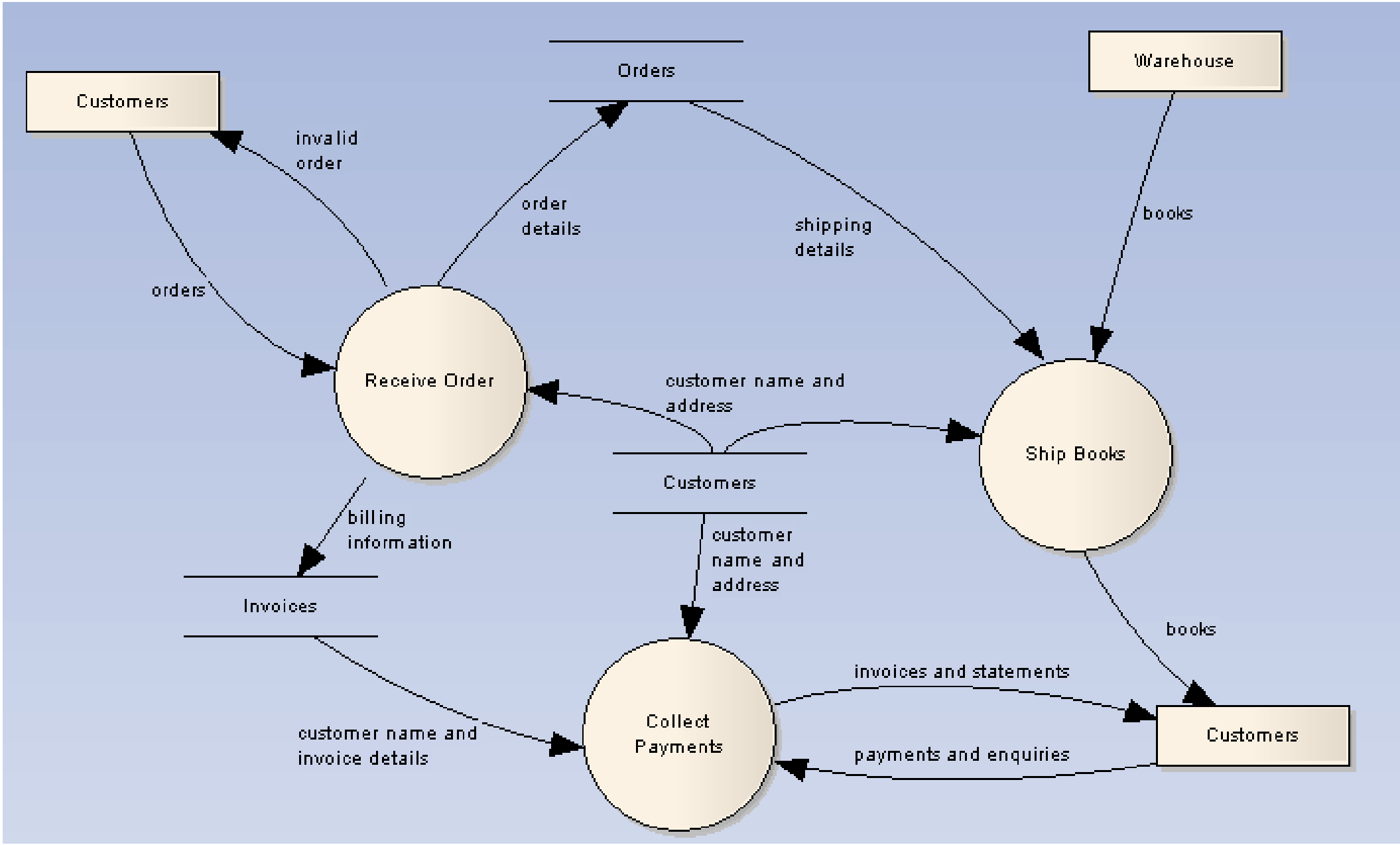
- DFD sú modely procesov. Je preto dobré umiestniť ich do balíčka „Procesný model“
- EA podporuje štandardne „Business Process Model“



Komponenty DFD



Príklad


Customers
Orders
Warehouse
invalid order
order details
shipping details
books
orders
Receive Order
customer name and address
Ship Books
Customers
billing information
customer name and address
Invoices
books
invoices and statements
Collect Payments
Customers
customer name and invoice details
payments and enquiries

DFD a UML?

- DFD a UML sú súčasti rozličných modelovacích filozofií a preto viac-menej nekompatibilné
- UML je „in“ a DFD sa javia ako zastaralé
- UML a metodiky UP a RUP v podstate nahradili štrukturované metodiky a k nim patriace modely

Napriek tomu

- Napriek tomu sa s DFD a ERD stále stretávame
- Tieto modely často dopĺňajú modely tried a iné objektové modely
- Nástroje pre objektové modelovanie poskytujú podporu aj pre DFD a ERD

Čím to je?

- Podľa môjho názoru dopĺňajú modely UP a RUP tam, kde tieto nepostačujú:
  - Modelovanie požiadaviek pre systémy, kde je málo interakcie medzi systémom a užívateľom
  - Vytvorenie uceleného pohľadu na perzistentné dáta uložené v relačnej databáze

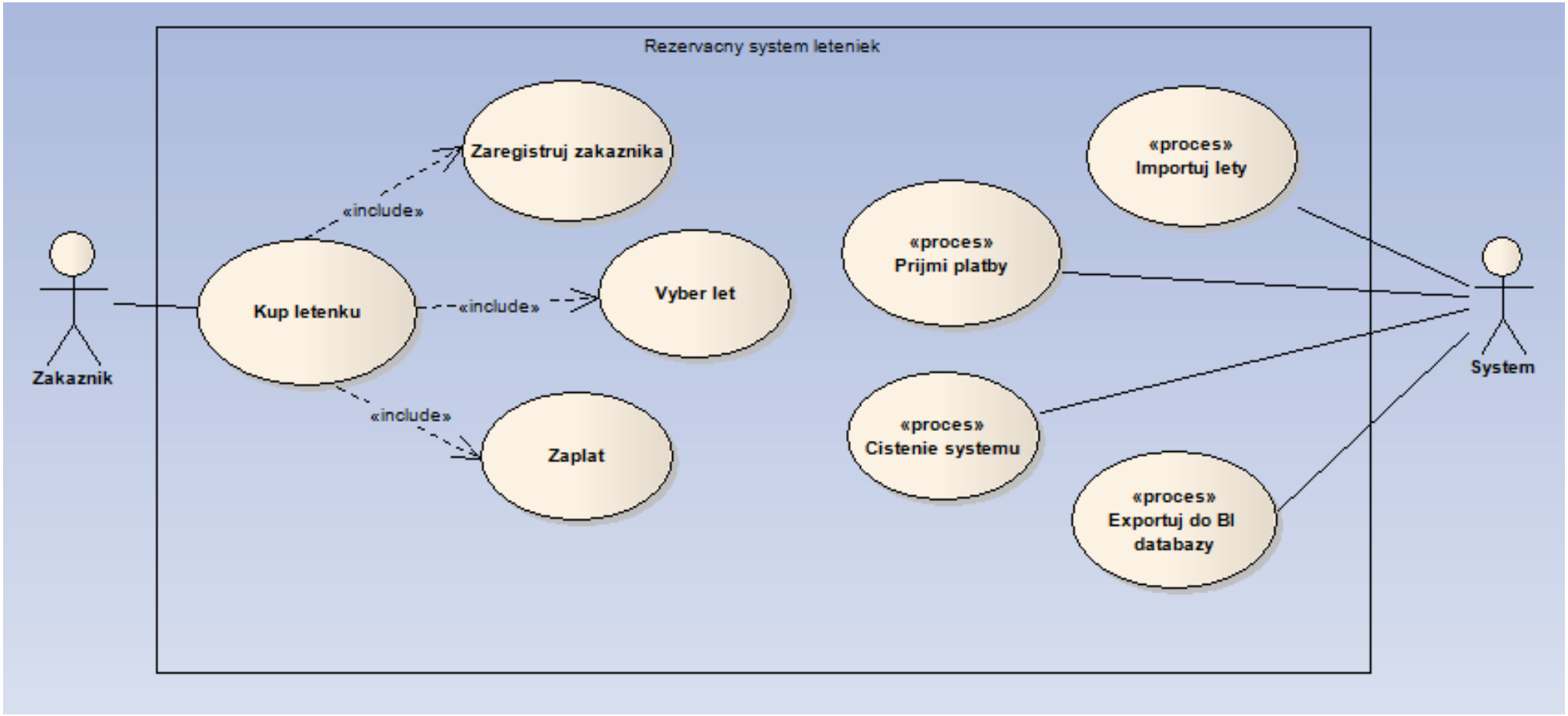
Modelovanie požiadaviek

- Jediný východzí bod modelovania v UP a RUP sú prípady užitia
- Prípady užitia vychádzajú z interakcie medzi užívateľom a systémom
- Ako však modelovať systémy kde takáto interakcia neexistuje?
- Samozrejme je možné použiť (zneužiť) prípady užitia tak ako to predpisujú metodiky
- Tento prístup však nevedie k uspokojivým výsledkom!

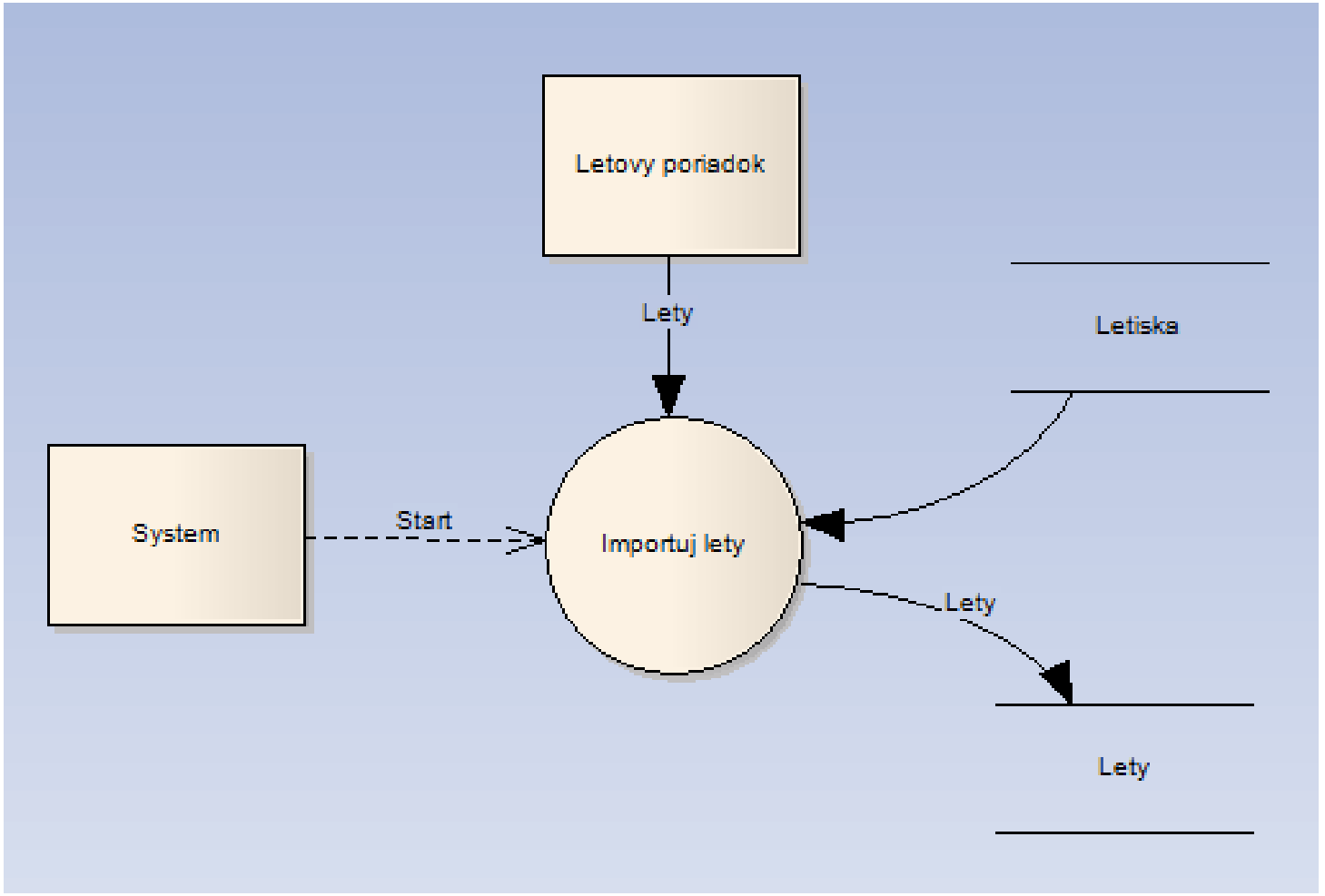
Alternatíva

- Použiť procesný model (DFD) ako doplnok modelu prípadov užitia
- Tie prípady užitia, ktoré neobsahujú interakciu s užívateľom (automatické procesy) nepopisovať ako prípady užitia
- Namiesto toho použiť nejakú metódu vhodnú na špecifikáciu procesov

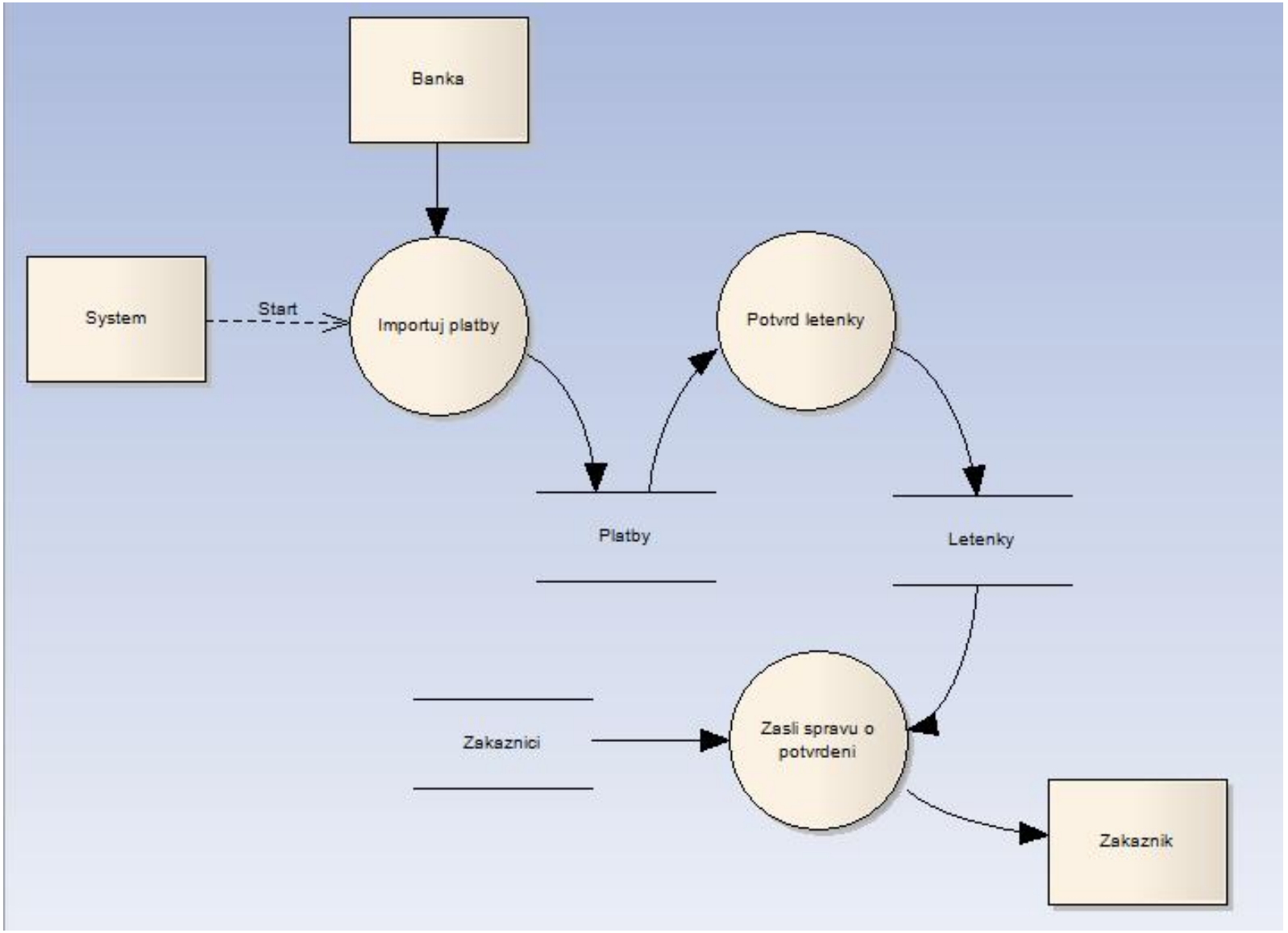
Príklad: Model prípadov užitia



Príklad: DFD 1.



Príklad: DFD 2.



Špecifikácia procesov

- Každý proces je jednoznačne špecifikovaný pomocou
  - Vstupov
  - Výstupov
  - Predpisu ako sa vstupy transformujú na výstupy

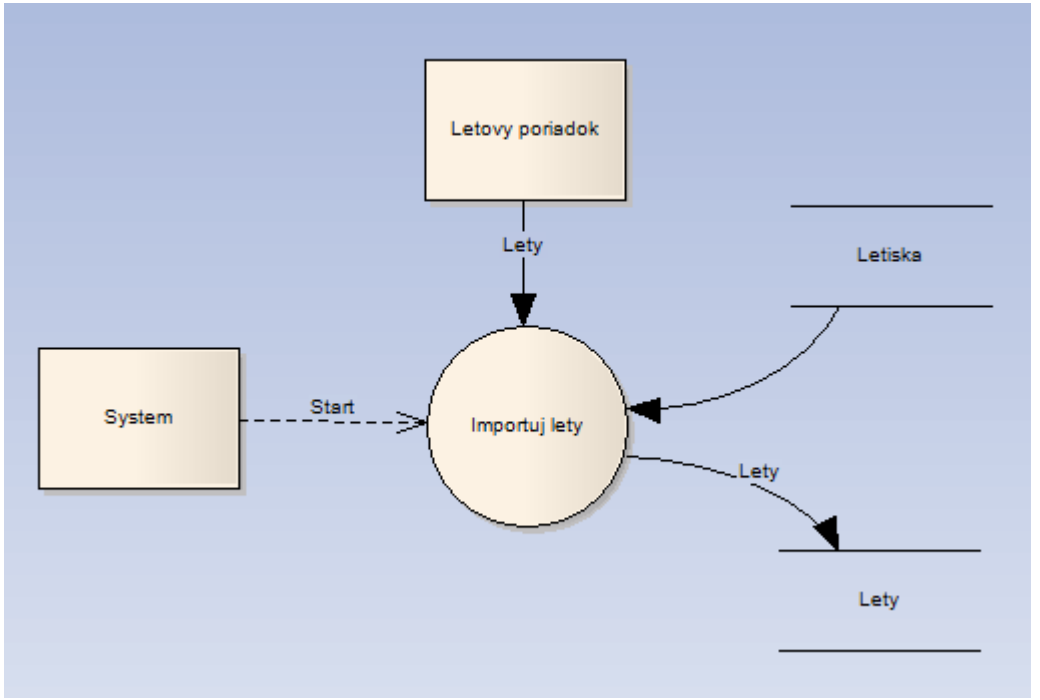
Špecifikácia vstupov a výstupov

- Vstupy a výstupy sú dátové toky. Ich špecifikácie sú preto totožné so špecifikáciou dátových tokov
- Pozor: špecifikácie sú úplné až keď sú definované všetky atribúty spolu s ich typmi

Vstupy a výstupy: príklad

Dátový tok Lety:

```
Let_id: int
Letisko_z: string
Letisko_do: string
Odlet: Datetime
Prilet: Datetime
```



Transformácia vstupu na výstup

- Slovný popis
- Popis pomocou pseudokódu
- Popis pomocou preconditions a postconditions

Slovný popis

„Romantizovaná próza“

Zo vstupu sa prečítajú údaje let_id, letisko_z, letisko_do, odlet a prílet. Vykoná sa kontrola údajov. Pri nej sa skontroluje či letiská, ktoré sa naimportovali, existujú v databáze rezervačného systému. Ak neexistujú, imort sa preruší. Ak existujú, tak sa lety vložia do databázy letov

Nevýhody romantizovanej prózy

- Pri zložitých procesoch sa ťažko vysvetľuje, čo proces má robiť. Špecifikácia je potom nepochopiteľná
- Vysoké nebezpečenstvo nejednoznačnosti predpisu
- Dôsledkom je rôzne pochopenie predpisu rôznymi programátormi

Popis pomocou pseudokódu:

- Pseudokód sa podobá na kód v nejakom programovacom jazyku
- Neviaže sa na predpisy nejakého konkrétneho jazyka
- Dizajnér si sám zvolí aké výrazy a konštrukcie použije
- Dôležitým aspektom je čitateľnosť
- Rizikom je to, že predpis skĺzne do podoby romantizovanej prózy

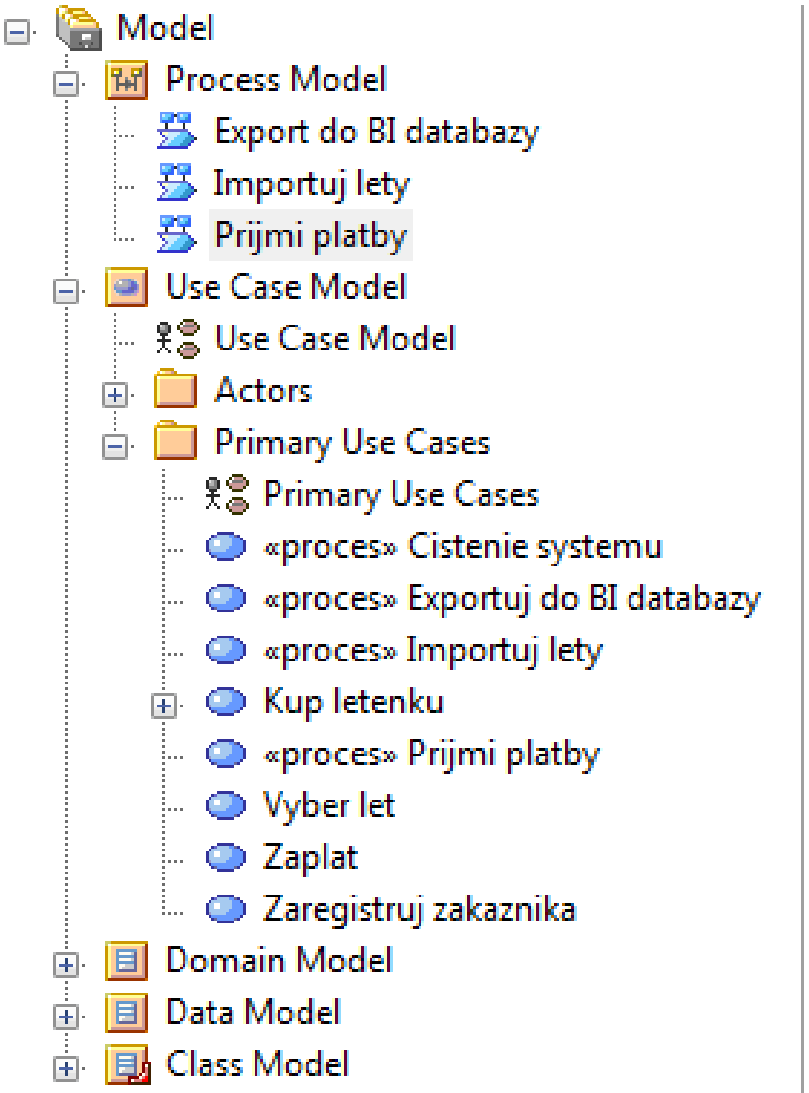
Pseudokód: príklad

```
Prečítaj zo vstupu {let_id,
letisko_z, letisko_do, odlet a
prílet}

Ak NajdiVDatabazeLetov(letisko_z) =
False potom Koniec.

Ak NajdiVDatabazeLetov(letisko_do) =
False potom Koniec

Uloz{let_id, letisko_z, letisko_do,
odlet a prílet}
```

Špecifikácia pomocou preconditions a postconditions

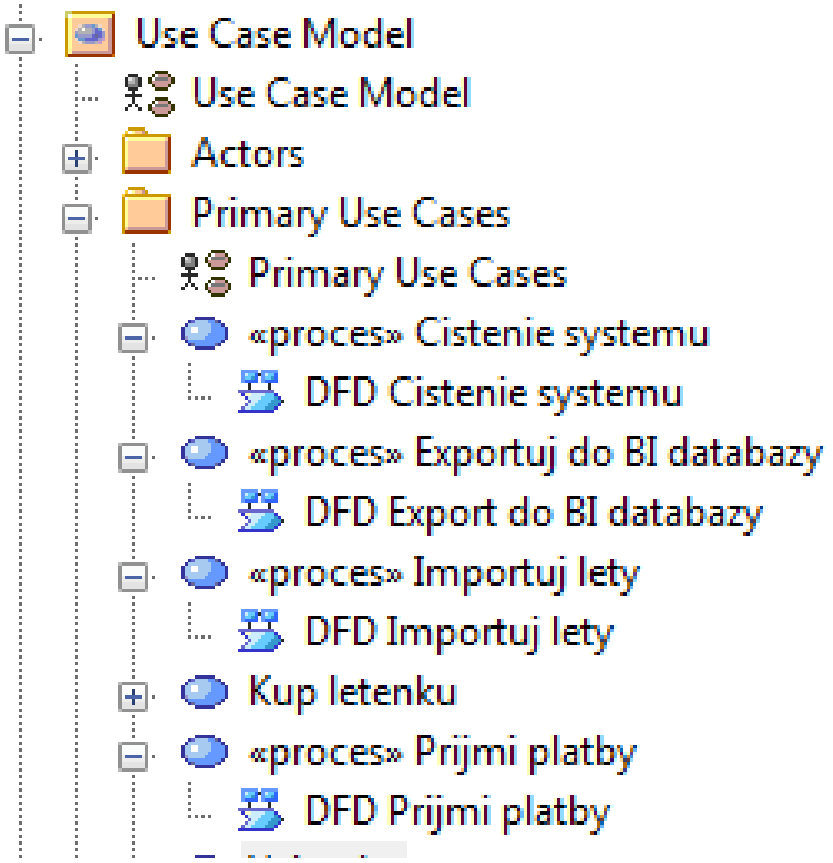
- Preconditions: Množina podmienok, ktoré musia byť splnené, aby proces mohol začať
  - Dôsledok: Ak preconditions nie sú splnené, proces sa nezačne (vyhlási chybu)
  - Proces musí na začiatku skontrolovať, či sú preconditions splnené
- Postconditions: Množina podmienok, ktorých splnenie musí proces zaručiť pri svojom ukončení

Štruktúra projektu



Iná možnosť

- Data Flow Diagram umiestniť k prípadu užitia, ku ktorému patria



Výhody takejto štruktúry

- Poskytuje dva (užitočné) pohľady na požiadavky
- Tým, že sú niektoré prípady užitia modelované zároveň ako procesy, môžeme
  - Vytvoriť realizácie prípadov užitia, ktoré zapadajú do štruktúry modelu softwaru podľa RUP
  - Popísať procesy pomocou niektorej metódy pre špecifikáciu procesov (a memusíme na to zneužívať prípady užitia)

Thirty years of relational, relational forever

C. J. Date

Základy relačného dátového modelu

Tri aspekty relačného modelu

- Štruktúra: dáta sú užívateľom vnímané ako tabuľky a nič iné než tabuľky
- Integrita: Tabuľky spĺňajú isté integritné pravidlá
- Manipulácia: Operátory, ktoré má užívateľ k dispozícii, aby mohol s tabuľami narábať

Príklad relačnej databázy

DEPT

EMP

Príklady operácií:

Reštrikcia:

DEPT where BUDGET>8M

Projekcia:

DEPT over DEPT#, BUDGET

Join:

DEPT and EMP over DEPT#

Uzavretosť (closure) relačného dátového modelu

- Výsledkom každej operácie na relačnom modeli je tabuľka (relácia)
- Dôsledok: Operácie je možné ľubovoľne spájať (pipelines)
- Impedance mismatch: RDBMS vracajú vždy tabuľky, kým odberatelia očakácajú jednotlivé záznamy a atomické hodnoty
- Je to argument proti používaniu RDBMS?

Relačný model je abstrakcia

- Dáta v relačnej databáze sa užívateľovi javia ako tabuľky
- To neznamená, že sú ako tabuľky aj implementované!

Coddov informačný princíp

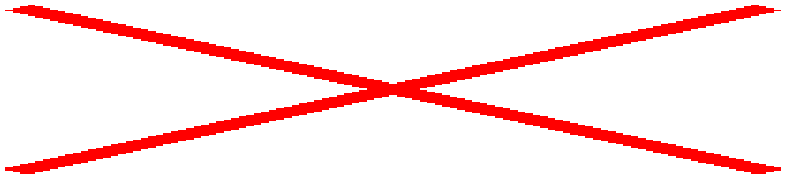
- The entire information content of the database is represented in one and only one way – namely, as explicit values in column positions in rows in tables
- Dôsledok: neexistujú smerníky (pointers), ktoré spájajú tabuľky databázy
- Spojenia sa realizujú pomocou kľúčov: primárne a cudzie kľúče

N-tice (tuples)

- Nech *Ti(i=1,2,...n)* je množina nie nutne rozdielnych typov
- Hodnota n-tice (povedzme t) nad týmito typmi je množina usporiadaných trojíc <*Ai, Ti, Vi*>, kde Ai je meno atribútu, *Ti* je typ atribútu a vi je hodnota typu *Ti*
- N: stupeň (degree, arity) n-tice t
- Uporiadaná trojica <*Ai, Ti, Vi*> sa nazýva komponentom n-tice t

N-tice (tuples) II.

- Usporiadaná dvojica <*Ai, Ti*> je atribútom n-tice t. Atribút je jednoznačne identifikovaný svojím menom *Ai*
- Hodnota *Vi* je hodnotou atribútu *Vi*
- Úplná množina atribútov sa nazýva hlavičkou n-tice
- Hlavička definuje typ n-tice



Vlastnosti n-tíc

- Každá n-tica obsahuje práve jednu hodnotu pre každý zo svojich atribútov
- Množina atribútov je neusporiadaná
- Každá podmnožina n-tice je tiež n-tica

Operácie na n-ticiach

Príklad:

```
Var Adresa1, Adresa2: Tuple
   {Ulica: char,
    Mesto: char,
    PSC: char
   }
```

Priradenie:

```
Adresa1:={Srobarova, Poprad, 05801}
Adresa2:=Adresa1
```

Operácie na n-ticiach II.

Porovnanie:

`A1=A2`

`A1={Ligusterlaan, Winterswijk, 7101WS}`

Všetky operácie relačnej algebry

`-Union, Intersect, Difference, Product, Restrict, Project, Join, Divide`

Relácie

- Relačná hodnota
- Relačný typ
- Relačná premenná

Relačná hodnota a relačný typ

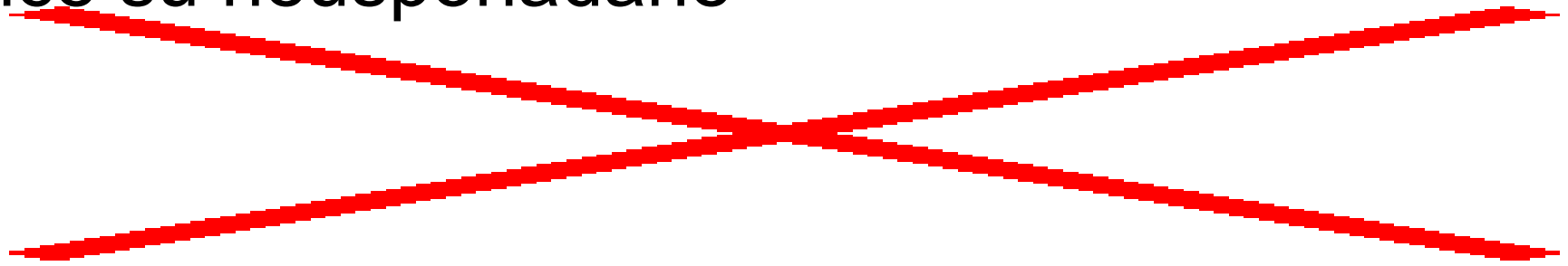
- Relačná hodnota pozostáva z hlavičky (heading) a tela (body)
- Hlavička: rovnaká definícia ako v prípade n-tíc
- Telo: množina n-tíc
- Relačný typ je definovaný hlavičkou:

```
RELATION{DEPT# char, DNAME char, BUDGET money}
```

- Kardinalita relácie: počet n-tíc

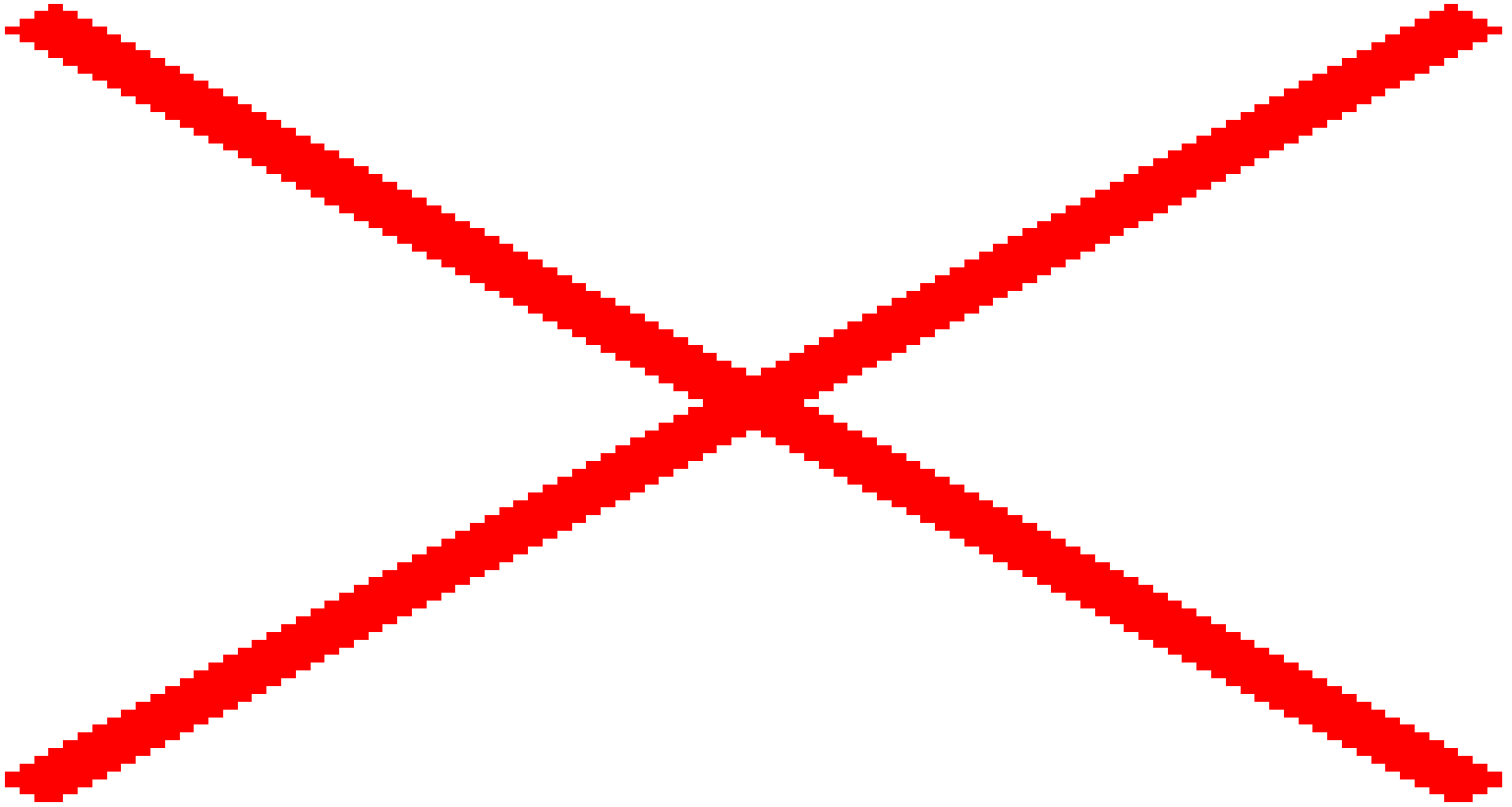
Vlastnosti relácií

- Relácie sú normalizované: každá n-tica obsahuje práve jednu hodnotu pre každý zo svojich atribútov ➔Relácia je v prvej normálnej forme. Hodnota atribútu pritom môže tiež byť typu RELÁCIA
- Atribúty sú neusporiadané
- N-tice sú neusporiadané



- N-tice sú v relácii jedinečné, neopakujú sa

Príklad relácie s atribútom typu RELÁCIA



Operácie na reláciách

- Priradenie a porovnanie
- Porovnanie:

`<relation exp> <relation comp op> <relation exp>`

- `<relation exp>` ľubovoľný výraz typu relácia
- `<relation comp op>` je jedno z:

$=$ rovná sa

$\supseteq$ nadmnožina

$\neq$ nerovná sa

$\subset$ vlastná podmnožina

$\supset$ vlastná nadmnožina $\subseteq$ podmnožina

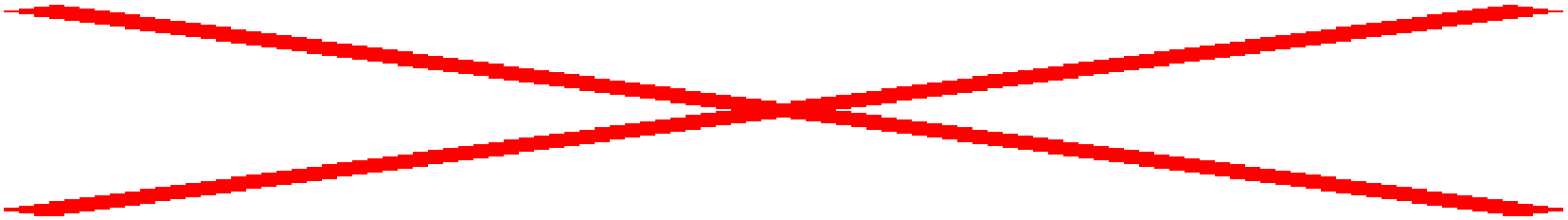
Relačná premenná: relvar

- Premenná, ktorej hodnoty sú relácie
- SQL: Create table: je to zároveň deklarácia typu ako aj premennej
- Closed World Assumption:
  - If an otherwise valid tuple – that is, one that conforms to the relvar heading – does not appear in the body of the relvar, then we assume that the corresponding proposition is false

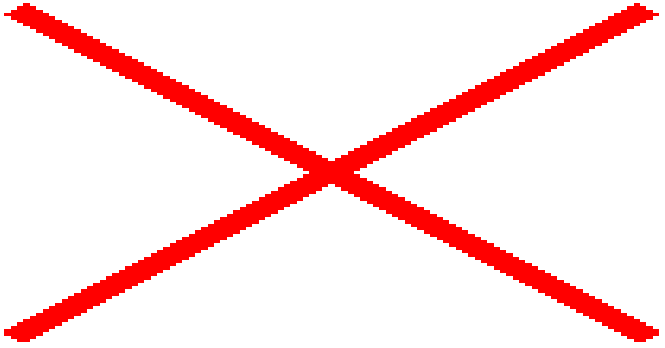
Relačná algebra

- Definuje operácie na relačnom modeli:
  - Zjednotenie (union)
  - Prienik (intersect)
  - Rozdiel (difference)
  - (Kartézsky) súčin (product)
  - Reštrikcia (restrict)
  - Projekcia (project)
  - Spojenie (join)
  - Podiel (divide)
- Uzavretosť: výsledkom každej operácie je relácia

Spojenie (union)



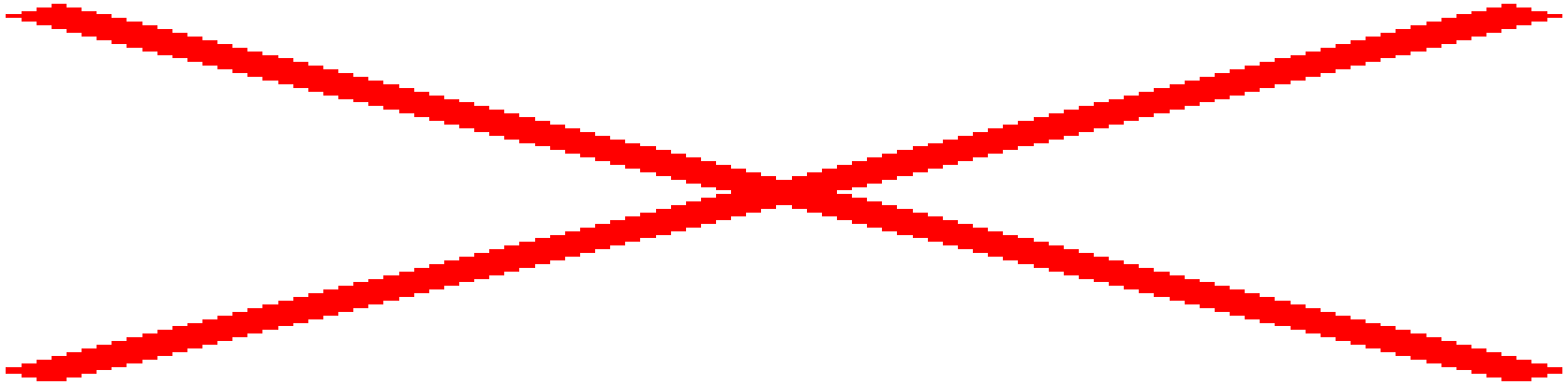
A a B musia byť rovnakého typu
A UNION B obsahuje všetky n-tice, ktoré sa nachádzajú v A, B alebo obidvoch reláciách



Projekcia

Nech relácia a má atribúty X,Y,...,Z a prípadne ďalšie
Potom projekcia a {X,Y,...Z} je relácia pozostávajúca z:

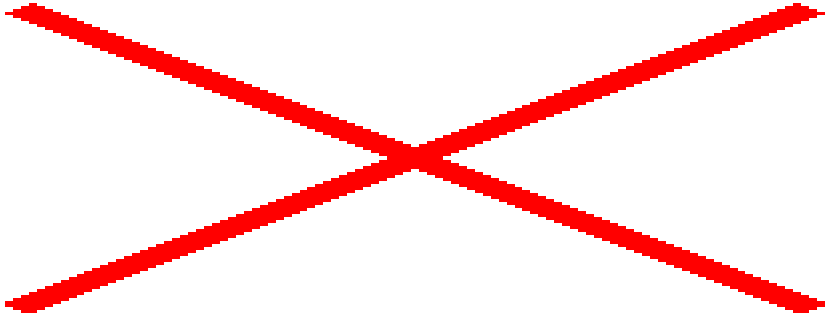
- hlavičky, ktorá je odvodená z hlavičky a odstránením atribútov, ktoré nie sú v monžine X, Y, ... Z
- n-tíc z relácie a pozostávajúcich z atribútov X,Y,...Z



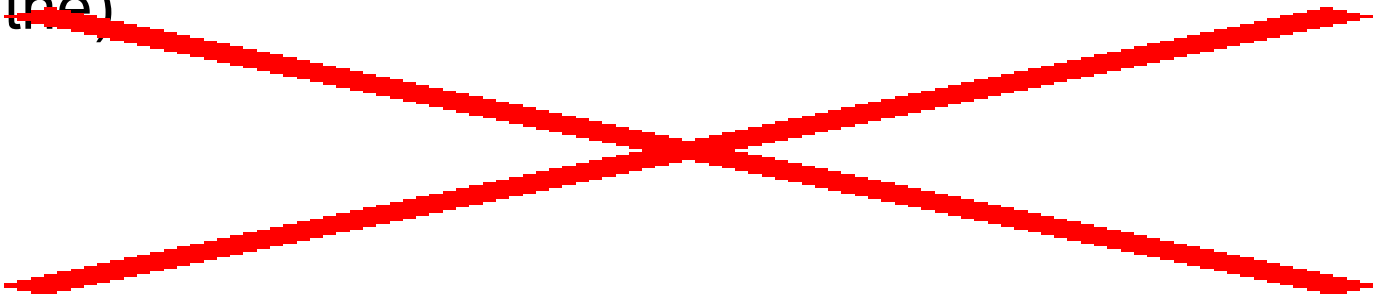
Kandidátne kľúče

- N-tice sú vždy jedinečné, relačné hodnoty nikdy neobsahujú dve rovnaké n-tice (z definície)
- Často aj podmnožiny atribútov nejakej relačnej premennej spĺňajú vlastnosť jedinečnosti
- Nech K je množina atribútov rel. premennej R. Potom K je kandidátnym kľúčom vtedy a len vtedy keď má nasledujúce dve vlastnosti:
  - Jedinečnosť: žiadna hodnota R nemá dve n-tice s rovnakou hodnotou K
  - Neredukovateľnosť: Žiadna vlastná podmnožina K nemá vlastnosť jedinečnosti

Kandidátne kľúče II.



Kandidátne kľúče môžu byť jednoduché alebo zložené (kompozitné)



–Kandidátne kľúče predstavujú základný (a jediný) mechanizmus pre adresovanie n-tíc

Kandidátne kľúče -cvičenie.

- Definujte všetky kandidátne kľúče nasledujúcej relácie

```
Var Svatba RELATION {

        Zenich char,

        Nevesta char,

        Dna datum
                                                }
```

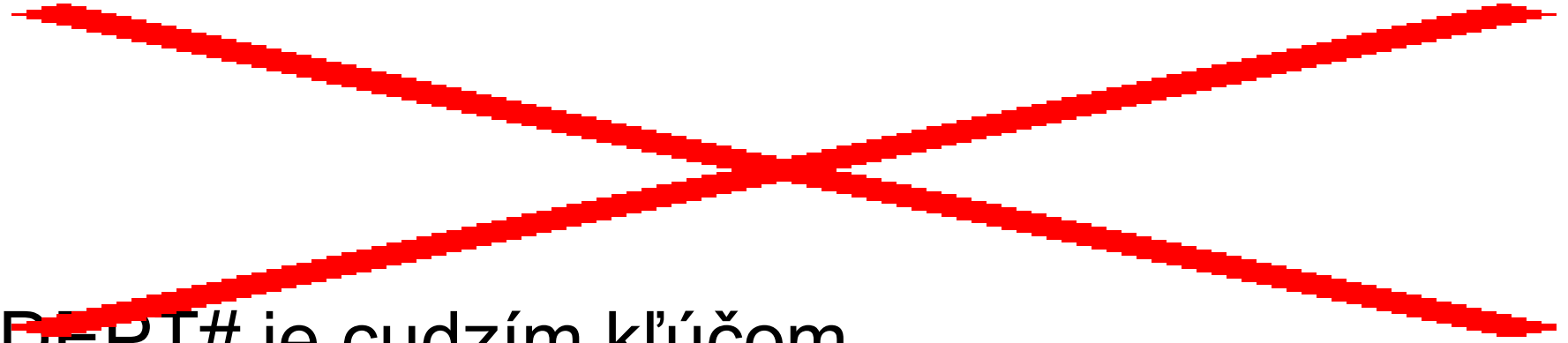
Primárny a alternatívny kľúč

- Jeden z kandidátnych kľúčov sa vyberie a označí sa ako primárny kľúč. Ostatné su alternatívne.
- Ktorý kľúč vybrať za primárny:
  - Codd: The normal basis for making the choice is simplicity, but this aspect is outside the scope of the relational model

Cudzie kľúče

- Množina atribútov rel. premennej R2 ktorých hodnoty musia zodpovedať hodnotám kandidátneho kľúča nejakej inej rel. prem. R1

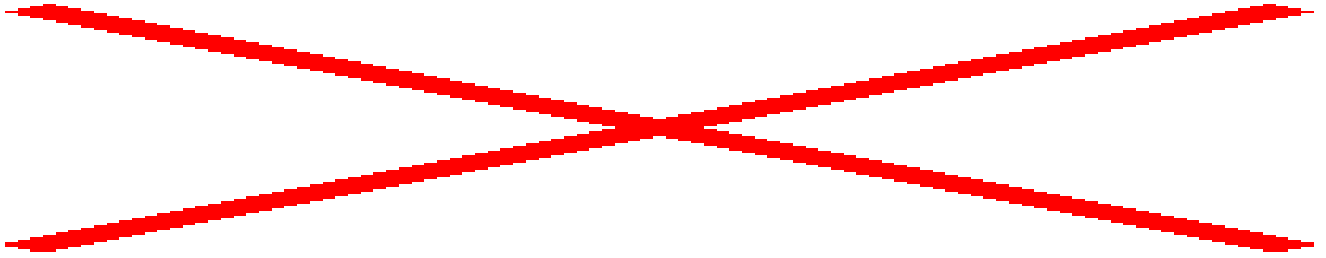
DEPT EMP



- EMP.DEPT# je cudzím kľúčom
- EMP.DEPT# môže nadobudnúť istú hodnotu, iba ak sa táto hodnota vyskytuje v hodnote relačnej premennej DEPT

Cudzie kľúče II.

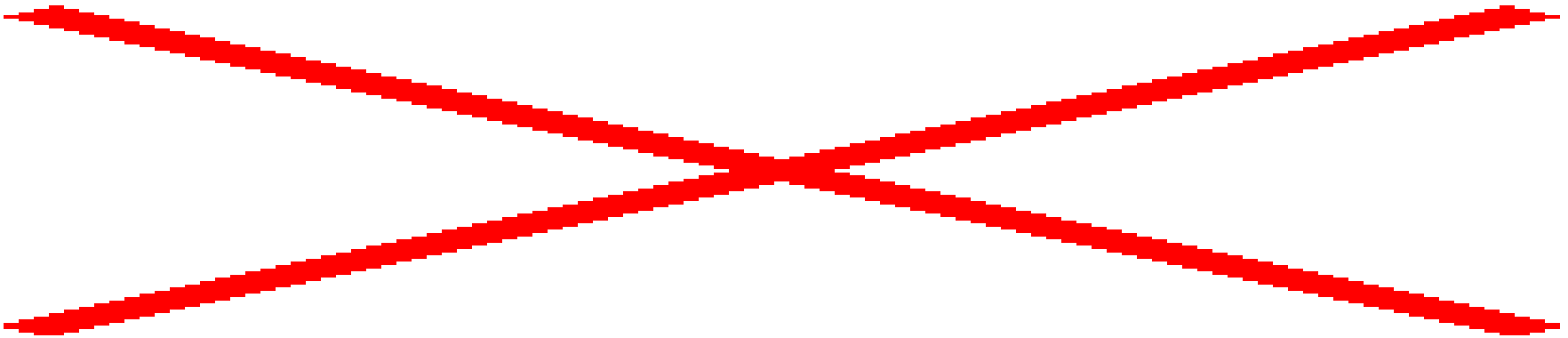
- Hodnota cudzieho kľúča predstavuje referenciu k n-tici, ktorá má kandidátny kľúč rovný tejto hodnote
- Cudzí kľúč môže byť jednoduchý alebo zložený
- R1 a R2 nemusia byť nutne rozdielne



- Rovnaké hodnoty cudzieho a kandidátneho kľúča definujú vzťahy (referencie) medzi reláciami

Referenčná integrita

- Databáza nesmie obsahovať žiadne hodnoty cudzích kľúčov, ku ktorým neexistuje zodpovedajúca hodnota kandidátneho kľúča



- Referenčné akcie: Cascading Delete a Update

Relačný model a jazyk SQL

- Jazyk SQL nie je (úplnou, dobrou) implementáciou relačného modelu
- Chris Date: SQL is for the relational model what American government is for the democracy
- Problémy:
  - SQL Table = relačný typ & relačná premenná
  - Nie je možné definovať užívateľské typy
  - Nepodporuje atribúty typu Relation
  - Nepodporuje dedičnosť
  - Výrazy jazyka nie je možné ľubovoľne zreťazovať

Jazyky typu D

- Chris Date a Hugh Darwen vytvorili sériu všeobecných pravidiel, ktoré by mali spĺňať jazyky pracujúce s relačnými databázami
- Jazyky, ktoré tieto pravidlá spĺňajú sa označujú ako jazyky typu D
- Date a Darwen vytvorili príklad takéhoto jazyka, Tutorial D (Tutorial: pre výukové účely)
- Existuje už aj niekoľko prakticky použiteľných a jedna komerčná implementácia

Implementácie jazykov typu D

- Rel: implementácia jazyka Tutorial D, vhodná na učenie sa a experimentovanie
- Dataphor: Komerčná implementácia jazyka D, nazvaná D4. Dataphor je vývojové prostredie okolo tohto jazyka
- Ingres: Open source databáza Ingres takisto zabudovala podporu jazyka D
- Počet implementácií rastie, použitie v praxi je minimálne

Výhody jazykov typu D

- Vysoká kvalita základných princípov a modelu, z ktorých vychádzajú (sound theoretical foundation)
- Jednoduchosť konceptu a nesmierna výrazová sila jazyka
- Jazyky typu D podporujú všetky požiadavky, ktoré bývajú uvádzané ako dôvod pre tvorbu objektových databázových systémov (zložité dátové štruktúry, dedičnosť...)
- Dobré možnosti integrovať jazyk typu D s inými vývojovými prostrediami

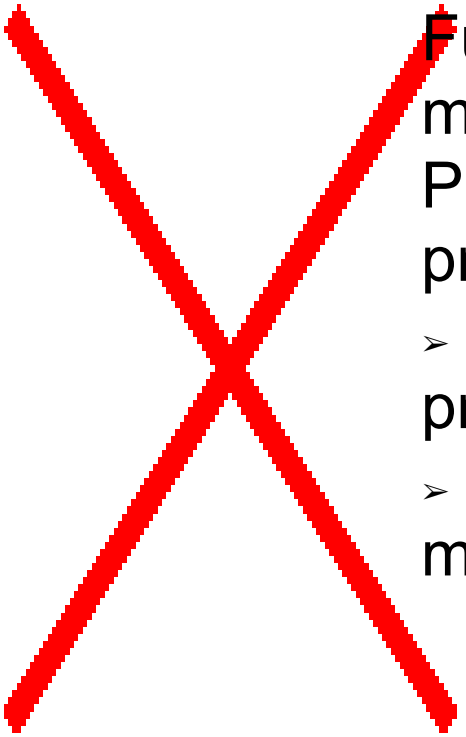
...not quite fundamental, but very nearly so

Hugh Darwen

Funkčné závislosti a normalizácia

Funkčné závislosti

- FD (functional dependency) predstavuje vzťah medzi dvoma množinami atribútov danej relačnej premennej s kardinalitou n:1



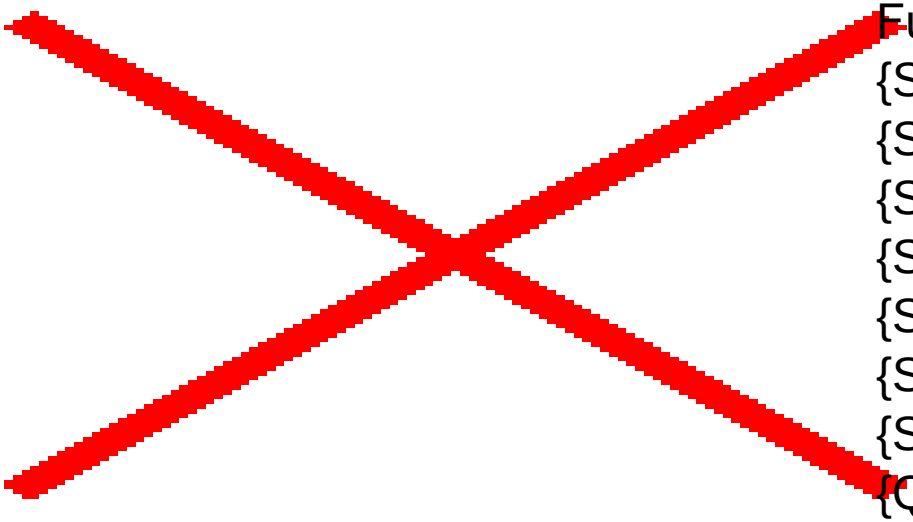
Funkčná závislosť z množiny atribútov {S#, P#} do množiny atribútov {QTY} :
Pre každú legálnu hodnotu tej3to relačnej premennej platí:

- Ku každej hodnote kombinácie {S#, P#} existuje práve jedna hodnota {QTY}
- Ľubovoľnému počtu hodnôt kombinácie {S#, P#} môže zodpovedať tá istá hodnota {QTY}

Formálna definícia: prvá verzia

- Nech *r* je hodnotou typu RELÁCIA a X a Y ľubovoľné podmnožiny atribútov *r*. Hovoríme, že Y je funkčne závislé na X vtedy a len vtedy keď každej hodnote X v *r* zodpovedá práve jedna hodnota Y.
  - Inými slovami, ak dve n-tice majú rovnakú hodnotu X, majú aj rovnakú hodnoty Y
- Formálny zápis: $X \rightarrow Y$, čítaj X funkčne determinuje Y. X je determinant, Y je dependent

Príklad



Funkčné závislosti:

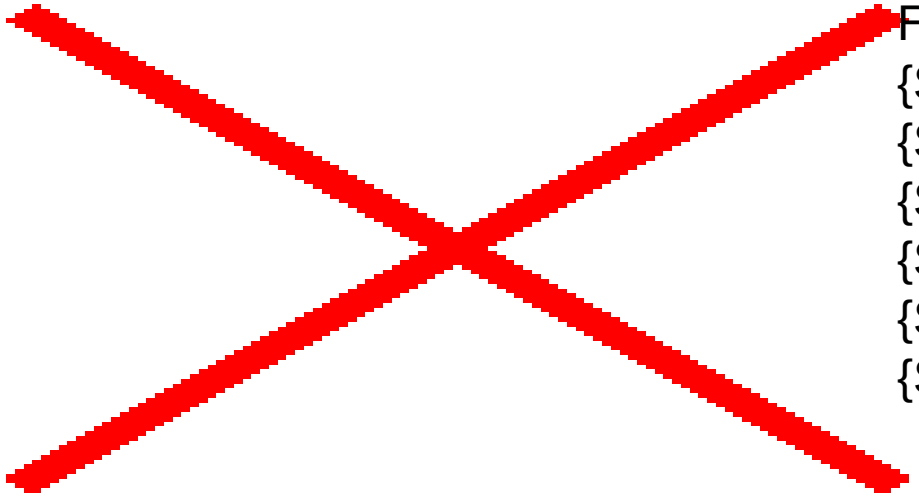
{S#} $\rightarrow$ {CITY}
{S#, P#} $\rightarrow$ {QTY}
{S#, P#} $\rightarrow$ {CITY}
{S#, P#} $\rightarrow$ {CITY,QTY}
{S#, P#} $\rightarrow$ {S#}
{S#, P#} $\rightarrow$ {S#,P#,CITY,QTY}
{S#} $\rightarrow$ {QTY}
{QTY} $\rightarrow$ {S#}

- Tento pohľad na funkčné závislosti sa obmedzuje na jednu hodnotu relačného typu
- Zaujímavejšie sú funkčné závislosti, ktoré platia pre všetky možné hodnoty nejakej relačnej premennej (t.j. pre nejaký relačný typ)

Formálna definícia: druhá verzia

- Nech R je relačná premenná a X a Y ľubovoľné podmnožiny atribútov R. Hovoríme, že Y je funkčne závislé na X vtedy a len vtedy keď pre každú prípustnú hodnotu premennej R platí, že každej hodnote X zodpovedá práve jedna hodnota Y.
- Od teraz budeme používať termín funkčná závislosť v tomto silnejšom význame

Príklad



Funkčné závislosti:

{S#} $\rightarrow$ {CITY}

{S#, P#} $\rightarrow$ {QTY}

{S#, P#} $\rightarrow$ {CITY}

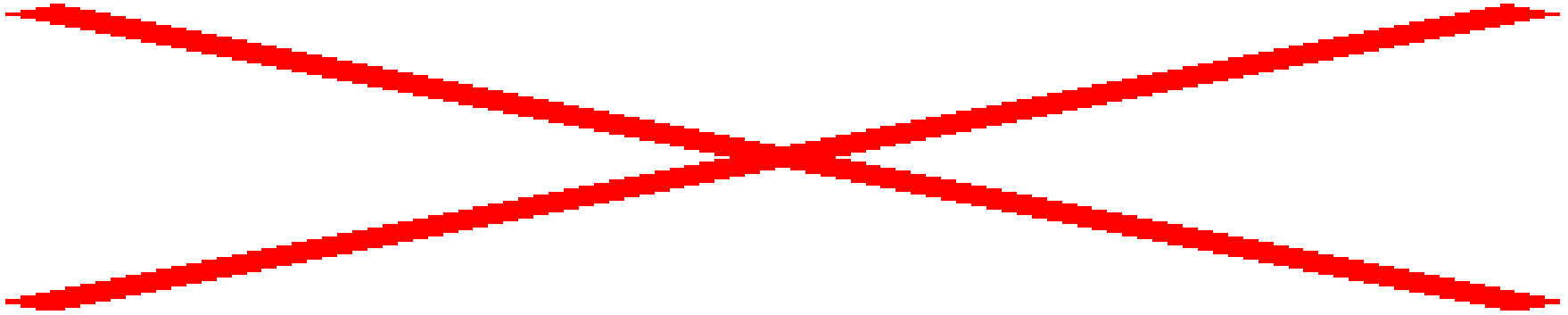
{S#, P#} $\rightarrow$ {CITY,QTY}

{S#, P#} $\rightarrow$ {S#}

{S#, P#} $\rightarrow$ {S#,P#,CITY,QTY}

Funkčné závislosti a kľúče

- Ak X je kandidátnym kľúčom R, tak všetky atribúty R musia nutne byť funkčne závislé na X
- Ak v nejakej relačnej premennej R platí $A \rightarrow B$ a A nie je kandidátnym kľúčom, tak R nutne obsahuje redundanciu



Triviálne a netriviálne závislosti

Závislosť je triviálna, ak za žiadnych okolností nie je možné porušiť ju. Napríklad:

$\{S\#, P\#\} \rightarrow S\#$

FD je triviálna, ak pravá strana je podmnožinou ľavej strany

V praxi nás samozrejme viac budú zaujímať netriviálne závislosti

Uzáver množiny FD

Niektoré funkčné závislosti implikujú iné závislosti:

$\{S\#,P\#\} \rightarrow \{CITY,QTY\}$ implikuje

$\{S\#,P\#\} \rightarrow \{CITY\}$ a

$\{S\#,P\#\} \rightarrow \{QTY\}$

Množina všetkých FD odvoditeľných z nejakej množiny funkčných závislosti S sa nazýva uzáverom množiny S a označuje sa S^+

Armstrongove axiómy

Pravidlá pre odvodzovanie nových závislostí:

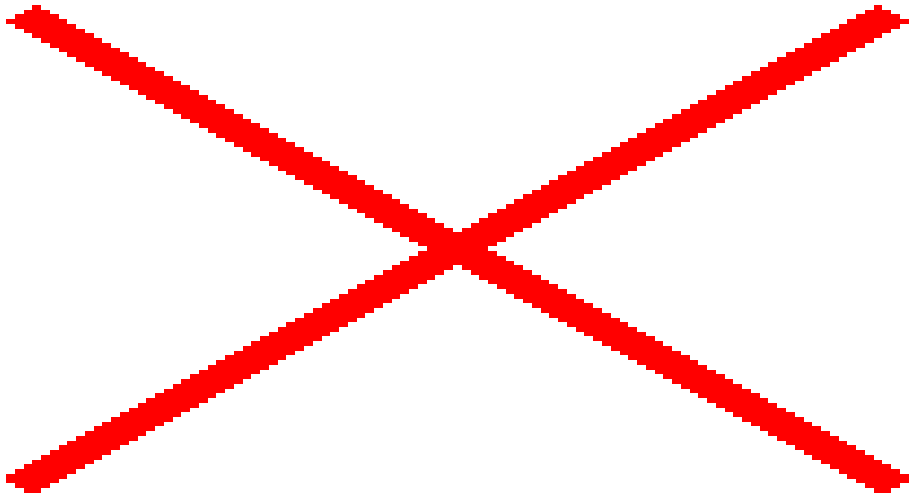
1. Reflexivita: Ak $B \subset A$, potom $A \rightarrow B$
2. Augmentácia: Ak $A \rightarrow B$, potom $AC \rightarrow BC$
3. Tranzitivita: Ak $A \rightarrow B$ a $B \rightarrow C$, potom $A \rightarrow C$

Táto množina pravidiel je úplná a postačujúca, Armstrong ich však definoval viac

Armstrongove axiómy II.

4. Self-determination: $A \rightarrow A$
5. Dekompozícia: Ak $A \rightarrow BC$, potom $A \rightarrow B$ a $A \rightarrow C$
6. Zjednotenie: Ak $A \rightarrow B$ a $A \rightarrow C$, potom $A \rightarrow BC$
7. Kompozícia: Ak $A \rightarrow B$ a $C \rightarrow D$, potom $AC \rightarrow BD$

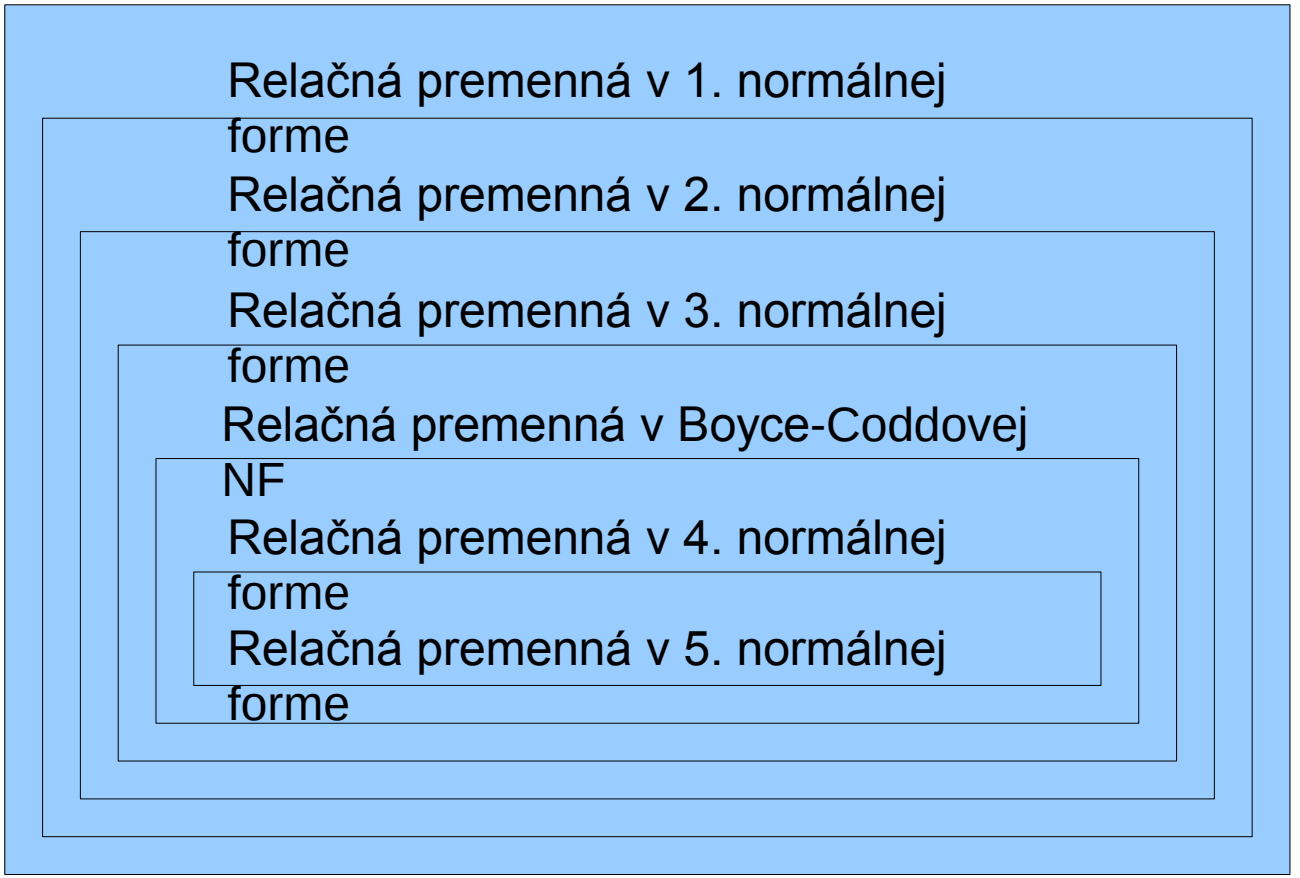
Normalizácia: úvod



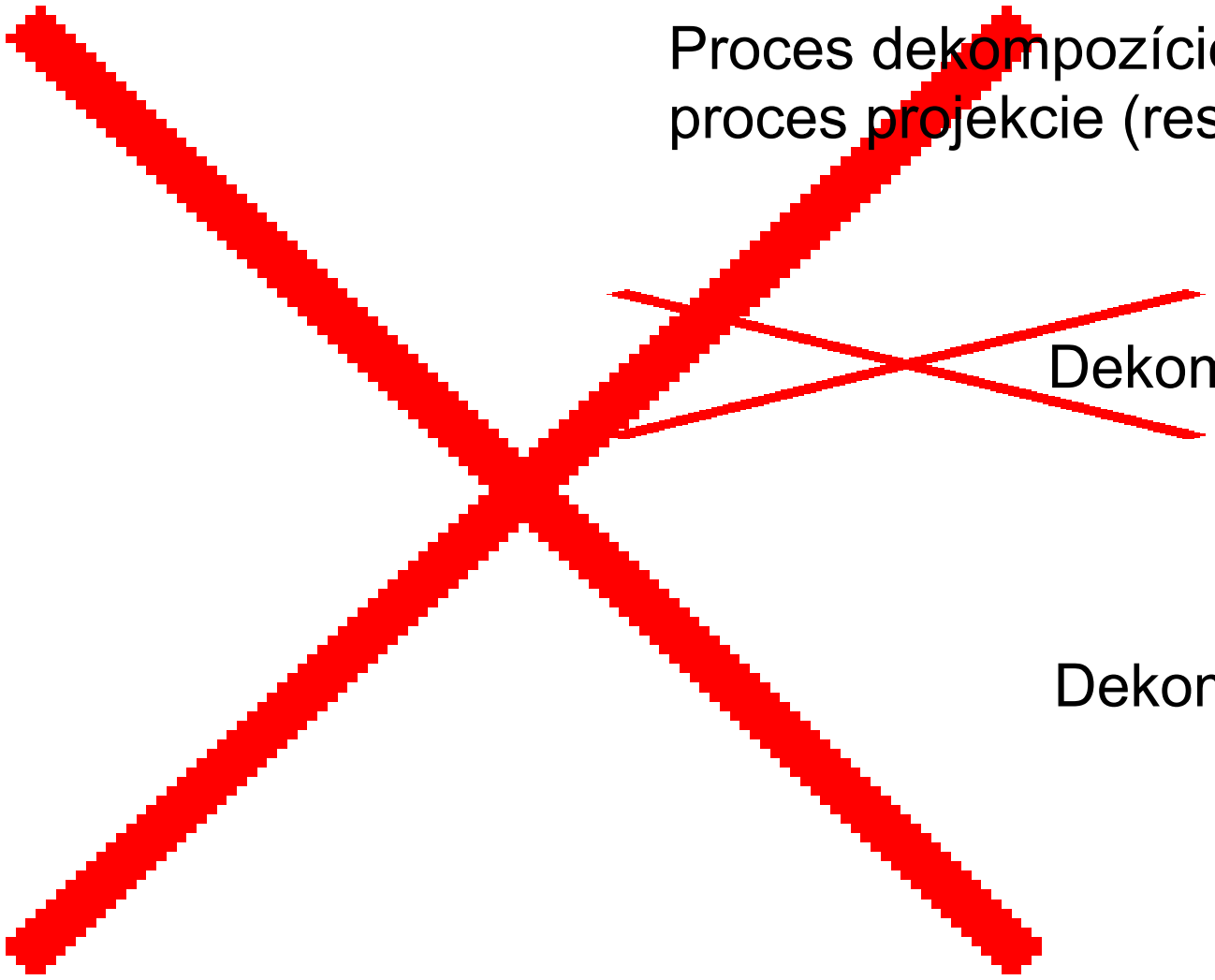
- Čo nie je v poriadku s touto reláciou?
- Problém je intuitívny, záležitosť “selského” rozumu (common sense)

- Ako to však formalizovať?
- Aké problémy vyplývajú z tohoto dizajnu?

Prehľad normálnych foriem



Bezstratová dekompozícia



Proces dekompozície=
proces projekcie (resp. niekoľkých projekcií)

Dekompozícia je bezstratová

Dekompozícia je stratová

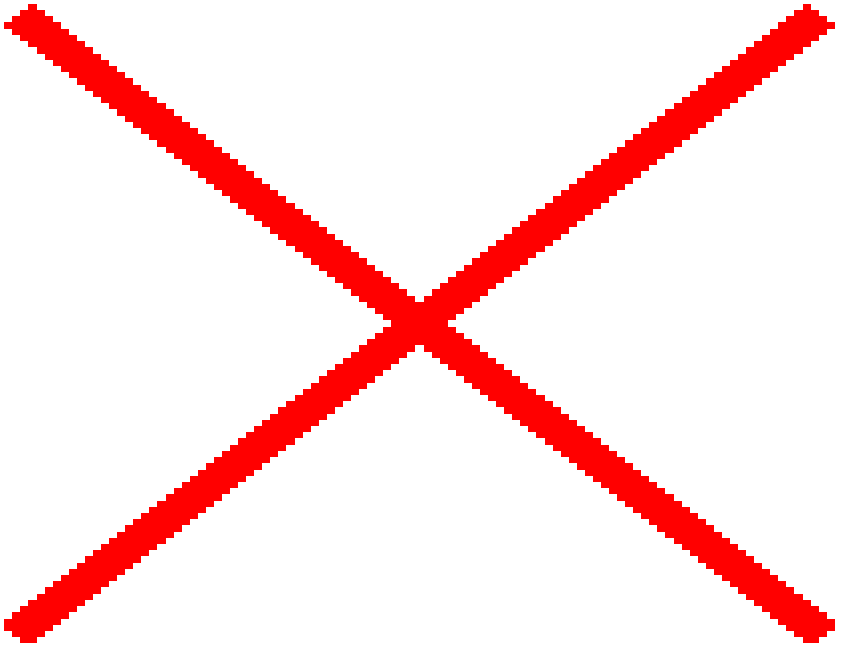
Kedy je dekompozícia bezstratová? Heathov teorém

- Nech R je relačná premenná s troma množinami atribútov A, B a C: R{A,B,C}. Potom platí, že ak A→B, tak R je rovné spojeniu projekcií {A,B} a {A,C} cez množinu A

Platí funkčná závislosť
Mesto→PSČ

Neplatí však FD
Mesto→Rodne cislo, Meno

Dekompozícia na
S1:{Mesto,PSC} a
S2:{Mesto, Rodne cislo, Meno}
je bezstratová

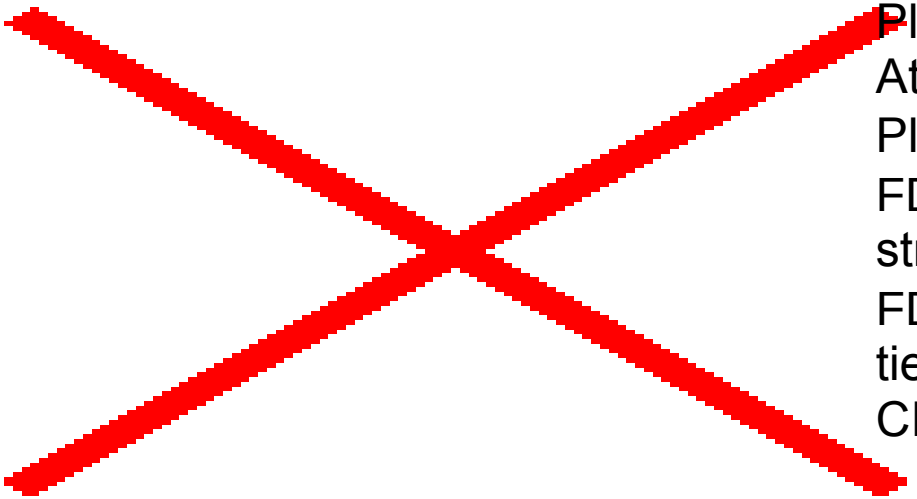


Bezstratová dekompozícia: zhrnutie

- Dekompozícia relácie R na projekcie R1, R2,..., Rn je bezstratová, ak R sa rovná spojeniu R1, R2,...,Rn
- V praxi budeme zrejme žiadať, aby všetky projekcie R1, R2,...,Rn boli potrebné na spätné zrekonštruovanie R
- Napríklad, {S#}, {S#, STATUS}, {S#, CITY} je síce bezstratová dekompozícia, {S#} však nie je na zrekonštruovanie S potrebné

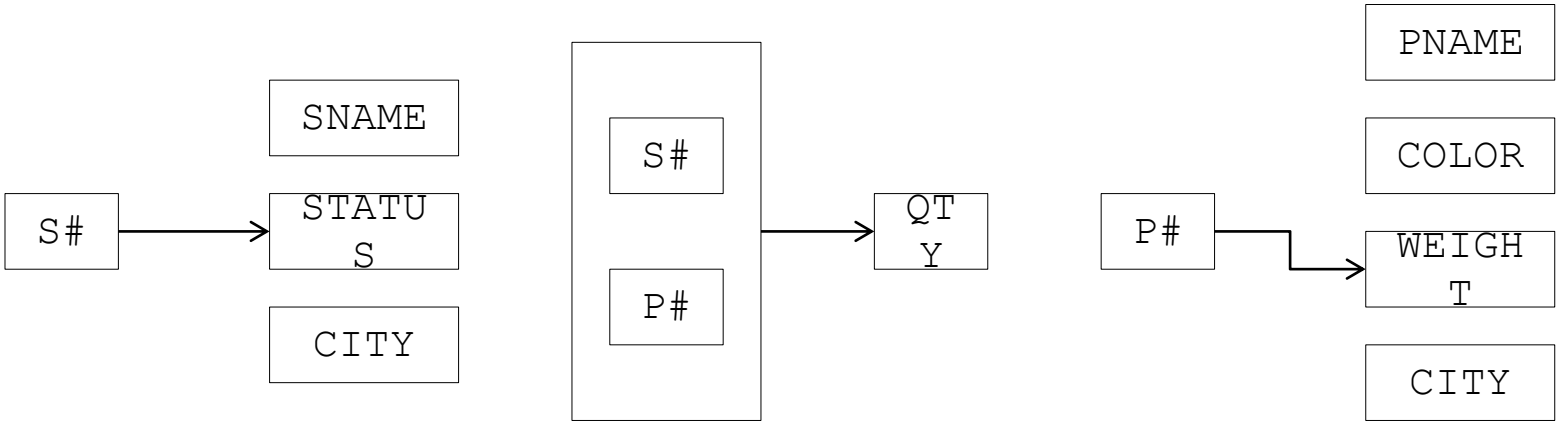
Neredukovateľnosť FD

- FD je (zľava) neredukovateľná ak jej ľavá strana nie je “príliš veľká”



Platí FD {S#, P#} → CITY
Atribút P# je v tejto FD nadbytočný
Platí totiž tiež FD {S#} → CITY
FD {S#, P#} → CITY je zľava (=jej ľavá strana) redukovateľná
FD {S#} → CITY je neredukovateľná, alebo tiež:
CITY je neredukovateľne závislé na S#

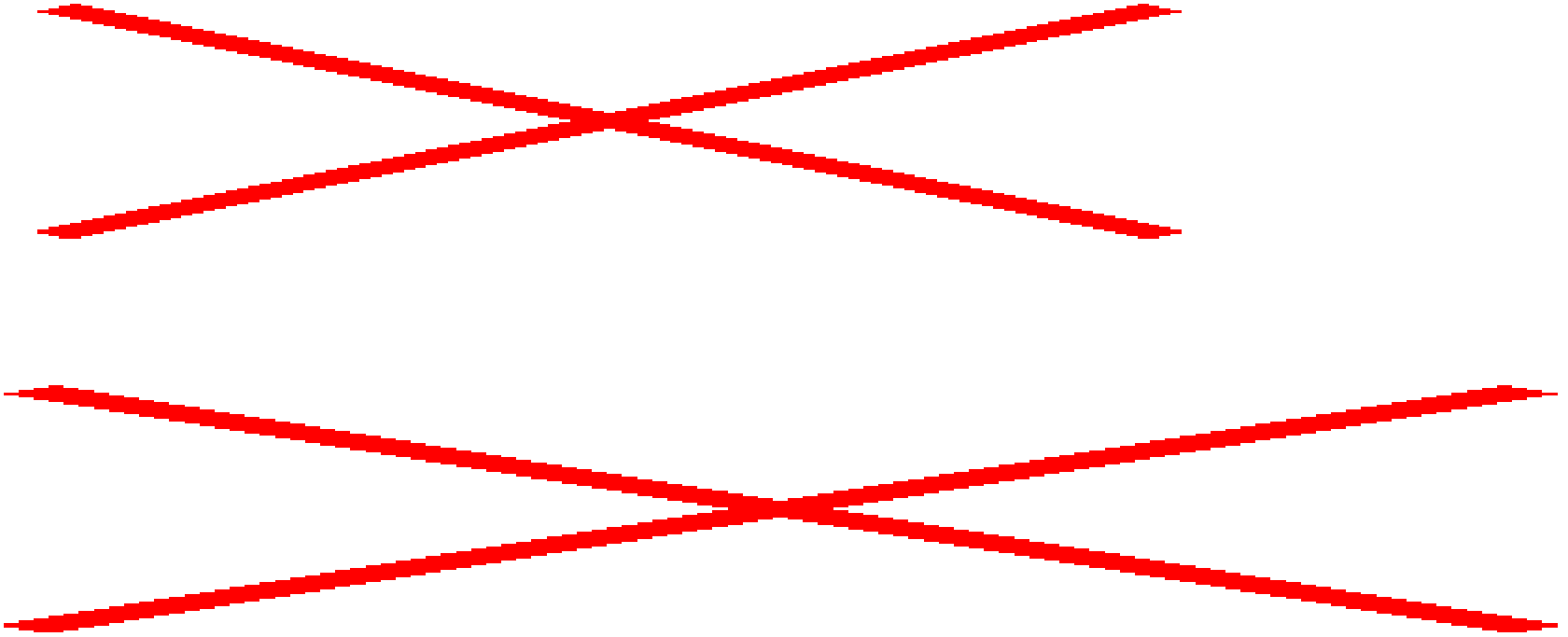
FD Diagramy



- Na ľavej strane diagramu sú v týchto príkladoch kandidátne kľúče: tie existujú vždy
- Šípky z iných atribútov než kandidátnych kľúčov znamenajú problémy: nenormalizované relácie

FD: sémantická záležitosť

- Týkajú sa významu dát



Normalizácia: 1., 2. a 3. normálna forma

- Kam smerujeme?
- Neformálna definícia 3NF:
  - Relačná premenná je v 3NF vtedy a len vtedy, keď nekľúčové atribúty sú zároveň:
    - Navzájom na sebe nezávislé
    - Neredukovateľne závislé na primárnom kľúči
  - Nekľúčový atribút: nie je súčasťou primárneho kľúča
  - Atribúty sú na sebe nezávislé, ak žiaden z nich nie je funkčne závislý na nejakej kombinácii tých ostatných (= každý atribút môžeme updatovať nezávisle od tých ostatných)

3NF ešte neformálnejšie

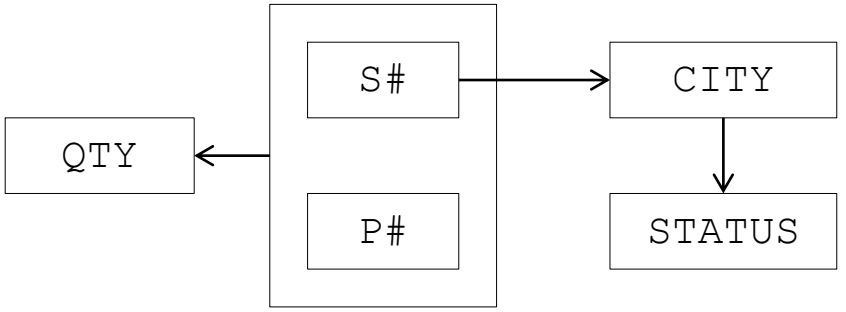
- Relačná premenná je v 3NF vtedy a len vtedy, keď každá n-tica pozostáva z primárneho kľúča, ktorý identifikuje nejakú entitu a množiny na sebe nezávislých atribútov, ktoré túto entitu nejakým spôsobom popisujú

Prvá normálna forma

- Relačná premenná je v 1NF vtedy a len vtedy, keď pre každú legálnu hodnotu tejto premennej obsahuje každý atribút každej n-tice práve jednu hodnotu
- Z definície pojmu relácia vyplýva, že každá relácia je v 1NF
- V praxi sa väčšinou hodnota atribútu typu RELÁCIA považuje za nelegálnu
- Relácie, ktoré sú v 1NF a nie v 2NF a 3NF majú nežiadúce vlastnosti

Relácia v 1NF a nie v 2NF a 3NF

```
First{S#, STATUS, CITY, P#, QTY}
   Primary Key{S#,P#}
```

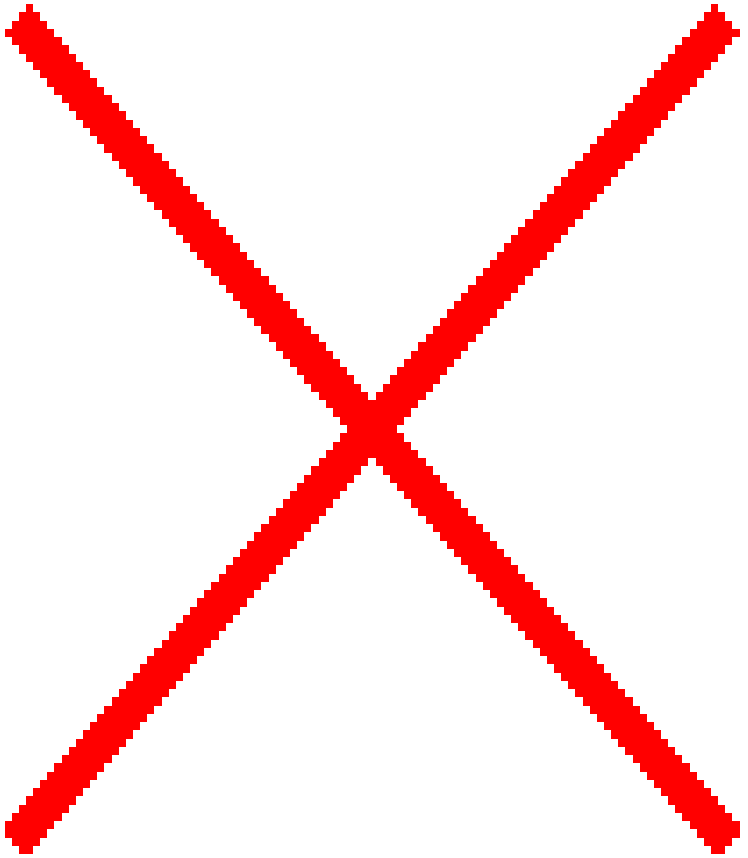


Cvičenie:

1. Vytvorte príklad relácie FIRST
2. Prečo nie je relácia v 3NF?
3. Aké problémy vznikajú v tejto relácii pri operáciách

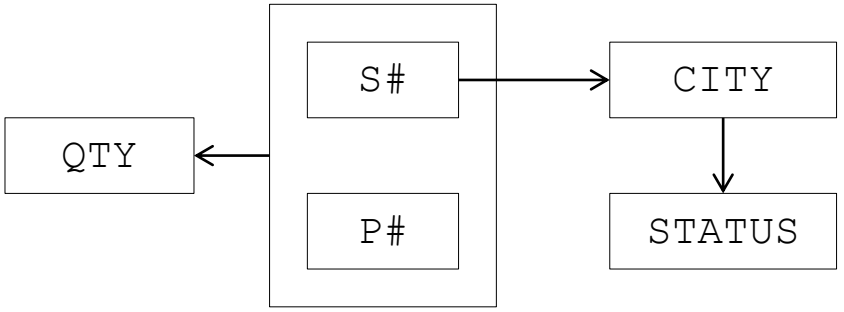
- insert
- update
- delete?

Problémy s reláciami v 1NF a nie v 2NF a 3NF



- Insert: nemôžeme uložiť fakt, že máme dodávateľa S5 v Pardubiciach, pokiaľ nedodáva žiaden produkt
- Update: Keď sa S1 presťahuje do Prahy, musíme túto informáciu updatovať pri všetkých dodávkach
- Delete: Zrušenie nejakého dodávateľa znamená vymazanie všetkých záznamov s týmto užívateľom
- Delete II.: Keď vymažeme dodávku P2 dodávateľom S3, stratíme aj informáciu o tom, že S3 sa nachádza v Paríži
- Riziko: Vznik nekonzistentných dát

Prečo nie je táto relácia v 3NF?



Nie všetky nekľúčové atribúty sú neredukovateľne závislé na primárnom kľúči: Je fakt, že platia FD {S#,P#} $\rightarrow$ {CITY} a {S#,P#} $\rightarrow$ STATUS, avšak platia aj FD {S#} $\rightarrow$ {CITY} a {S#} $\rightarrow$ STATUS. Pôvodné FD sa teda dajú redukovať

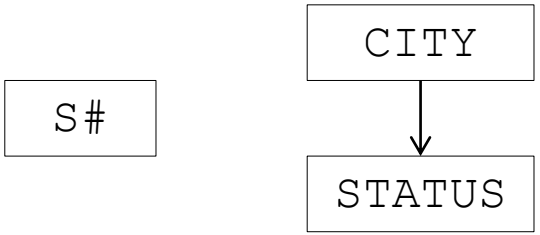
Nekľúčové atribúty nie sú na sebe nezávislé. Platí totiž FD {CITY} $\rightarrow$ {STATUS}

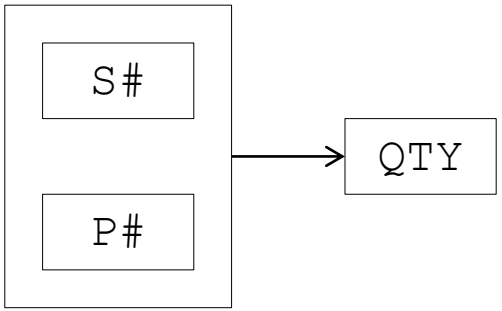
Druhá normálna forma

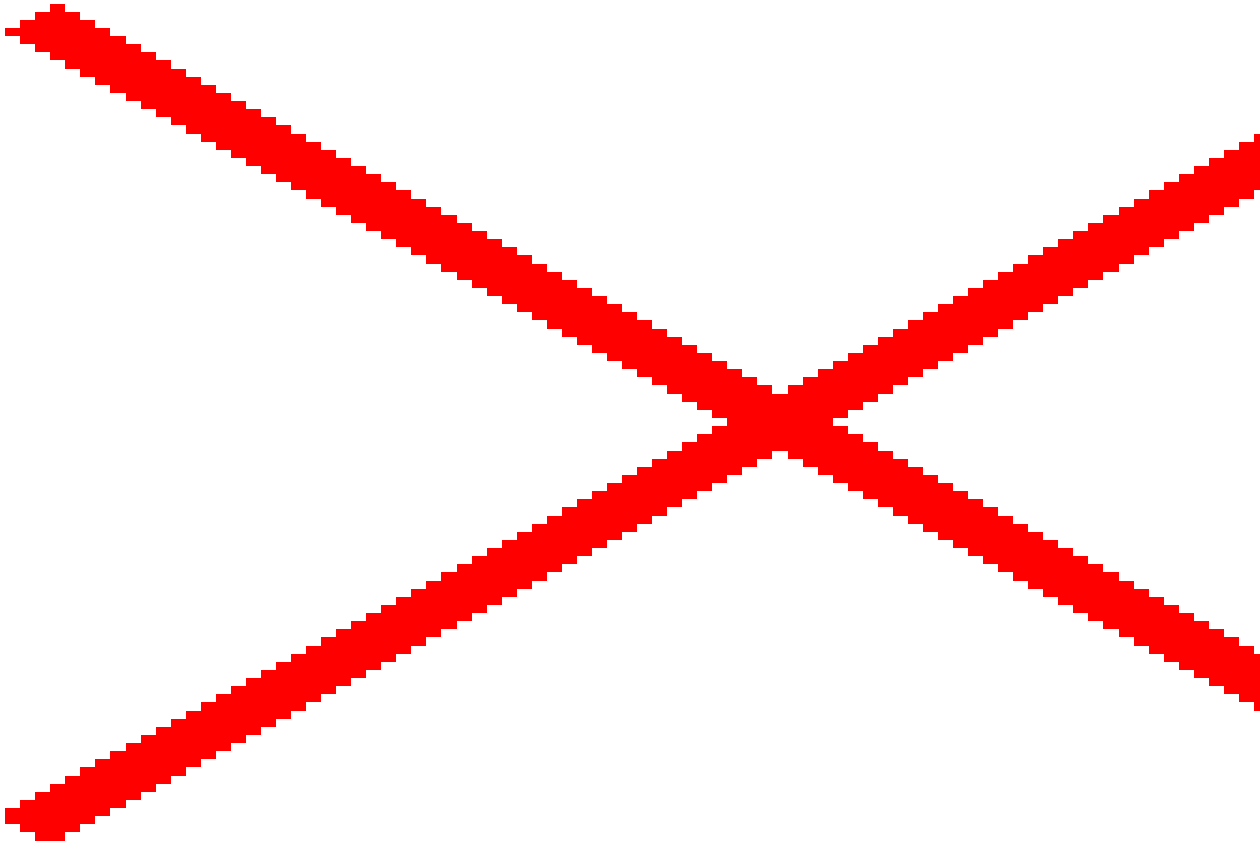
Relačnú premennú FIRST nahradíme dvoma novými premennými:

SECOND{S#, CITY, STATUS}

SP{S#, P#, QTY}







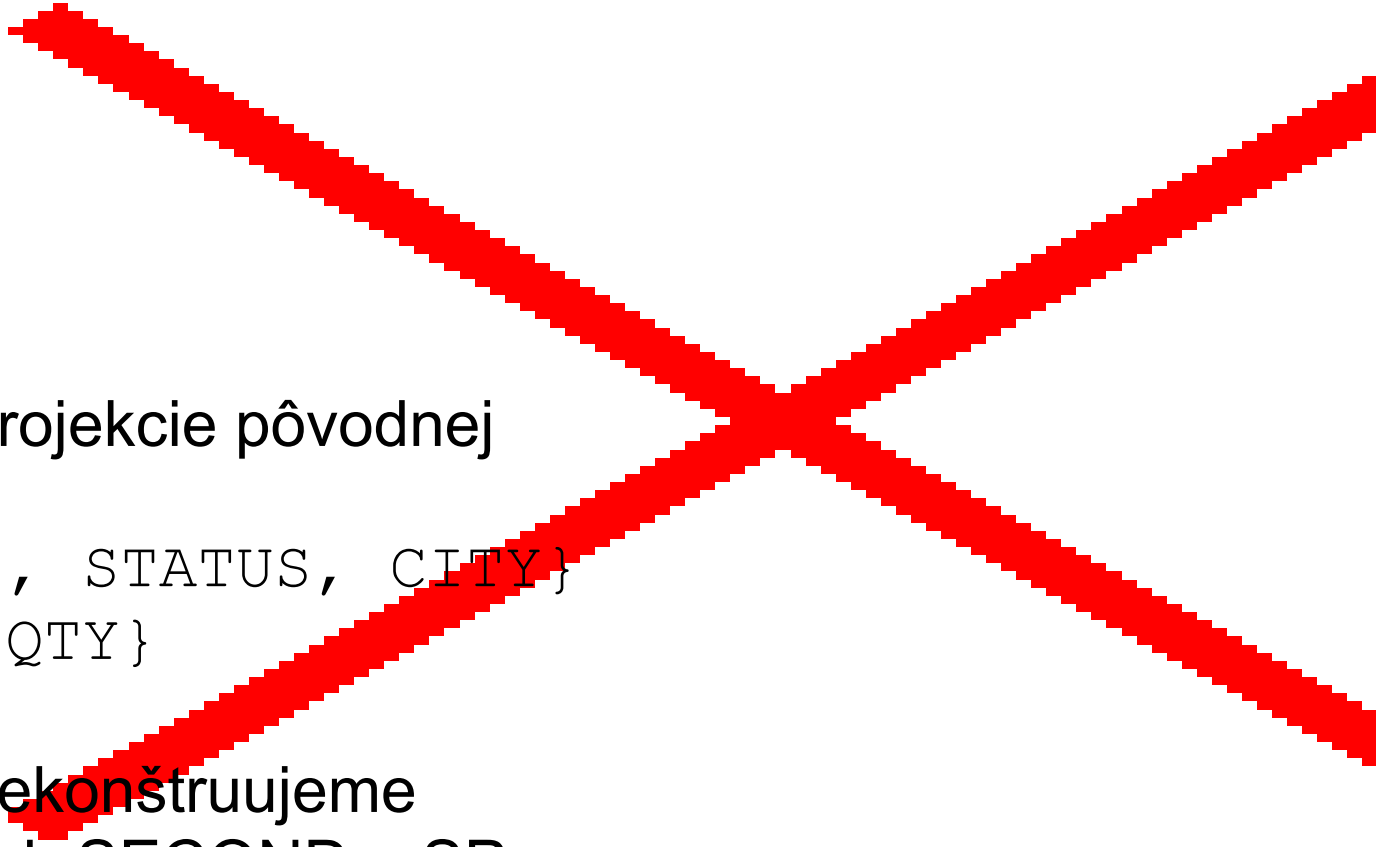
Druhá normálna forma: definícia

Relačná premenná je v druhej normálnej forme vtedy a len vtedy, keď

- je v prvej normálnej forme a
- každý nekľúčový atribút je neredukovateľne závislý na primárnom kľúči

- Relačná premenná, ktorá je v 1NF avšak nie v 2NF, sa vždy dá zredukovať na ekvivalentnú množinu relačných premenných v 2NF:
  - daná premenná sa nahradí množnou svojich projekcií
  - pôvodná premenná sa zrekonštruuje spojením (joinom) týchto projekcií

Príklad



SECOND a SP sú projekcie pôvodnej premennej FIRST:

```
SECOND=FIRST{S#, STATUS, CITY}
SP=FIRST{S#,P#,QTY}
```

Premennú FIRST zrekonštruujeme spojením premenných SECOND a SP cez atribút S#

Prvý krok normalizácie: zhrnutie

- Prvým krokom normalizácie je teda vytvorenie projekcií pôvodnej relačnej premennej za účelom eliminácie redukovateľných (“nonirreducible”) funkčných závislostí:
- Teda: premennú

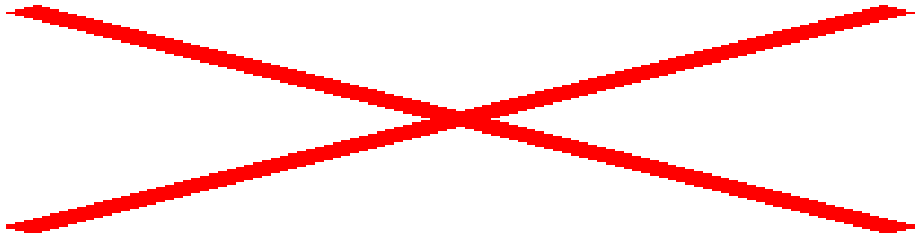
```
R{A,B,C,D}
Primary Key {A,B},
```

pričom platí $A \rightarrow D$

- Nahradíme dvoma novými premennými `R1{A,D}` a `R2{A,B,C}`

Problémy s reláciou v 2NF (a nie v 3 NF)

Cvičenie:



Problémom je vzájomná závislosť nekľúčových atribútov
PSC $\rightarrow$ Mesto, resp. tranzitívna závislosť atribútu mesto na atribúte Rodné číslo cez atribút PSČ
Rodne cislo $\rightarrow$ PSC, PSC $\rightarrow$ Mesto, Rodne cislo $\rightarrow$ Mesto

Podobne v relačnej premennej SECOND je nekľúčový atribút STATUS závislý na nekľúčovom atribúte CITY

Riešenie

Riešením je nahradiť reláciu SECOND jej dvoma projekciami:

```
SC{S#, CITY}

CS{CITY, STATUS}
```

FD Diagramy pre tieto relačné premenné:

3NF: formálna definícia

Definícia predpokladá iba jeden kandidátny kľúč, ktorý budeme ďalej označovať ako primárny kľúč.

Definícia:

Relačná premenná je v 3NF vtedy a len vtedy, ak je v 2NF a každý nekľúčový atribút je netranzitívne závislý na primárnom kľúči

Zachovanie závislostí

Relačnú premennú SECOND sme mohli bezstratovo dekomponovať aj takto:

SC{S#, CITY} a SS{S#, STATUS}

a relačnú premennú OSOBA:

OS1{Rodne cislo, Meno, Ulica, Mesto} a

OS2{Rodne cislo, PSC}

Problém? Stratili zme závislosť PSC→Mesto

Relačné premenné treba dekomponovať tak, aby sme závislosti nestrácali

Boyce-Coddova NF

Definícia 3NF (podľa Datea) vychádza z existencie jedného kandidátneho kľúča. Pôvodná Coddova definícia síce zohľadňuje viaceré kandidátne kľúče, je však neuspokojivá vtedy keď daná relačná premenná:

1. Má dva alebo viac kandidátnych kľúčov, ktoré
2. Sú zložené a
3. Prekrývajú sa

Pôvodná definícia 3NF bola nahradená silnejšou definíciou a nazvaná BCNF

BCNF: definícia

Formálna definícia: Relačná premenná je v BCNF vtedy a len vtedy, keď každá netriviálna, zľava neredukovateľná funkčná závislosť má za svoj determinant kandidátny kľúč.

-táto definícia je konceptuálne jednoduchšia než definícia 3NF lebo neobsahuje odkaz na 2NF

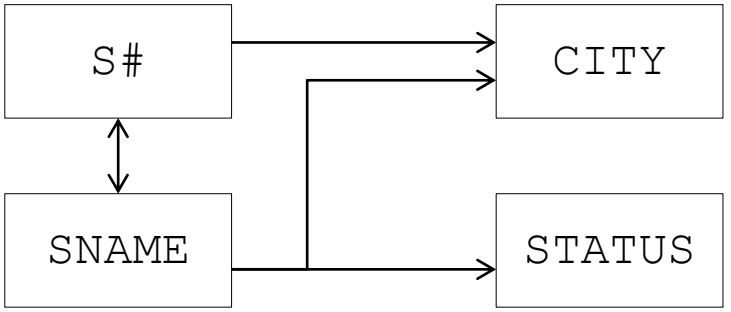
Neformálna definícia: Relačná premenná je v BCNF vtedy a len vtedy, keď každý determinant je kandidátny kľúč, alebo:

všetky šípky vo FD diagrame vychádzajú z kandidátnych kľúčov

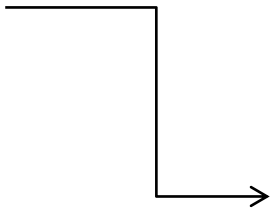
Príklad

`S{S#, SNAME, STATUS, CITY}`

Nech S# a SNAME sú kandidátne kľúče

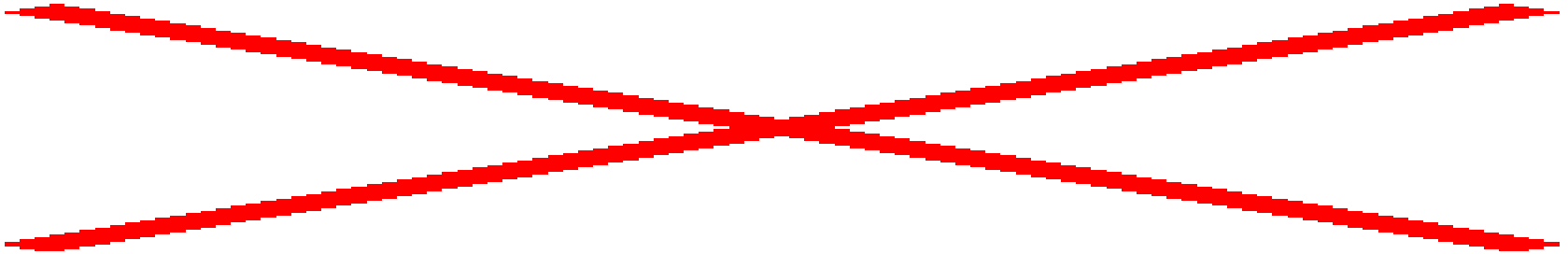


Relácia je v BCNF: všetky šípky vychádzajú z kandidátnych kľúčov



Cvičenie

Znormalizujte reláciu:



Pre nasledujúcu množinu kandidátnych kľúčov:
{DCS_NR}

a pre nasledujúcu množinu FD:

DCS_NR->FWP_NR
DCS_NR->BATCH_ID
DCS_NR->DOC_DOCTYPE
DCS_NR->AANTAL_PAGINAS
DCS_NR->ENDORSMENT_NR
FWP_NR->BATCH_ID

3NF versus BCNF

Ak vyjdeme zo zjednodušených definícií podľa Datea, môžeme povedať, že BCNF zahŕňa v sebe 3NF, pričom o 3NF sa dá hovoriť iba v prípade relačných premenných, ktoré majú iba jeden kandidátny kľúč.

Ak vyjdeme z pôvodných presných Coddových definícií, dajú sa vymyslieť relačné premenné, ktoré sú v BCNF a nie v 3NF

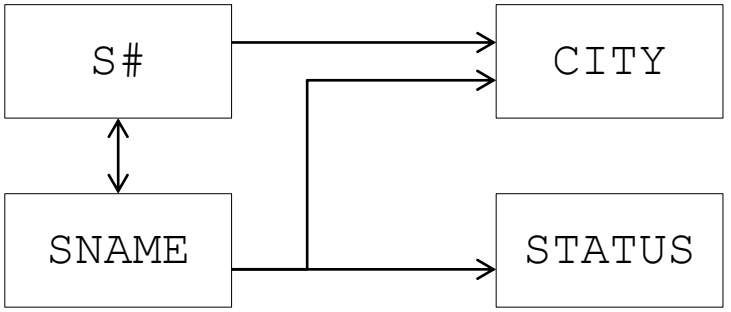
Pôvodné Coddove definície

- A prime attribute is an attribute that participates in at least one candidate key.
- A relation is in first normal form if ... none of its domains has elements which are themselves sets.
- A relation R is in second normal form if it's in first normal form and every nonprime attribute is fully dependent on each candidate key of R.
- A relation R is in 3NF if it's in 2NF and every nonprime attribute is nontransitively dependent on each candidate key of R.

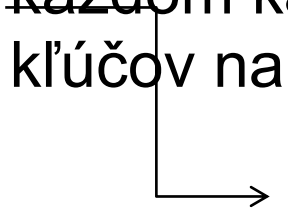
Príklad

`S{S#, SNAME, STATUS, CITY}`

Nech S# a SNAME sú kandidátne kľúče



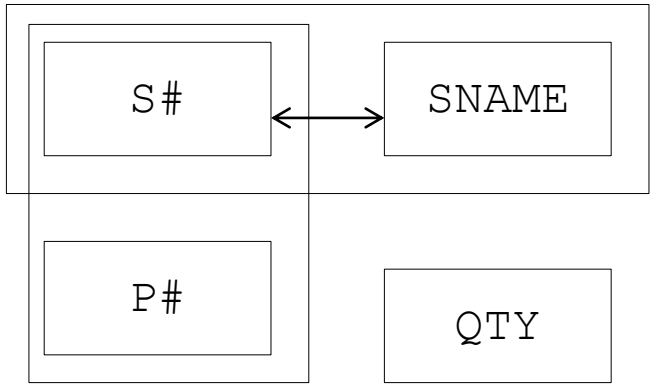
Relácia je v tiež v 2NF a 3NF podľa pôvodných definícií:
Nonprime atribúty STATUS a CITY sú netranzitívne závislé na každom kandidátnom kľúči. Definície nehovoria nič o závislosti kľúčov na sebe navzájom (S# a SNAME)



Príklad II.

`SSP{S#, SNAME, P#, QTY}`

Pri predpoklade jednoznačnosti atribútu SNAME sú kandidátnymi kľúčmi {S#, P#} a {SNAME, P#}



Táto relačná premenná je v 3NF (podľa pôvodnej Coddovej definície) avšak nie v BCNF: nie všetky šípky vychádzajú z kandidátnych kľúčov.

Cvičenie:

- Problémy?
- Riešenie?

Viachodnotové závislosti - úvod

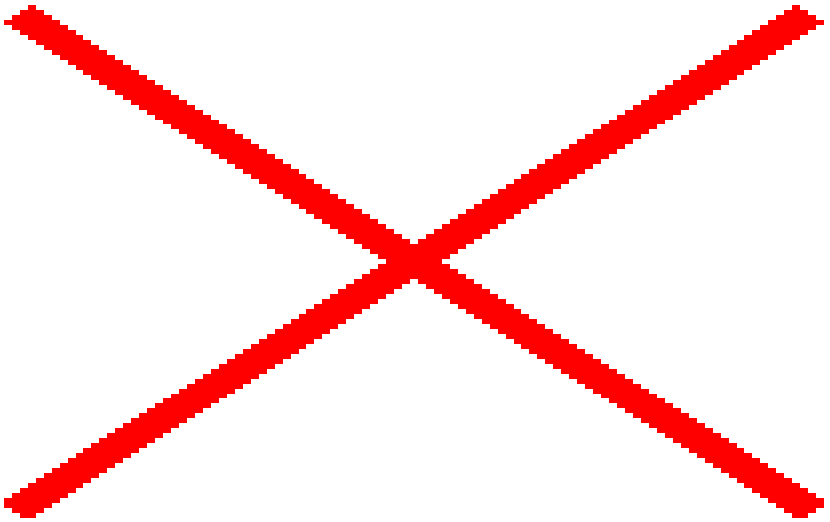
Multi-valued dependencies (MVD) sú zobecnením konceptu FD

Course	Teachers	Texts
Physics	Teacher: Prof. Green, Prof. Brown	Text: Basic Mechaniscs, Principles of Optics
Math	Teacher: Prof. Green	Text: Basic Mechaniscs, Vector Analysis, Trigonometry

Učitelia a texty sú na sebe nezávislé. Každý učiteľ môže používať ľubovoľný text.

Viachodnotové závislosti – úvod II.

Ten istý príklad bez atribútov typu RELATION:



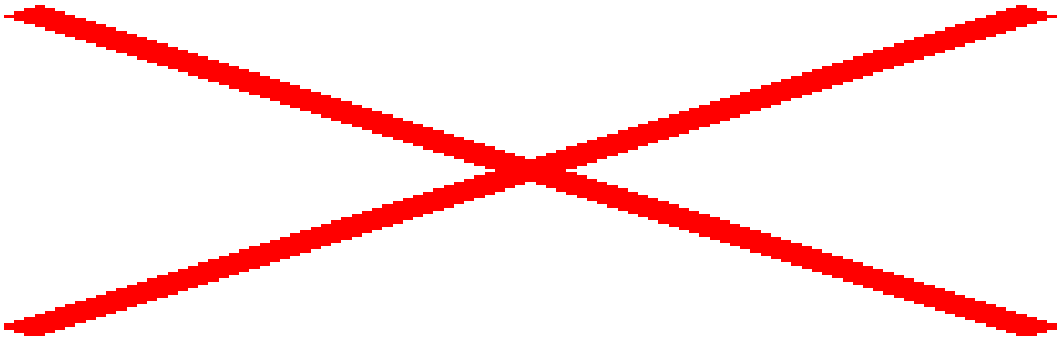
Nepríjemná vlastnosť (integritné obmedzenie):

Ak n-tice (c, t1, x1) a (c, t2, x2) sú obsiahnuté v danej relácii, tak aj n-tice (c, t1, x2) a (c, t2, x1) musia nutne byť obsiahnuté v tejto relácii

Ak sa objaví nová učebnica fyziky, tak ju musíme skombinovať s každým učiteľom. Naopak, každého nového učiteľa fyziky musíme skobinovať so všetkými učebnicami. Problémy takejto relácie sú zjavné.

Riešenie úvodného problému

Riešením je pochopiteľne dekomponovať pôvodnú relačnú premennú na dve nové:



Atribúty Teacher a Text v pôvodnej relačnej premennej sú viachodnotovo závislé na atribúte Course:

Course $\rightarrow\rightarrow$ Teacher a Course $\rightarrow\rightarrow$ Text

Viachodnotová závislosť: definícia

Nech R je relačná premenná a nech A, B a C sú podmnožiny množiny atribútov R. Hovoríme, že B je viachodnotovo závislé na A ($A \rightarrow\rightarrow B$) vtedy a len vtedy, keď pre každú prípustnú hodnotu R množina hodnôt atribútu B, ktorá patrí k danej kombinácii AC je závislá výlučne od hodnoty atribútu A (a teda úplne nezávislá od hodnoty atribútu C)

MVD ako zovšeobecnenie FD

FD je taká MVD, kde množina hodnôt závislých na danom determinante je jednoprvková množina

Problém relácií ako napríklad CTX je, že obsahujú MVD, ktoré nie sú FD

Faginov teorém

Nech R{A,B,C} je relačná premenná, kde A, B a C sú množiny atribútov. Potom R je rovné spojeniu projekcií {A, B} a {A,C} vtedy a len vtedy, keď R spĺňa viachodnotové závislosti $A \rightarrow\rightarrow B$ a $A \rightarrow\rightarrow C$

Faginov teorém je zovšeobecnením Heathovho teorému.

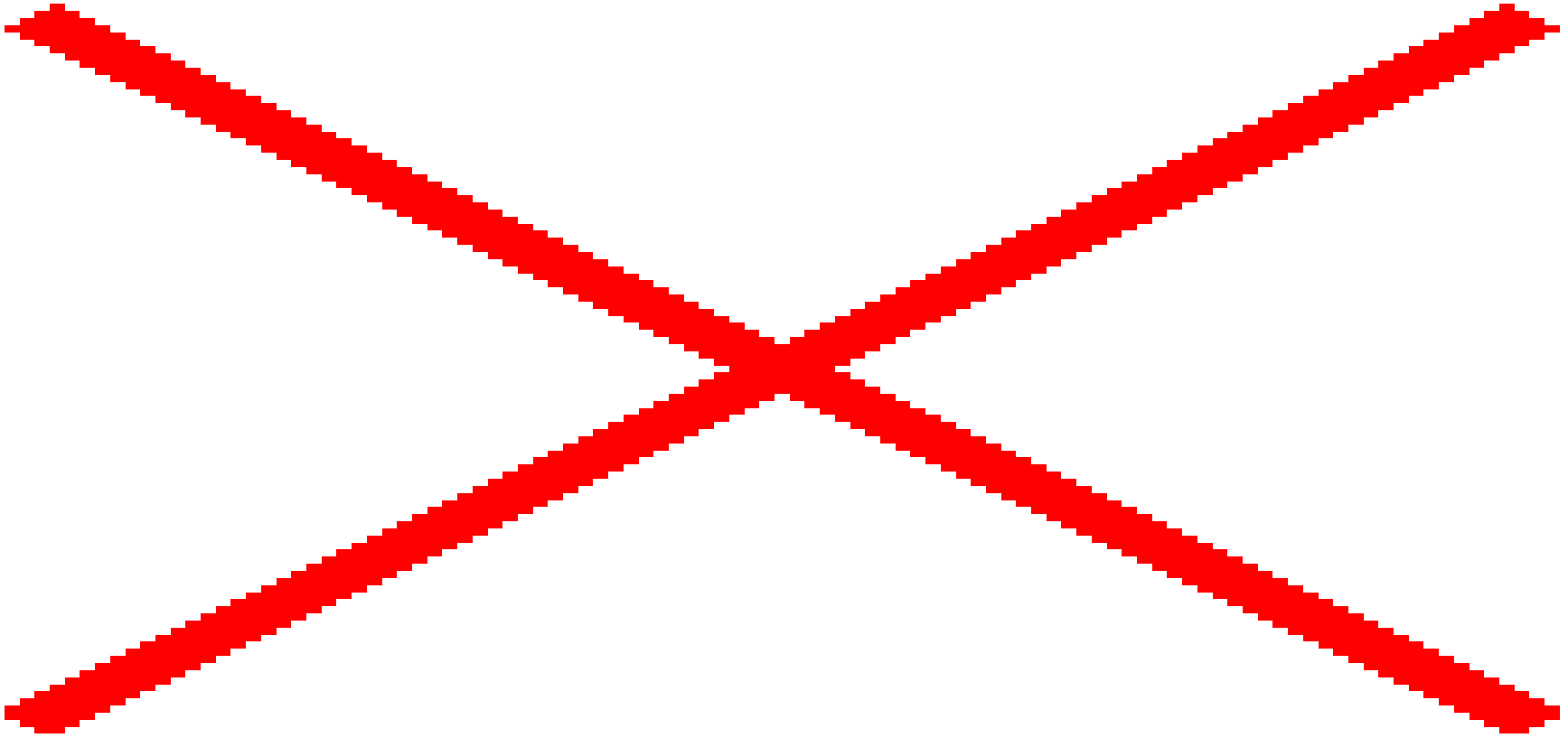
Štvrtá normálna forma

Relvar R is in 4NF if and only if, whenever there exist subsets A and B of the attributes of R such that the nontrivial MVD $A \rightarrow\rightarrow B$ is satisfied, then all attributes of R are also functionally dependent on A.

Inými slovami: jediné netriviálne závislosti v R sú vo forme $K \rightarrow X$, teda funkčné závislosti nekľúčových atribútov na nejakom superkľúči K

Premenná CTX nie je v 4NF, pretože obsahuje MVD, ktoré nie sú FD.

Ortogonálny dizajn: úvodné príklady



S3 sa MUSÍ objaviť zároveň SA a SB. Dôvodom je “The Closed World assumption”

Ortogonálny dizajn: prvé priblíženie

Problémom v uvedených príkladoch je, že relačné premenné majú prekrývajúce sa významy.

Princíp ortogonálneho dizajnu: Žiadne dve (základné) relačné premenné by v danej databáze nemali mať prekrývajúci sa význam

Dve relačné premenné nemôžu mať prekrývajúci sa význam, pokiaľ nie sú rovnakého typu

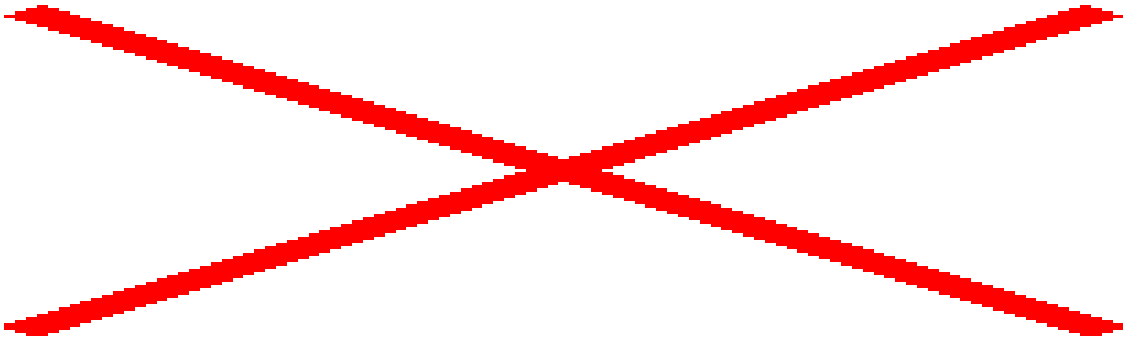
Dôsledok princípu ortogonálneho dizajnu

Pri vkladaní nejakej n-tice stačí špecifikovať, že táto n-tica má byť vložená do databázy, nemusíme teda špecifikovať do ktorých relačných premenných

Existuje totiž práve jedna relačná premenná, do ktorej daná n-tica patrí

Tento dôsledok sa dá rozšíriť na všetky operácie, nielen INSERT

Ortogonálny dizajn: druhé priblíženie



Premenné SX a SY nemajú prekrývajúci sa význam.
Ich projekcie {S#, SNAME} však áno.

Ortogonálny dizajn: definícia

Nech A a B sú dve rozličné základné relačné premenné. Potom nesmú existovať bezstratové dekompozície premenných A a B na A_1, A_2,..., A_m a B_1, B_2,..., B_n také, že niektorá projekcia A_i z množiny A_1, A_2,..., A_m a niektorá projekcia B_i z množiny B_1, B_2,..., B_n majú prekrývajúci sa význam

Z definície bezstratovej dekompozície nie je žiadna projekcia potrebná na rekonštrukciu pôvodnej rel. premennej redundantná. Preto v našom príklade projekcie {S#} síce majú prekrývajúci sa význam avšak neporušujú princíp ortogonálneho dizajnu.

Návrhové vzory

Alebo

Someone has already solved your problems

Úvod

- Našim cieľom je „dobrý OO dizajn“
- Vieme už, čo je to dobrý dizajn
- Jeden zo spôsobov ako dobrý dizajn dosiahnuť je použitie návrhových vzorov
- Návrhové vzory sú obecné riešenia často sa opakujúcich návrhových problémov
- Vzory nie sú teoretická konštrukcia. Sú zbierkou dlhoročných skúseností profesionálnych návrhárov

Literatúra

- Gamma, Helm, Johnson, Vlissides: Design Patterns. Addison Wesley, 1995
- Freeman: Head First Design Patterns (A Brain-Friendly Guide). O´Reilly 2004
- www.objects.cz

„Gang of four“

- Erich Gamma
- Richard Helm
- Ralph Johnson
- John Vlissides
- Kniha „Design patterns. Elements of reusable object-oriented software“
- Návrh štruktúry definície návrhových vzorov

Definície návrhových vzorov

- Názov a klasifikácia
- Zámer (Intent)
- Iné známe názvy
- Motivácia
- Aplikovateľnosť
- Štruktúra vzoru
- Účastníci
- Spolupráca
- Dôsledky
- Implementácia
- Ukážka kódu
- Známe použitia
- Príbuzné vzory

Druhy návrhových vzorov

- Návrhové vzory sa dajú rozdeliť do troch skupín:

- Creational
  - Singleton
- Structural
  - Facade
  - Adapter
  - Decorator
- Behavioral
  - Observer

Ako používať návrhové vzory

- Treba ich poznať. (Naučiť sa)
- Pri návrhu sa neustále snažiť posúdiť, či máme do činenia so situáciou, ku ktorej sa hodí nejaký vzor
- Ak áno, tak situáciu vyriešime podľa vzoru
- Pridaná hodnota vzorov: spoločný jazyk návrhárov

Čo návrhové vzory NIE SÚ

- Návrhové vzory nie sú hotové knižnice, ktoré by už obsahovali hotové riešenia
- Nemôžete ich priamo zapojiť do kódu
- Vzory treba zapojiť do hlavy: sú to spôsoby myslenia

Mapovanie objektového modelu do relačného modelu

Úvod

- Problém: aplikácia používajúca databázu je často postavená objektovo
- Rozdiel medzi objektovým modelom dát v aplikácii a relačným modelom dát v databáze
- Objekty z aplikácie je treba uložiť do riadkov relačných tabuliek

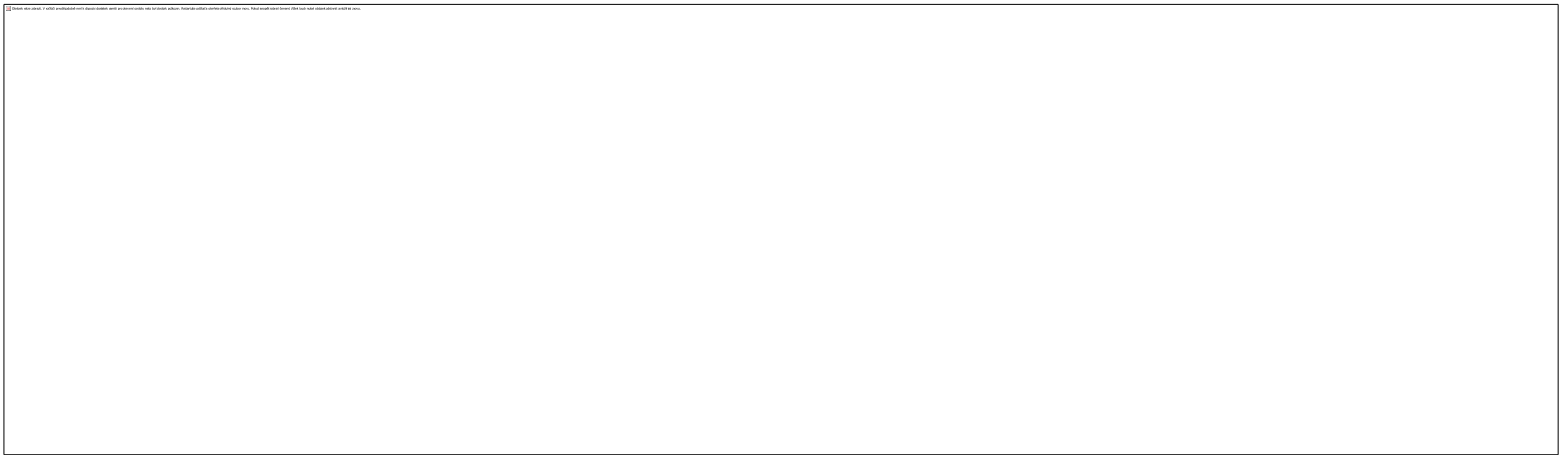
Konceptuálny model databázy

Od ERD k databáze

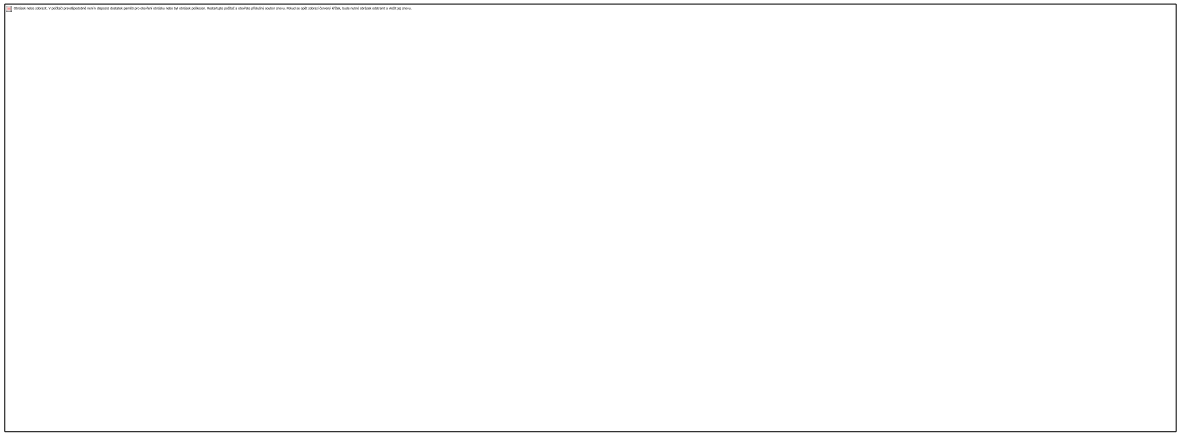
- Každému entitnému typu bude zodpovedať jedna tabuľka databázy
- Každému atribútu z ERD bude zodpovedať jeden atribút tabuľky
- Identifikačný kľúč sa stane primárnym kľúčom tabuľky
- Vzťahy sa realizujú pomocou cudzích kľúčov

Cudzí kľúč

- Nejaký (ne)kľúčový atribút entitného typu A sa stane súčasťou entitného typu B a slúži na prepojenie výskytov entít typu A a B

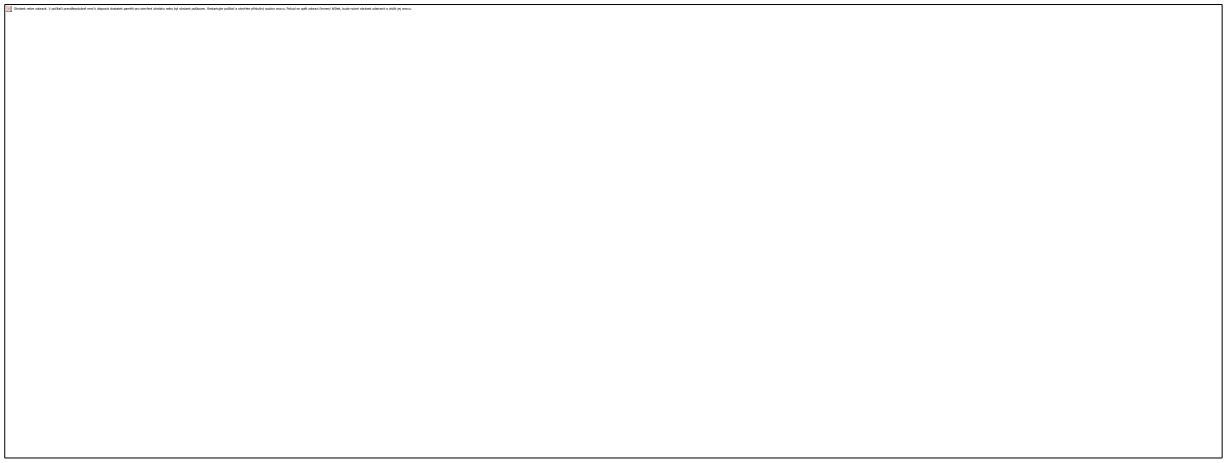


Cudzie kľúče v prípade vzťahu 1:1 (extrémne riešenie)



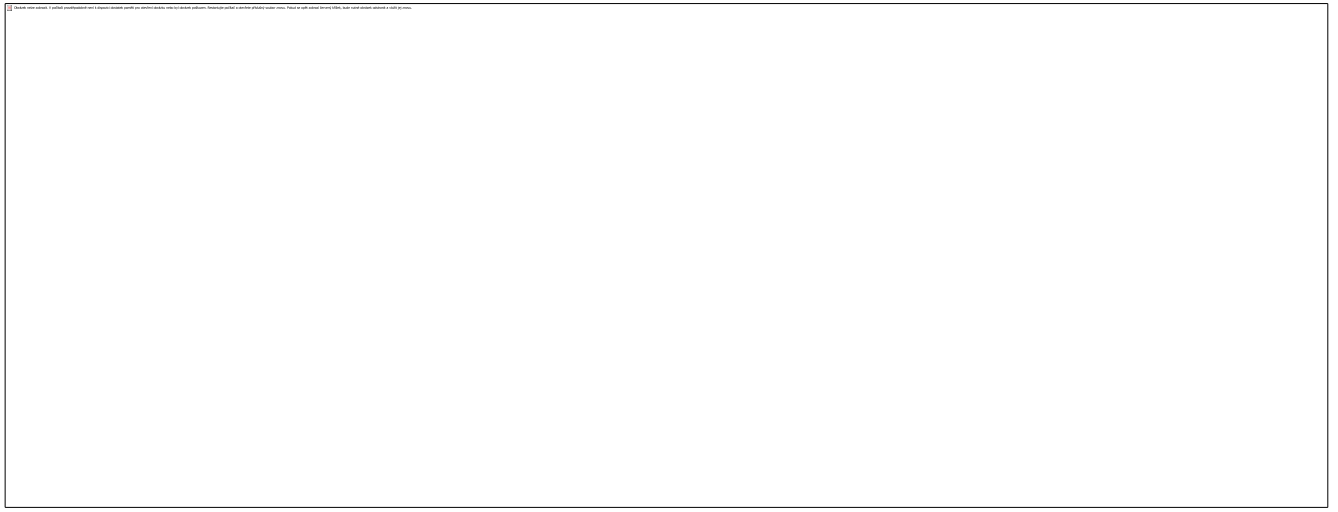
Primárny kľúč entitného typu A je kandidátnym kľúčom entitného typu B a zároveň primárny kľúč entitného typu B je kandidátnym kľúčom entitného typu A

Cudzie kľúče v prípade vzťahu 1:1 (jednoduchšie riešenie)



Primárny kľúč entitného typu A je kandidátnym kľúčom entitného typu B

Cudzie kľúče v prípade vzťahu 1:1 (ešte jednoduchšie riešenie)



Obidva entitné typy majú rovnaký primárny kľúč

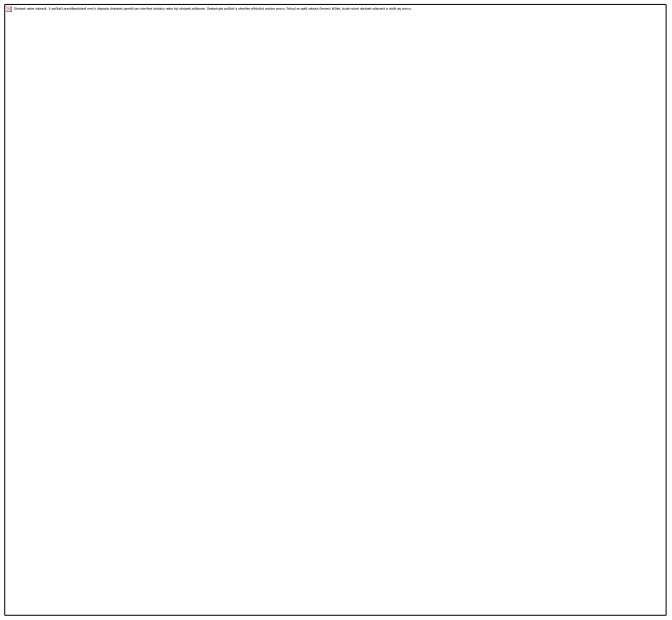
Cudzie kľúče v prípade vzťahu 1:1 (doplnok)

Vyššieuvedené pravidlá budú platiť, aj keď výraz „primárny kľúč“ nahradíme výrazom „kandidátny kľúč“

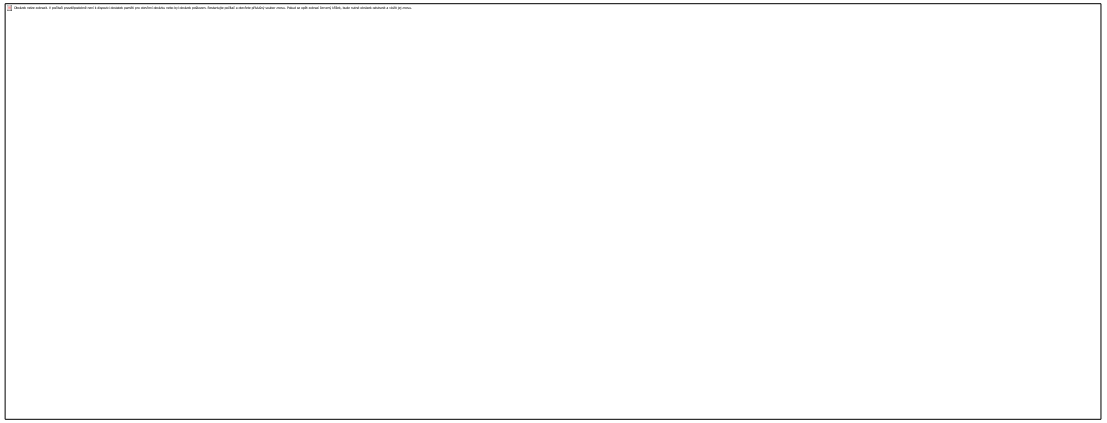
Kandidátny kľúč entitného typu A je zároveň kandidátnym kľúčom entitného typu B

Cudzie kľúče v prípade vzťahu 1:1 (zlúčenie entitných typov)

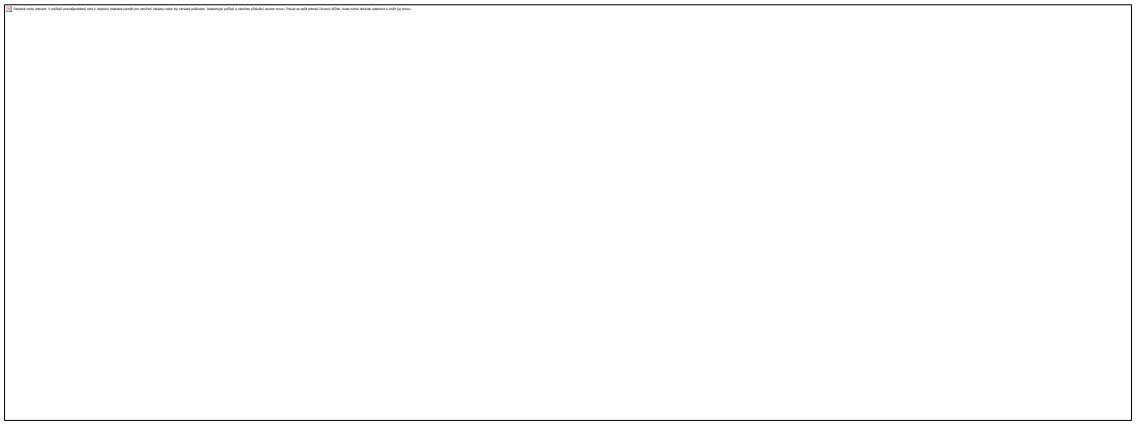
- V prípade vzťahu 1:1 je možné entitné typy A a B zlúčiť do jedného entitného typu
- Otázka: akú má toto riešenie nevýhodu?



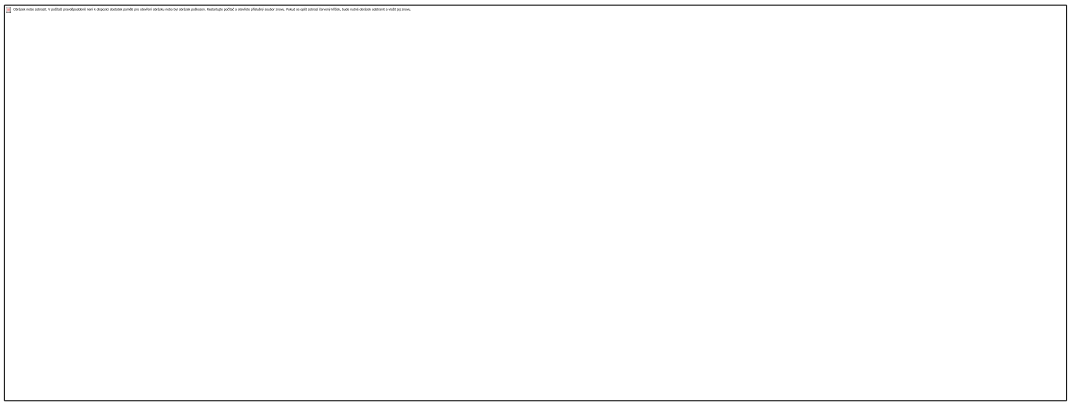
Cudzie kľúče v prípade vzťahu 1:N (alternatíva 1)



Kandidátny kľúč entitného typu A je súčasťou kandidátneho kľúča entitného typu B



Cudzie kľúče v prípade vzťahu 1:N (alternatíva 2)

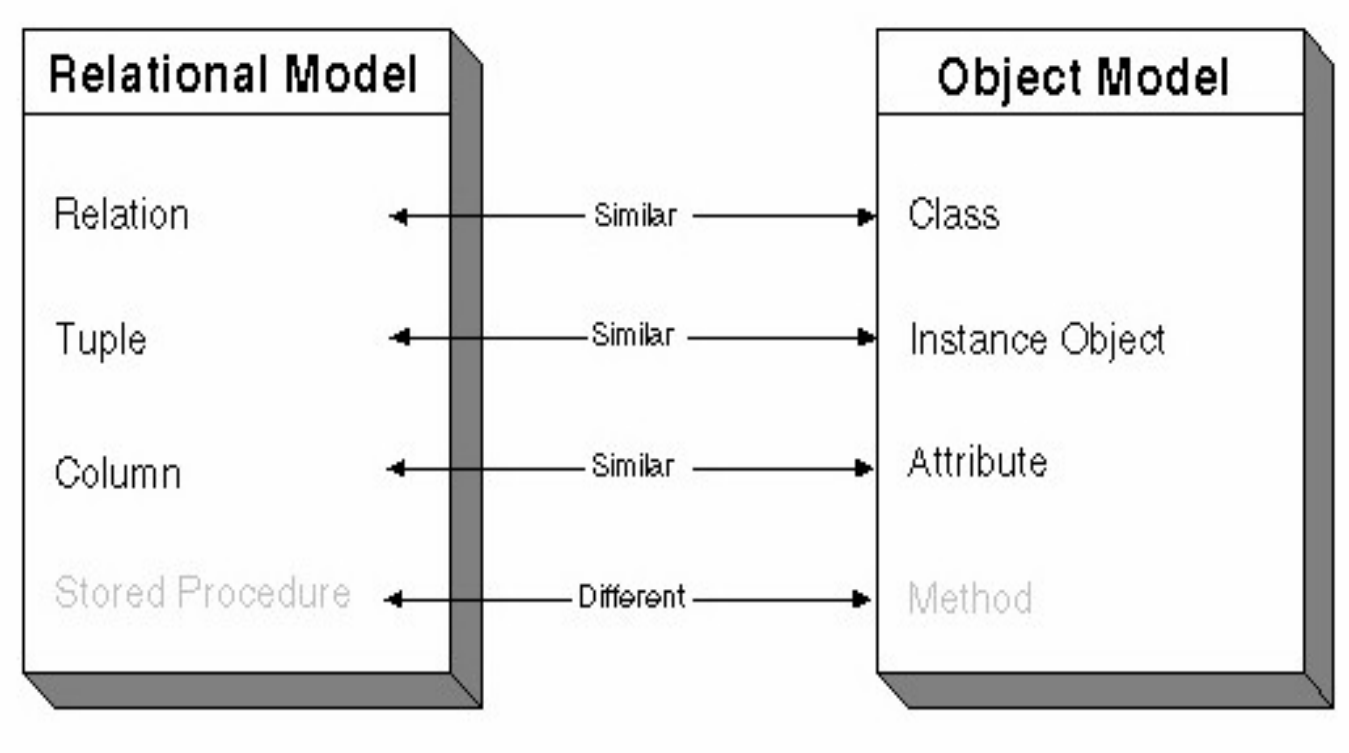


Kandidátny kľúč entitného typu A je nekľúčovým atribútom entitného typu B

Vzťah M:N

- Vzhľadom na to, že predpokladáme, že každý dátový model je v 3NF, nebudeme tu vzťah M:N rozoberať.
- Prečo?

Objektovo-relačné mapovanie



- Mapovanie medzi relačným a objektovým modelom, tak ako býva (v objektovom svete) veľmi často prezentované
- Je nesprávne. Prečo?

Objektový model vs. Relačný model

- Otázka: čo je ekvivalentom relácie („relačnej tabuľky“) v objektovom svete?
- Relácia je typový generátor. Obsahuje množinu prvkov nejakého typu (n-tice)
- Ekvivalentom by teda mal byť nejaký typový generátor (kolekcia)

Entitný typ versus trieda

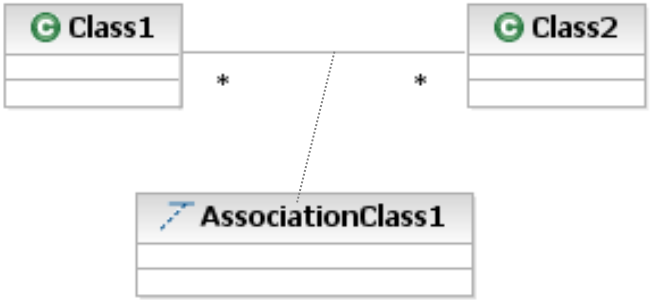
- Relačný model:
  - ERD obsahuje entitné typy
  - Pre každý entitný typ bude vytvorená relácia (relačná tabuľka)
- Objektový model
  - Model tried obsahuje triedy
  - Pre každú triedu bude vytvorené... čo vlastne?
- Pri objektovo-relačnom modelovaní bude triede zodpovedať jeden alebo viac entitných typov

Typy asociácií

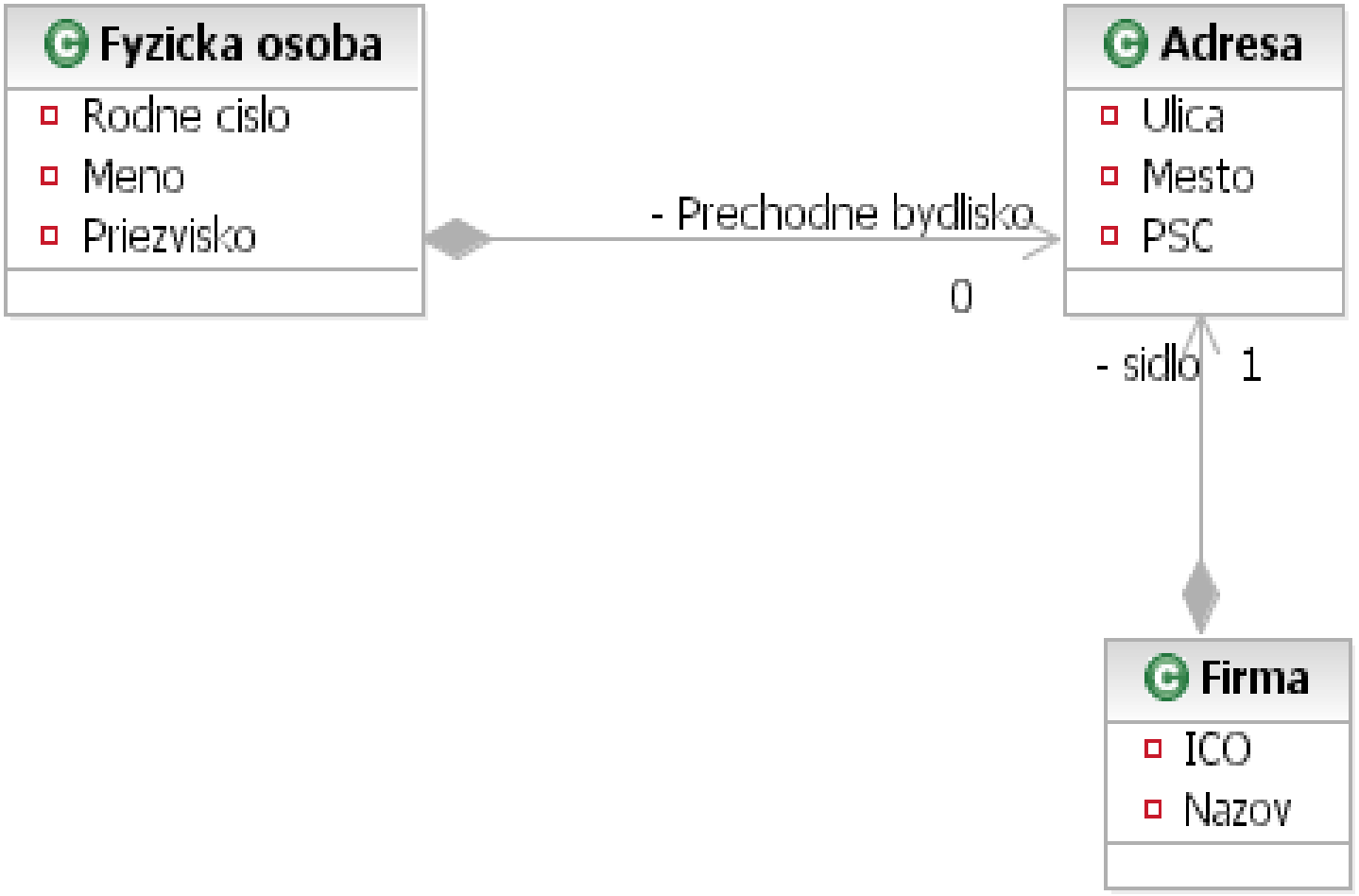
- Kompozícia

- Agregácia

- Asociačná trieda



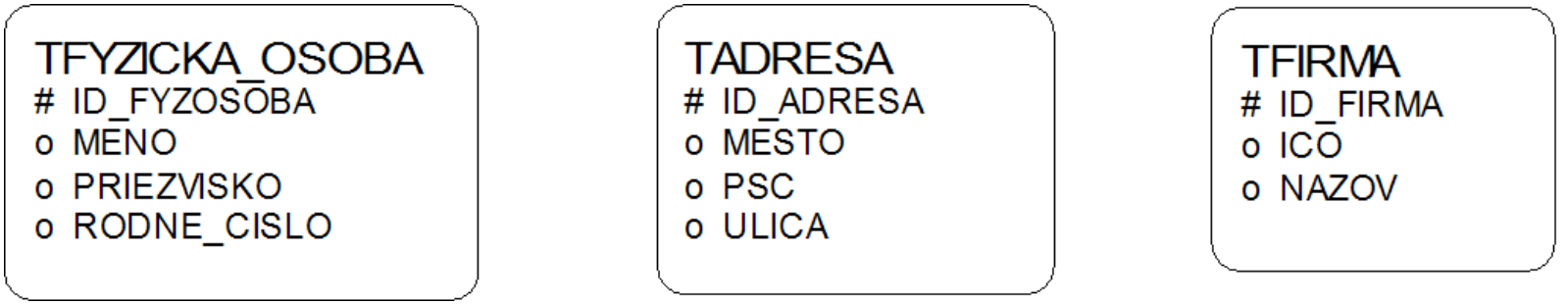
Príklad


Fyzicka osoba
Rodne cislo
Meno
Priezvisko
- Prechodne bydlisko
0
Adresa
Ulica
Mesto
PSC
- sidlo
1
Firma
ICO
Nazov

Čisté mapovanie kompozície “ku jednej” do relačného modelu

- Každá trieda z príkladu bude reprezentovaná jednou reláciou (tabuľkou)

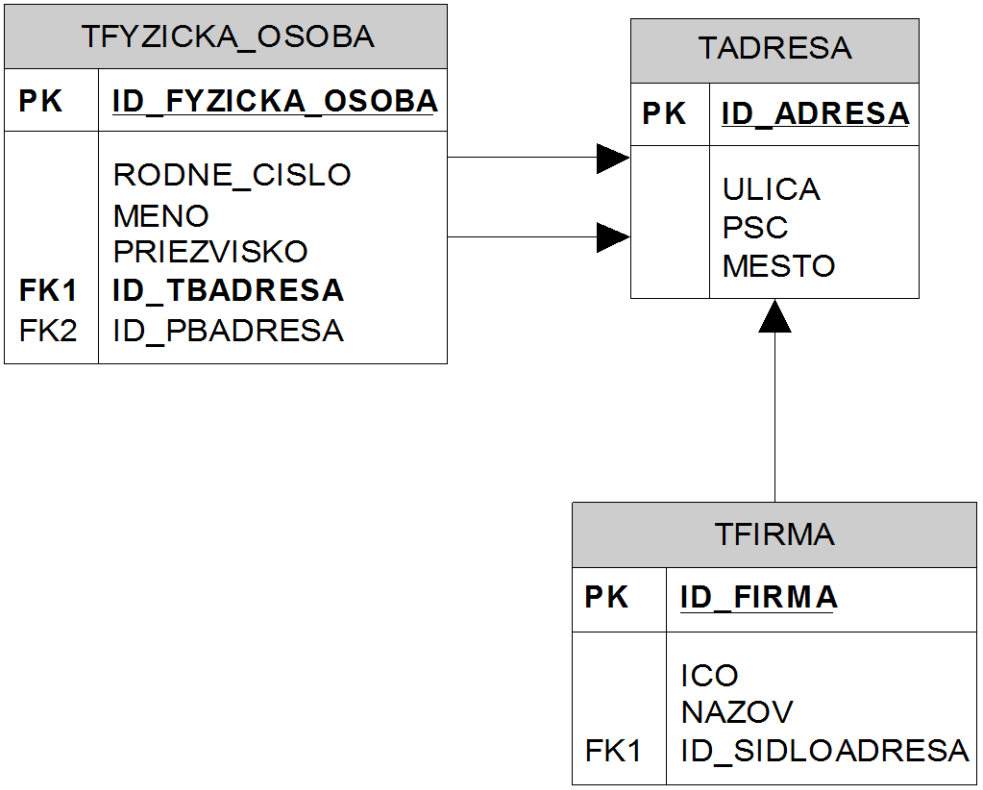
- Čo však asociácie? Kvôli jednoduchosti príkladu vybavíme každú tabuľku systémovým primárnym kľúčom



Potrebujeme modelovať nasledujúce vzťahy

- Každý výskyt objektu triedy FYZICKA_OSOBA obsahuje jeden vyskyt objektu triedy ADRESA v roli *trvale bydlisko*
- Každý výskyt objektu triedy FYZICKA_OSOBA obsahuje jeden vyskyt objektu triedy ADRESA v roli *prechodne bydlisko*
- Každý výskyt objektu triedy FIRMA obsahuje jeden vyskyt objektu triedy ADRESA v roli *sidlo*

Vzťahy modelujeme pomocou cudzích kľúčov



Mapovanie kompozitu “ku jednej” s rozpustením kompozitu

TFYZICKA_OSOBA	
PK	**ID_FYZICKA_OSOBA**
	RODNE_CISLO MENO PRIEZVISKO TB_ULICA TB_MESTO TB_PSC PB_ULICA PB_MESTO PB_PSC

TFIRMA	
PK	**ID_FIRMA**
	ICO NAZOV SIDLO_ULICA SIDLO_MESTO SIDLO_PSC

- Tabuľka TAdresa zanikla
- Atribúty tejto tabuľky boli prenesené do iných tabuliek a premenované
- „Rozpustenie“ je dobrá paralela toho, čo sa stalo

Problémy s rozpustením kompozitu: strata znovupoužiteľnosti

- Ak sa napríklad zmení štruktúra adresy, budú sa musieť zmeniť všetky tabuľky, ktoré adresu obsahujú
- Ak na nejakej ulici dôjde k prečíslovaniu, budú sa musieť všetky riadky všetkých tabuliek, ktoré prečíslované adresy obsahujú

Problémy s rozpustením kompozitu: strata transparentnosti návrhu

- Strata transparentnosti návrhu je nepríjemnejší problém než problémy s opakovaním, ktoré sme si práve popísali
- Príklad: je treba zmeniť štruktúru adresy (pridať okres)
- Pri riešení takejto úlohy môžu v súvislosti s objektovo-relačným mapovaním nastať tri rôzne situácie

Situácia 1. (veľmi priaznivá)

- Tabuľka TAdresa nebola rozpustená a existuje analytický model tried
- Pri pridaní okresu sa zásah týka iba jednej tabuľky a nie toho, kto jej údaje využíva
- V analytickom modeli sa zmení trieda Adresa a všetko je v poriadku

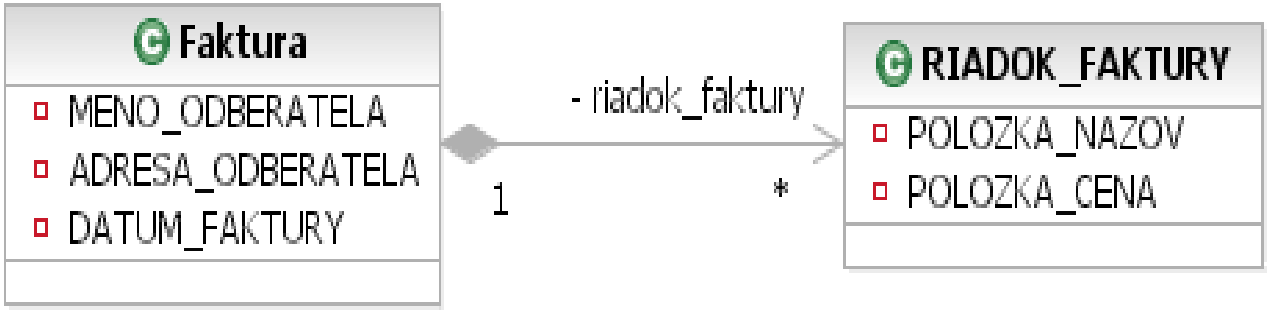
Situácia 2. Priaznivá ale pracná

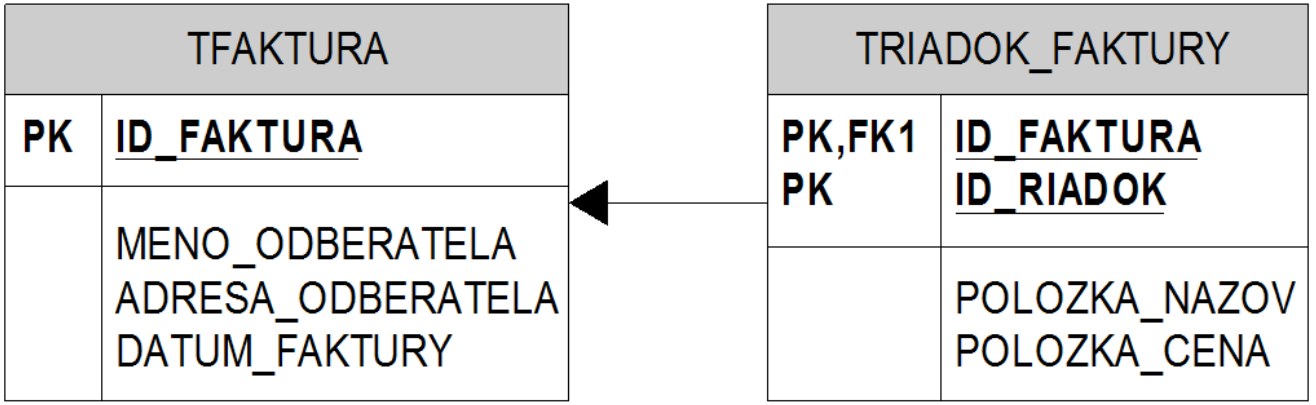
- Tabuľka TAdresa bola rozpustená ale máme k dispozícii analytický model tried, model databázy a mapovanie medzi nimi
- Vieme preto, kde všade sa adresy nachádzajú a môžeme bezpečne vykonať zmeny všade kde treba

Situácia 3. Nepriaznivá

- Táto situácia môže potenciálne spôsobiť katastrofu
- Trieda TAdresa bola rozpustená a modely nie sú k dispozícii
- Miesta, kde je treba previesť zmenu, je nutne pracne vyhľadávať a riziko, že na niečo zabudneme, je vysoké
- Aspket zvyšujúci riziko: pri rozpúšťaní tried môže dôjsť k premenovaniu stĺpcov. Bez modelov je ich vyhľadávanie potom veľmi ťažké

Mapovanie asociácie “ku N” do relačného modelu “čisté”





Uvedený príklad je mapovaním kompozície. Rovnako sa mapuje aj agregácia

Mapovanie kopmpozície “ku N” s rozpustením kompozitu.

V praxi: “denormalizovaná” “tabuľka”

TFAKTURA	
PK	**ID_FAKTURA**
	MENO_ODBERATELA ADRESA_ODBERATELA DATUM_FAKTURY POLOZKA_NAZOV1 POLOZKA_CENA1 POLOZKA_NAZOV2 POLOZKA_CENA2 POLOZKA_NAZOV3 POLOZKA_CENA3

Mapovanie kopmpozície “ku N” s rozpustením kompozitu II.

“Rozšírený” relačný model pripúšťa atribúty typu relácia:

```
Type TRIADOK_FAKTURY = RELATION {
                                ID_RIADOK int,
                                POLOZKA_NAZOV
char(50)                                        POLOZKA_CENA
money
                                }
Type TFAKTURA = RELATION   {
                      ID_FAKTURA int,
                      MENO_ODBERATELA char(50),
                      ADRESA_ODBERATELA char(50),
                      DATUM_FAKTURY datetime,
                      RIADKY_FAKTURY TRIADOK_FAKTURY
                      }
```

TFaktura

ID	MENO ODBERATELA	ADRESA ODBERATELA	DATUM
F1	Univerzita Pardubice	Studentská 1 Pardubice	23-01-09
F2	J. Novák	Vodičkova 1 Praha	16-02-09
F3	A. Jansen	Damrak 3 Amsterdam	01-02-09

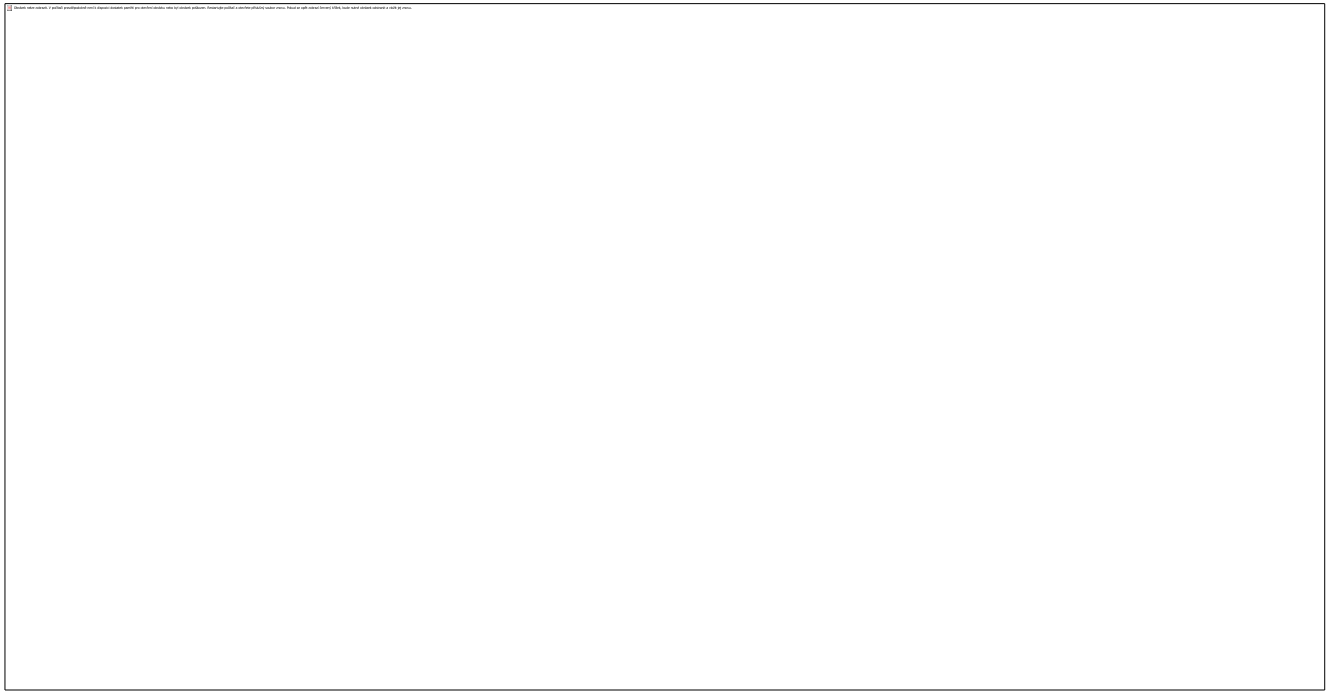
RIADKY FAKTURY

ID	POLOZKA NAZOV	CENA
R1	Polozka1	2000
R2	Polozka2	3000
R3	Polozka3	4000
R1	Polozka1	200
R2	Polozka2	300
R1	Polozka1	2500
R2	Polozka2	3000

Pozor!

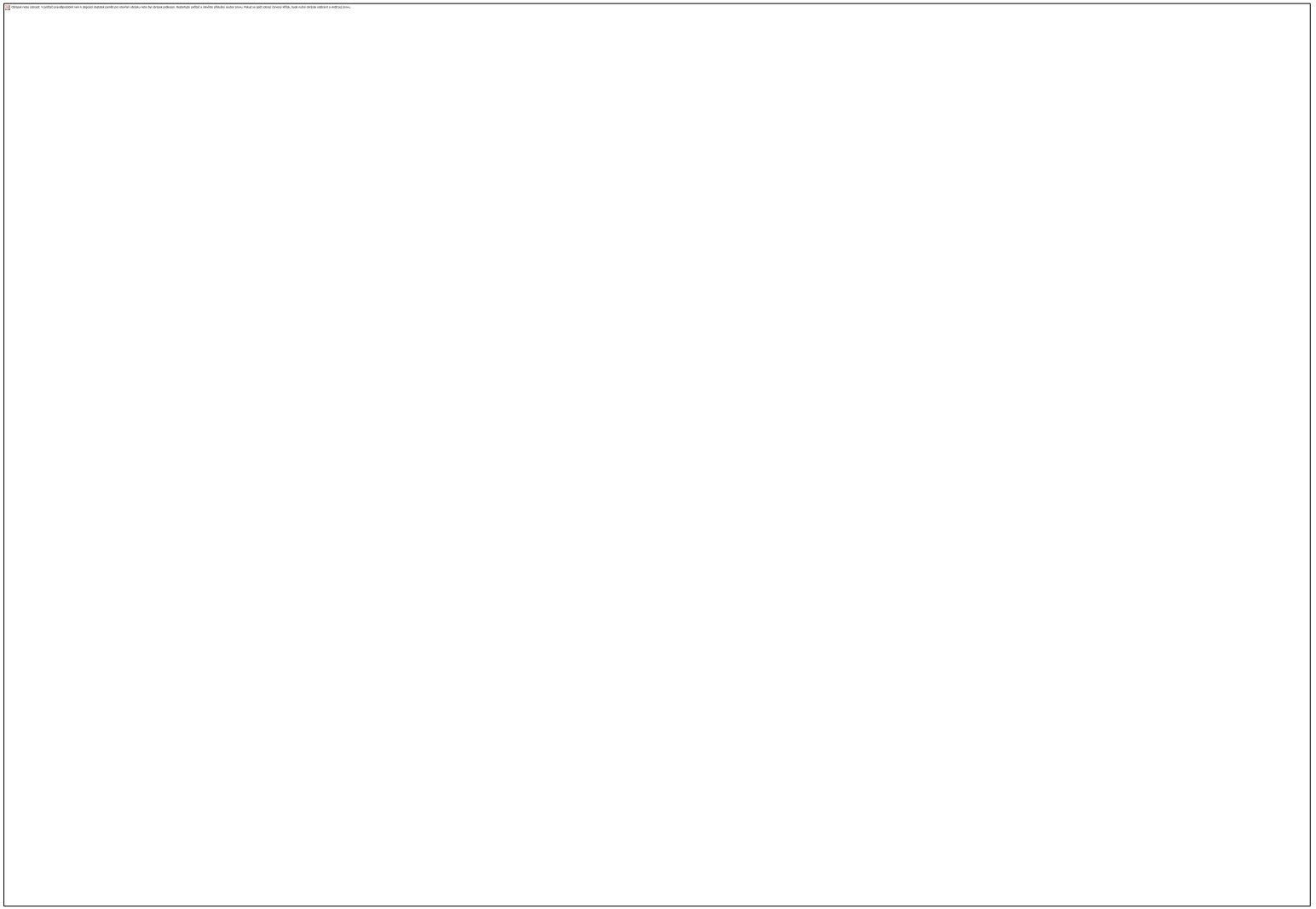
- To, čo sme si povedali, platí na logickej úrovni (na úrovni typov)
- V praxi to bude vyzerať tak, že nie triedam typu <<entity>> budú zodpovedať relačné tabuľky, ale kolekciám týchto tried

Asociačné triedy

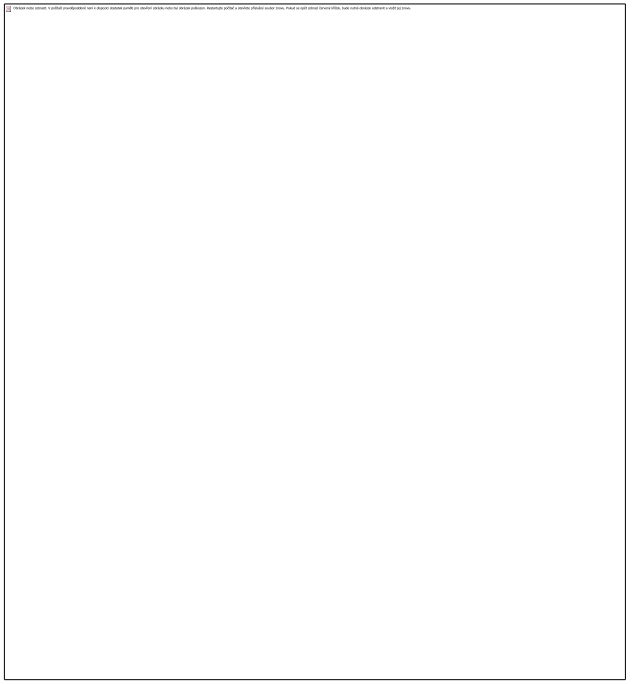


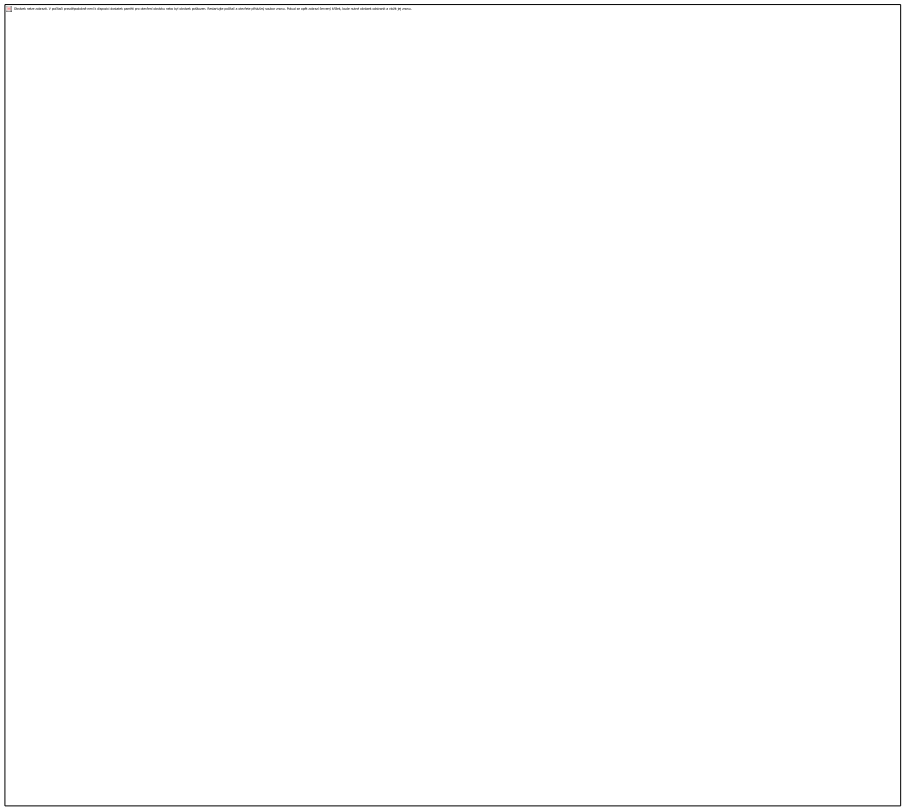
- Asociačné triedy často reprezentujú vzťahy M:N
- Asociačné triedy reprezentujú asociácie, ktoré majú vlastnosti zaujímavé z hľadiska modelu (modelovenej aplikácie)

Asociačná trieda v relačnej databáze

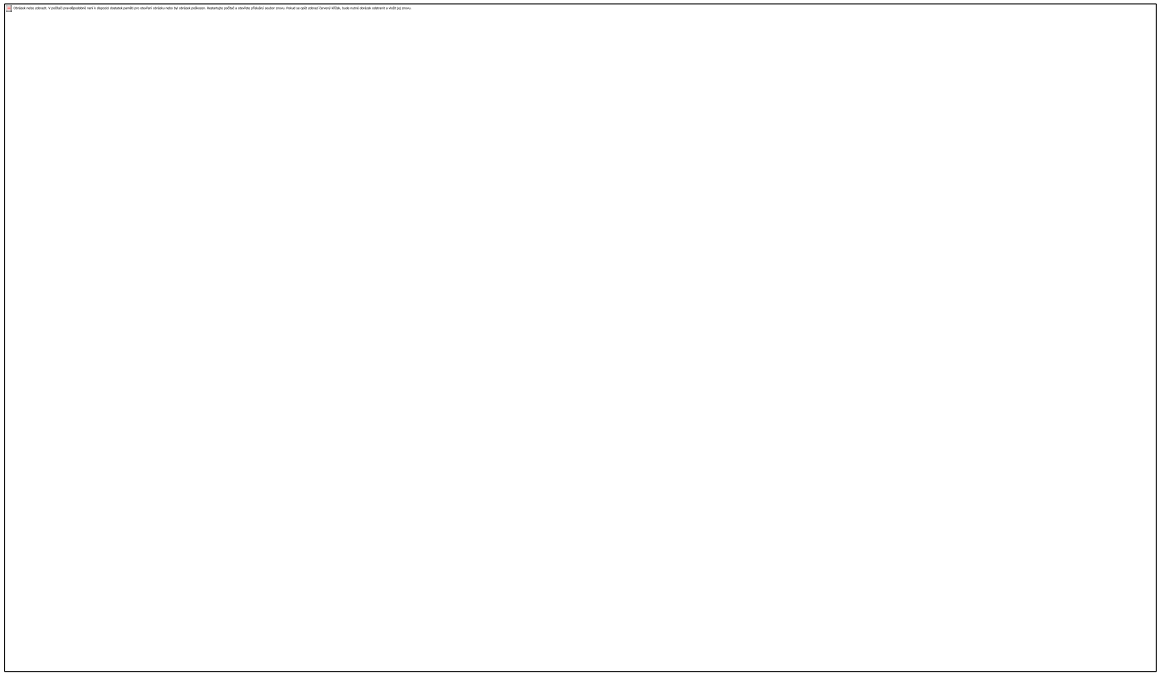


Hádanka

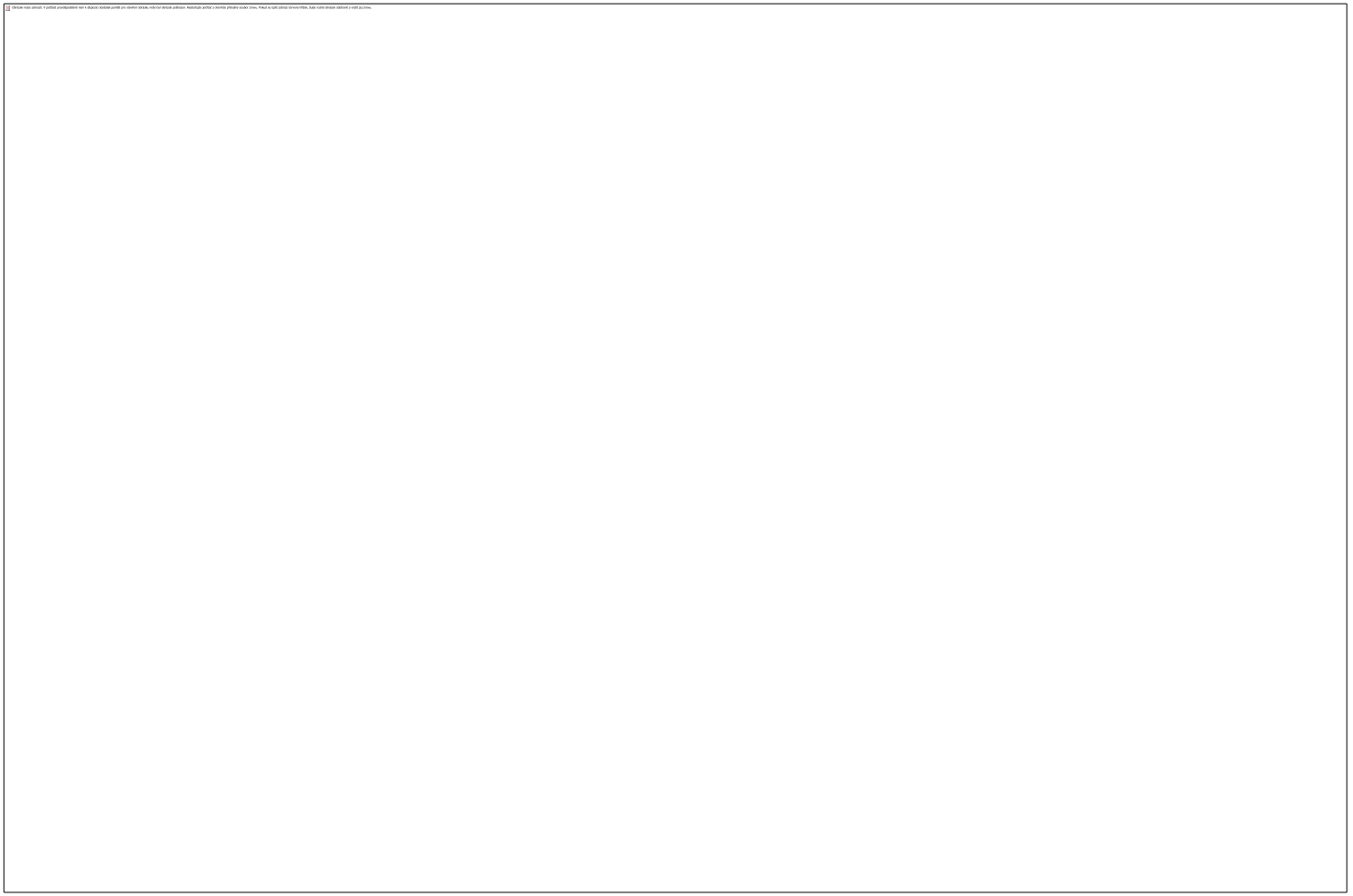




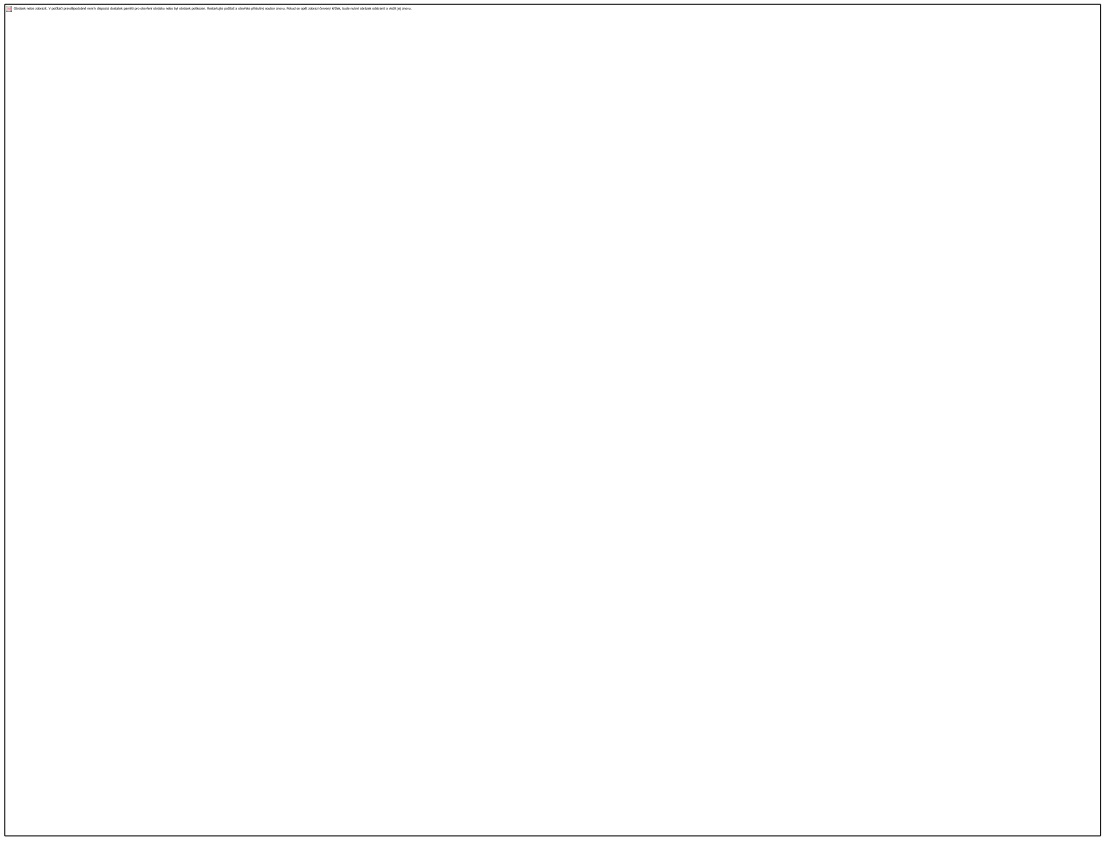
Hádanka: odpoveď



Mapovanie dedičnosti: analytický model

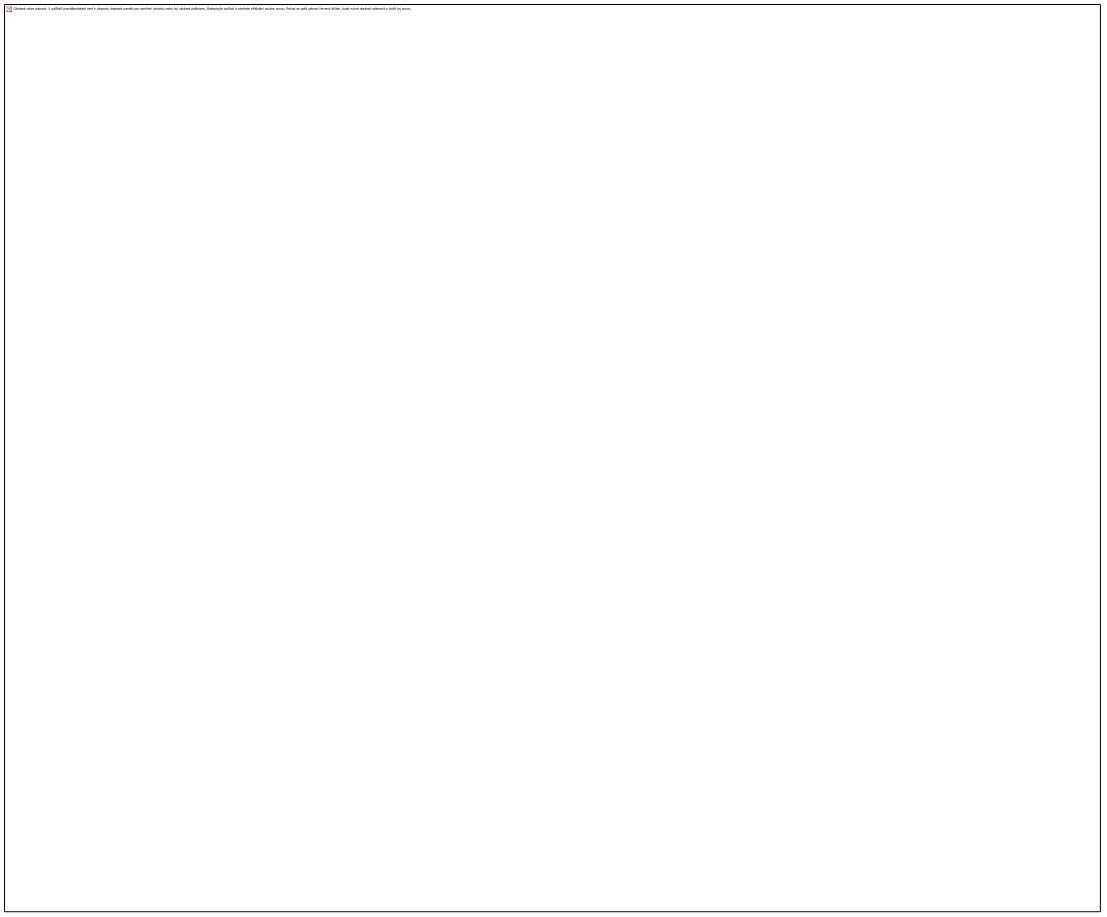


Mapovanie dedičnosti do RDB podľa vzoru „Čisté mapovanie 1:1“ I.



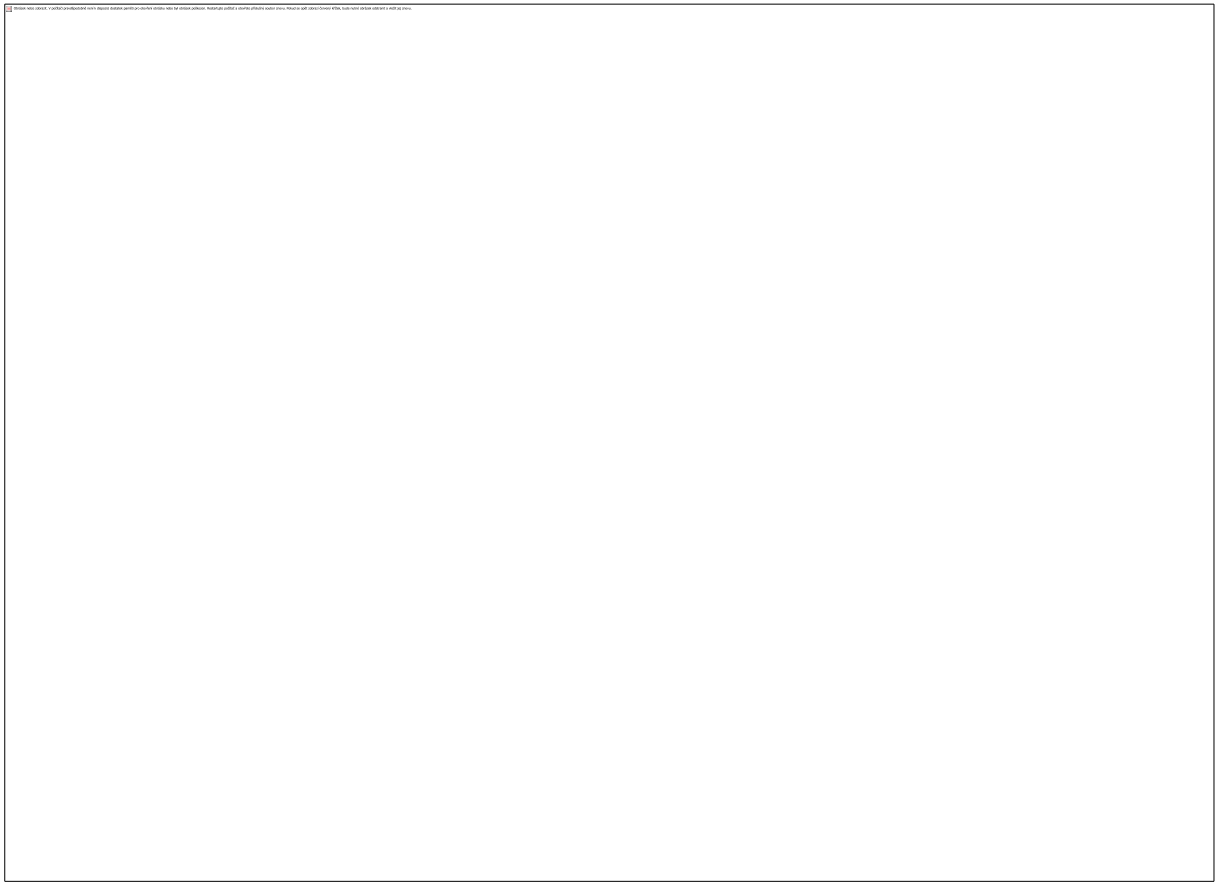
- Každej analytickej triede bude zodpovedať jedna tabuľka
- Tabuľky môžu mať rovnaký primárny kľúč

Mapovanie dedičnosti do RDB podľa vzoru „Čisté mapovanie 1:1“ II.

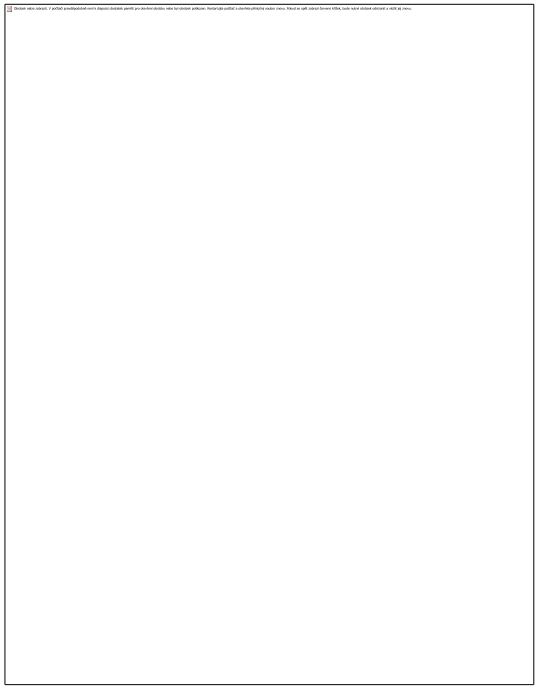


Primárny kľúč tabuľky Osoba sa stal cudzím kľúčom v tabuľkách Student a Ucitel

Mapovanie dedičnosti do RDB podľa vzoru „Čisté mapovanie 1:1“

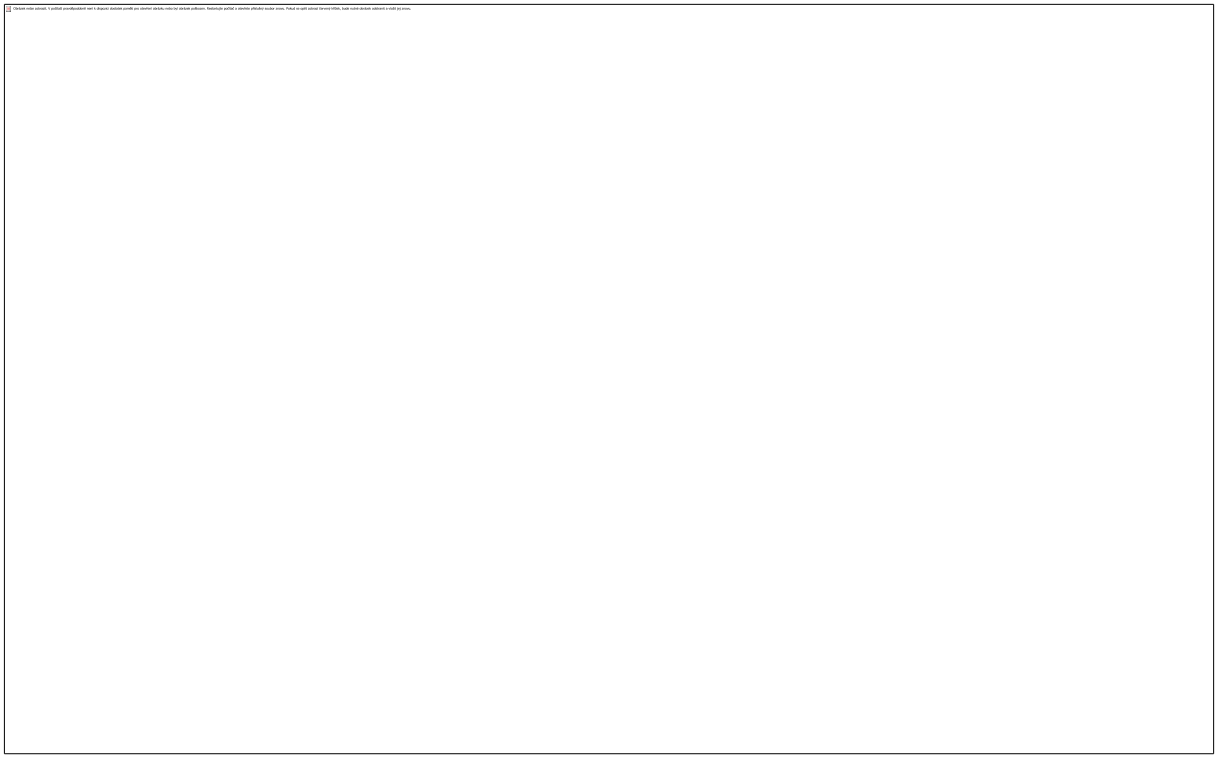


Mapovanie dedičnosti do RDB podľa vzoru „kočkopes“

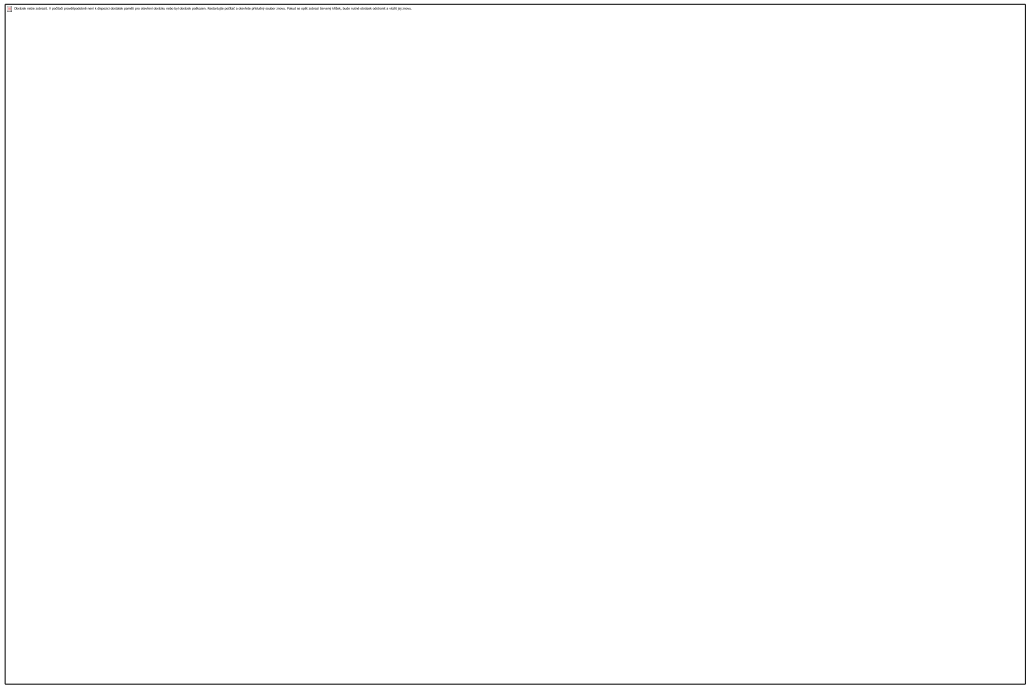


- Údaje zo všetkých tried sa zlúčia do jednej tabuľky
- Riadok tejto tabuľky môže reprezentovať ako učiteľa tak študenta

Mapovanie dedičnosti do RDB podľa vzoru „kočkopes“

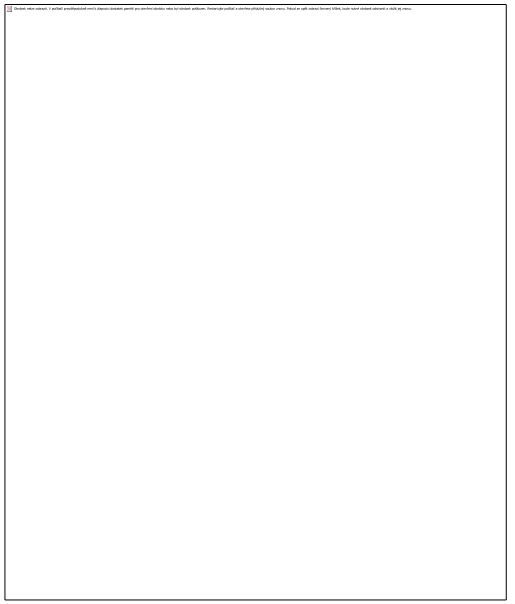


Možný problém vzoru „kočkopes“



Čo znamená takýto riadok tabuľky?

Mapovanie dedičnosti do RDB podľa antivzoru „preťaženie stĺpcov“



- Pre študenta bude Stlpec1 obsahovať ročník a Stlpec2 názov diplomovej práce
- Pre učiteľa bude Stlpec1 obsahovať platovú tried a Stlpec2 názov katedry

Problémy spojené s týmto antivzorom?

???

Observer

Alebo

Don ´t miss out when something interesting happens

Úloha: Meteorologická stanica (weather station)

Máme WeatherDataObject, ktorý vracia:

- Teplotu
- Vlhkosť vzduchu
- Tlak vzduchu

Úlohou je navrhnúť aplikáciu, ktorá bude informovať o počasí pomocou (zatiaľ) troch rôznych užívateľských rozhraní:

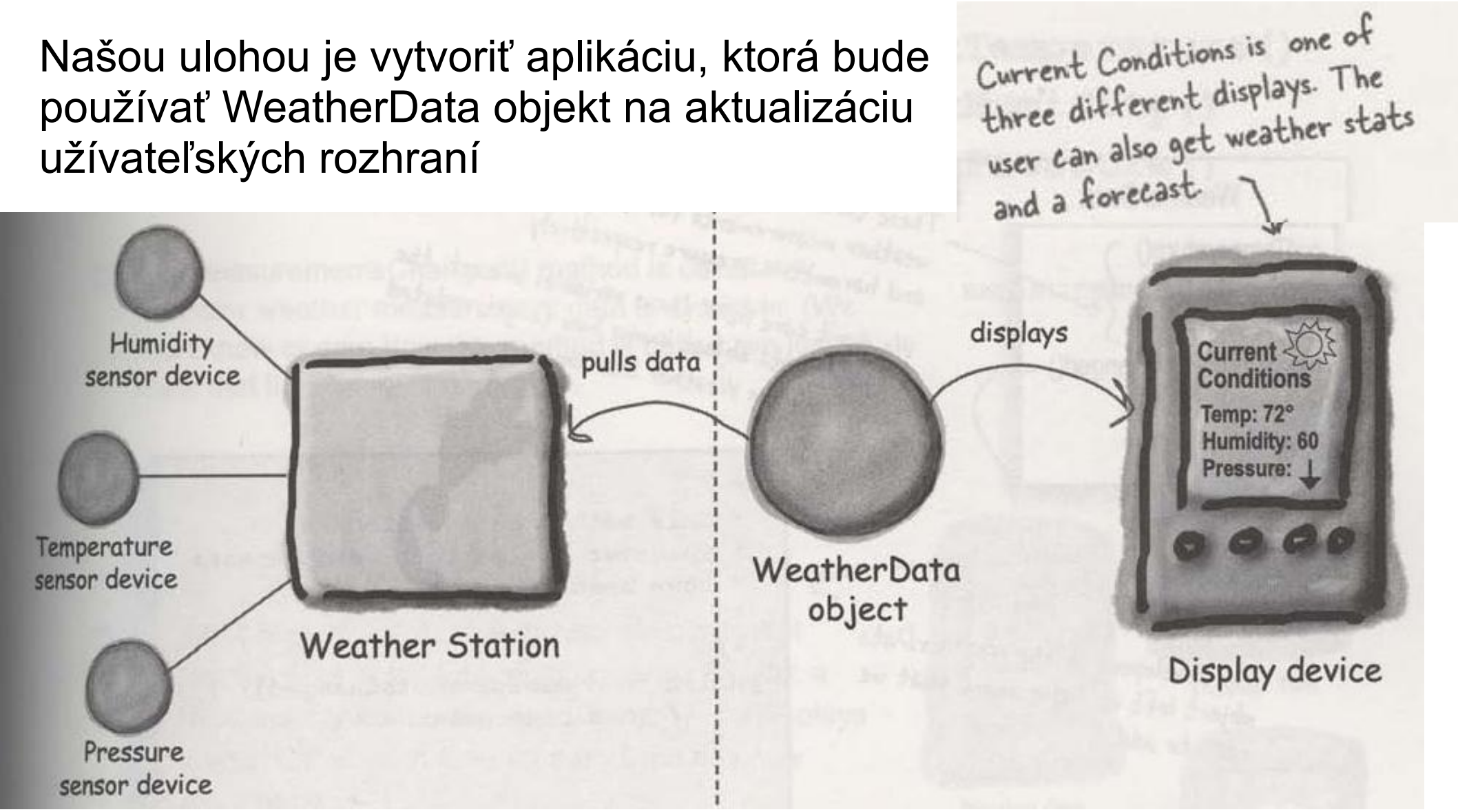
1. Aktuálne počasie
2. Štatistika o počasí za uplynulé obdobie
3. Predpoveď počasia

Úloha: pokračovanie

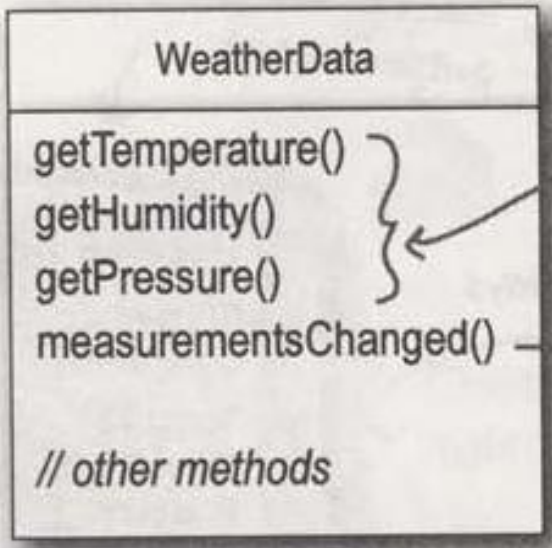
Navrhnite tiež API, ktoré by zákazníci mohli použiť na vytváranie vlastných užívateľských rozhraní

Meteorologická stanica: prehľad

Našou ulohou je vytvoriť aplikáciu, ktorá bude používať WeatherData objekt na aktualizáciu užívateľských rozhraní



WeatherData class



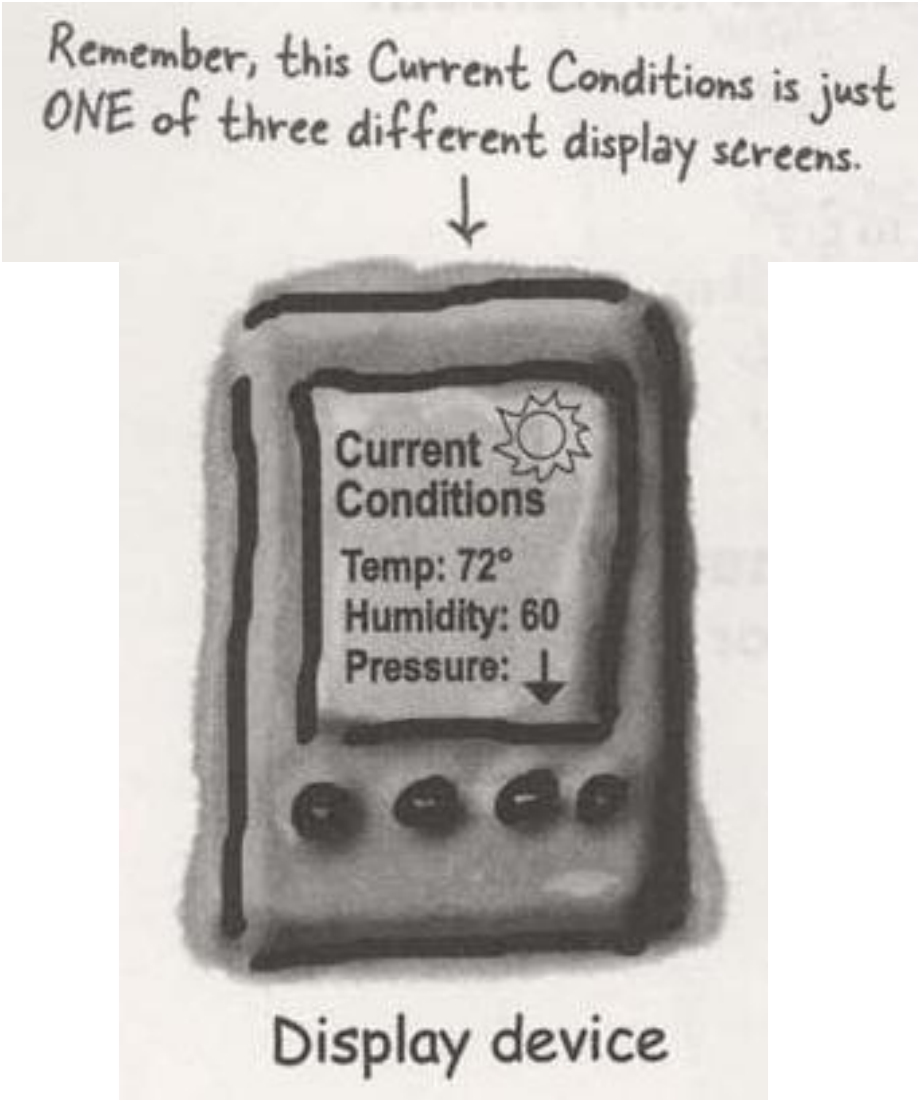
These three methods return the most recent weather measurements for temperature, humidity and barometric pressure respectively.

We don't care HOW these variables are set; the WeatherData object knows how to get updated info from the Weather Station.

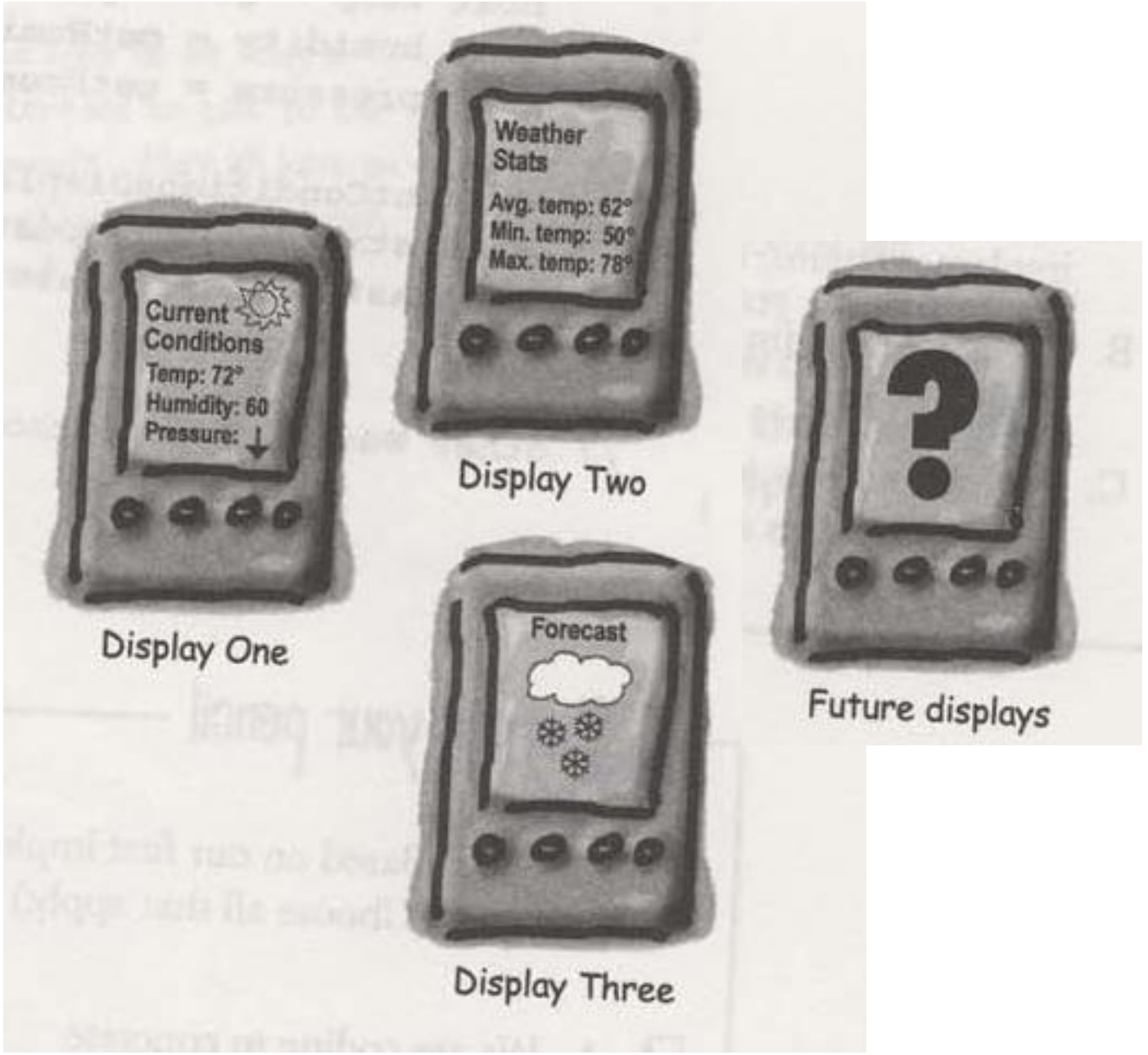
The developers of the WeatherData object left us a clue about what we need to add...

```
/*
 * This method gets called
 * whenever the weather measurements
 * have been updated
 *
 */
public void measurementsChanged() {
    // Your code goes here
}
```

WeatherData.java



Úlohou je teda navrhnúť a naprogramovať metódu MeasurementsChanged()


Current
Conditions
Temp: 72°
Humidity: 60
Pressure:
Display One
Weather
Stats
Avg. temp: 62°
Min. temp: 50°
Max. temp: 78°
Display Two
Forecast
Display Three
Future displays

Jedno z možných riešení:

```
public class WeatherData {

    // instance variable declarations

    public void measurementsChanged() {

        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

        currentConditionsDisplay.update(temp, humidity, pressure);
        statisticsDisplay.update(temp, humidity, pressure);
        forecastDisplay.update(temp, humidity, pressure);
    }

    // other WeatherData methods here
}
```

Grab the most recent measuremets by calling the WeatherData's getter methods (already implemented).

Now update the displays...

Call each display element to update its display, passing it the most recent measurements.

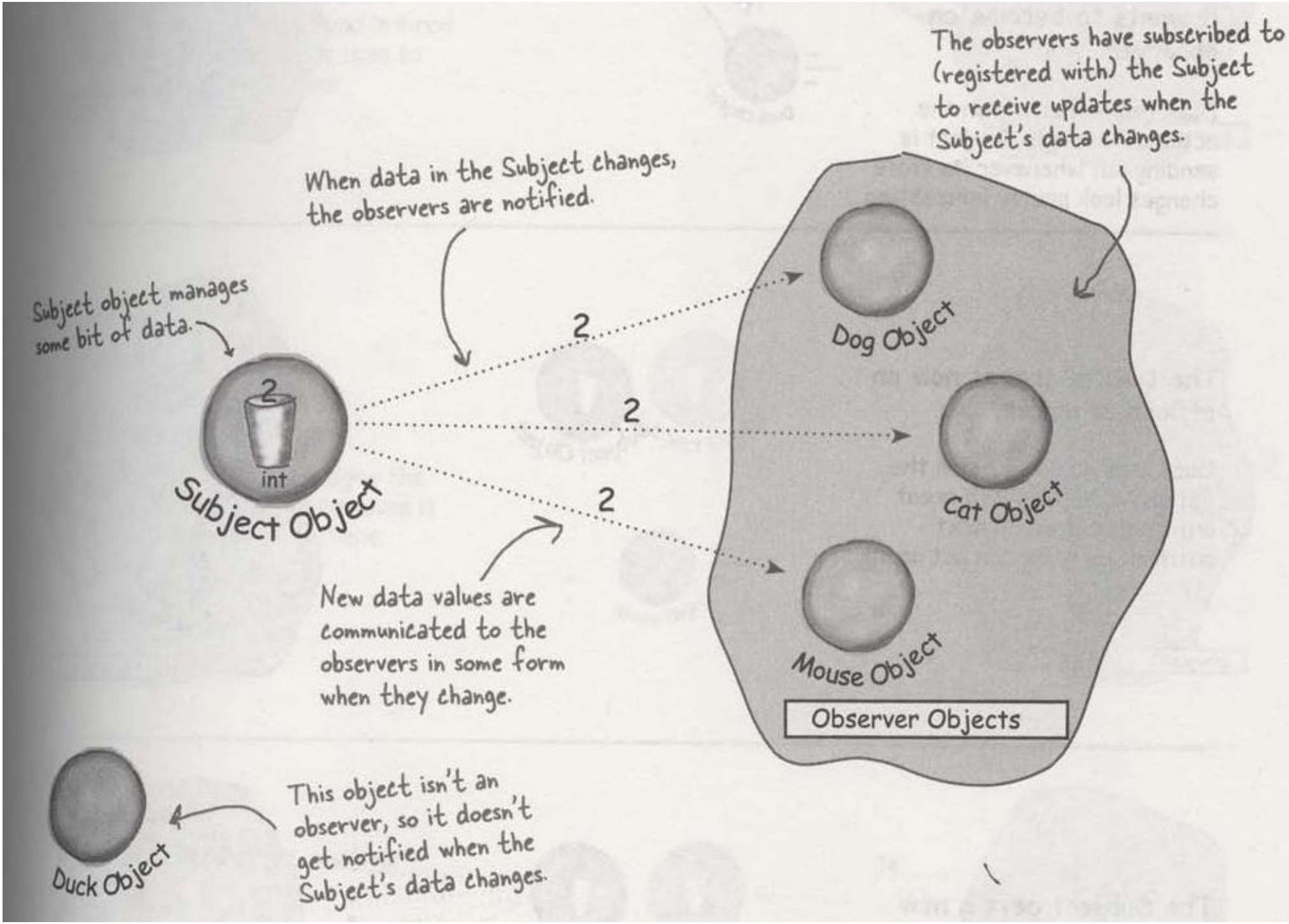
Čo si o tomto riešení myslíte Vy?

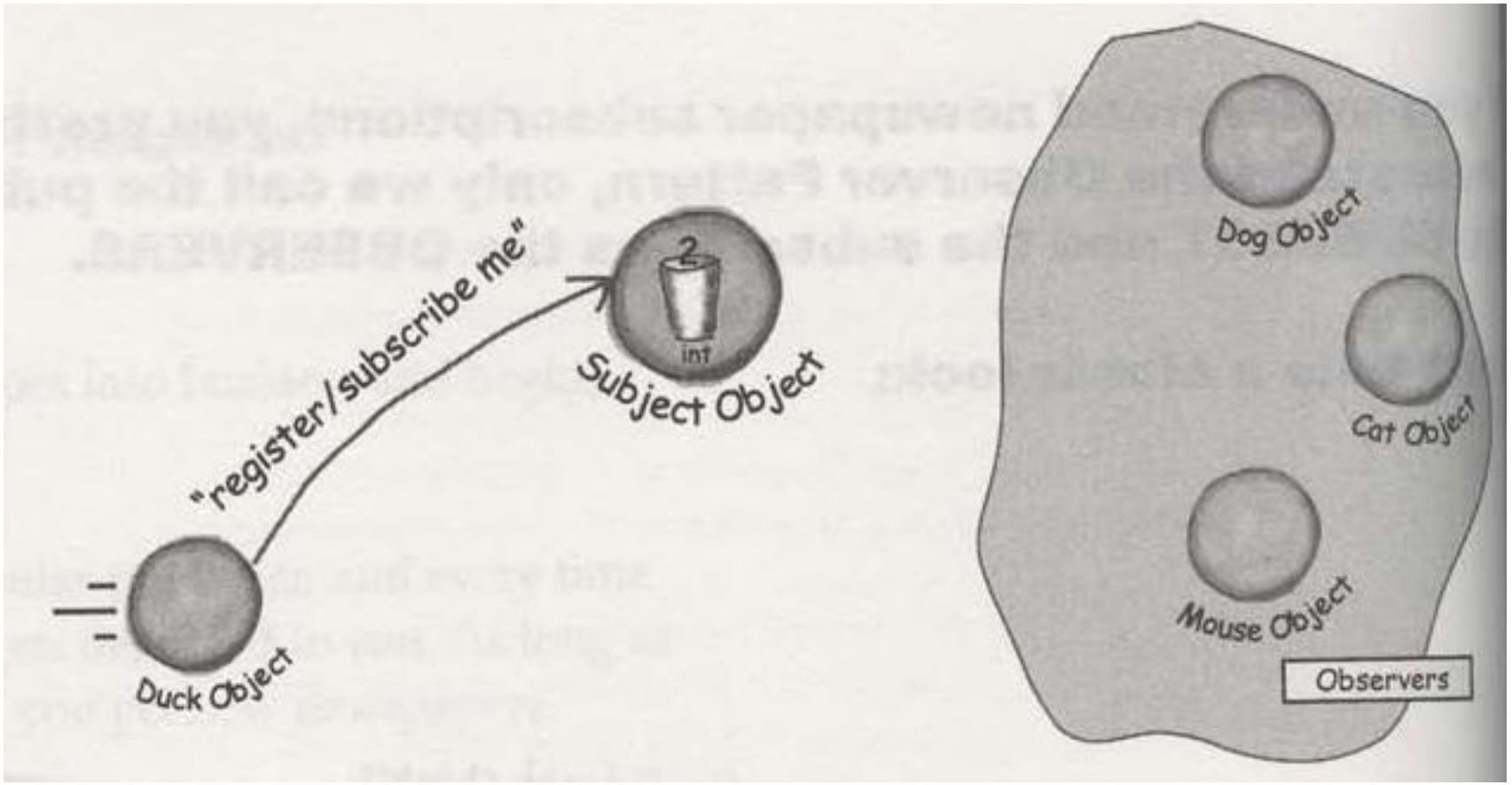
Nevýhody predchádzajúceho riešenia

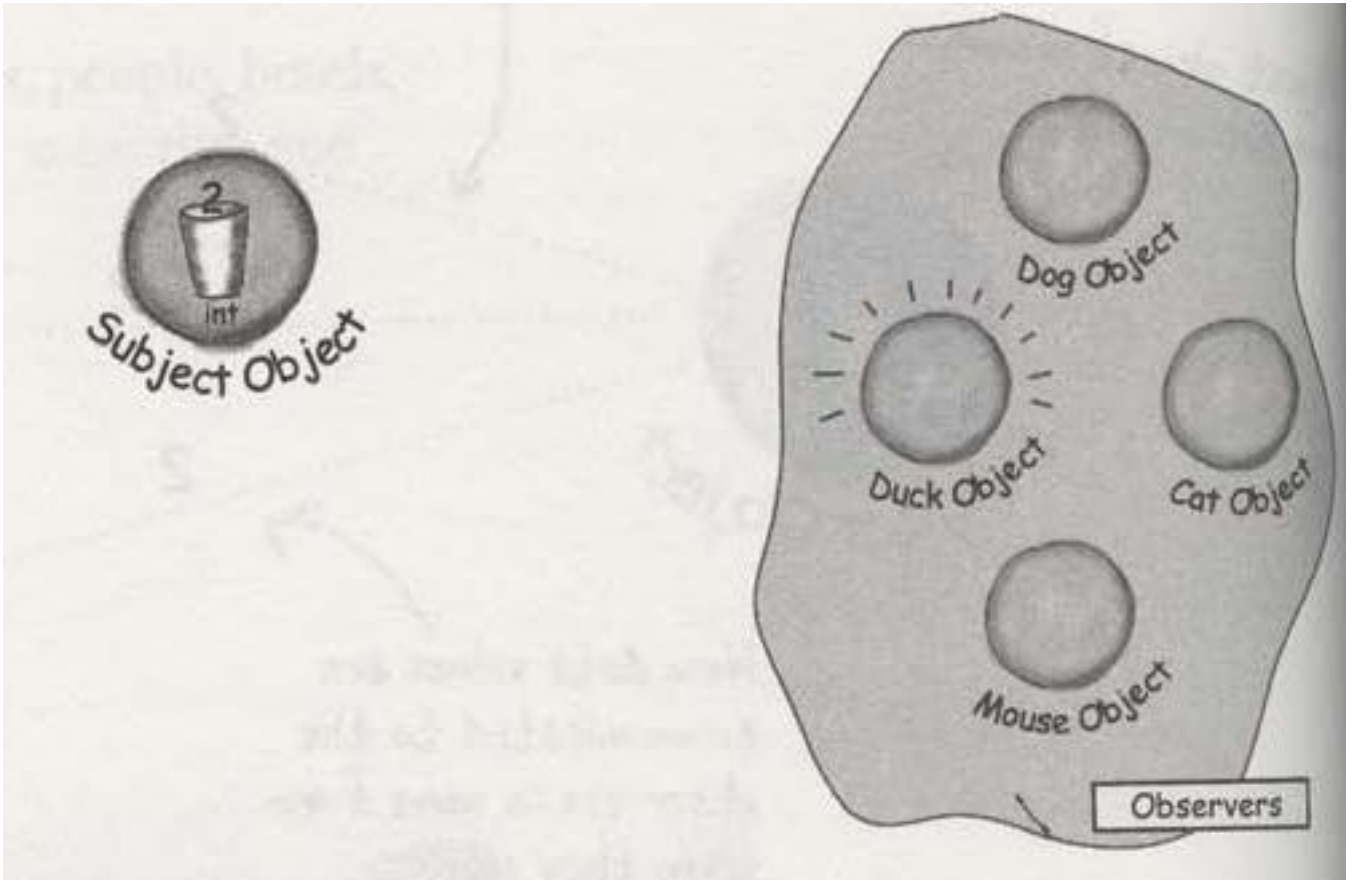
- Metóda MeasurementsChanged volá priamo metódy konkrétnych tried (zvýšené zaťaženie)
- Ak by sme chceli pridať nový display, museli by sme zasahovať do kódu MeasuremantsChanged (a znova ju kompilovať!)
- Nemáme možnosť pridať nový display počas behu aplikácie

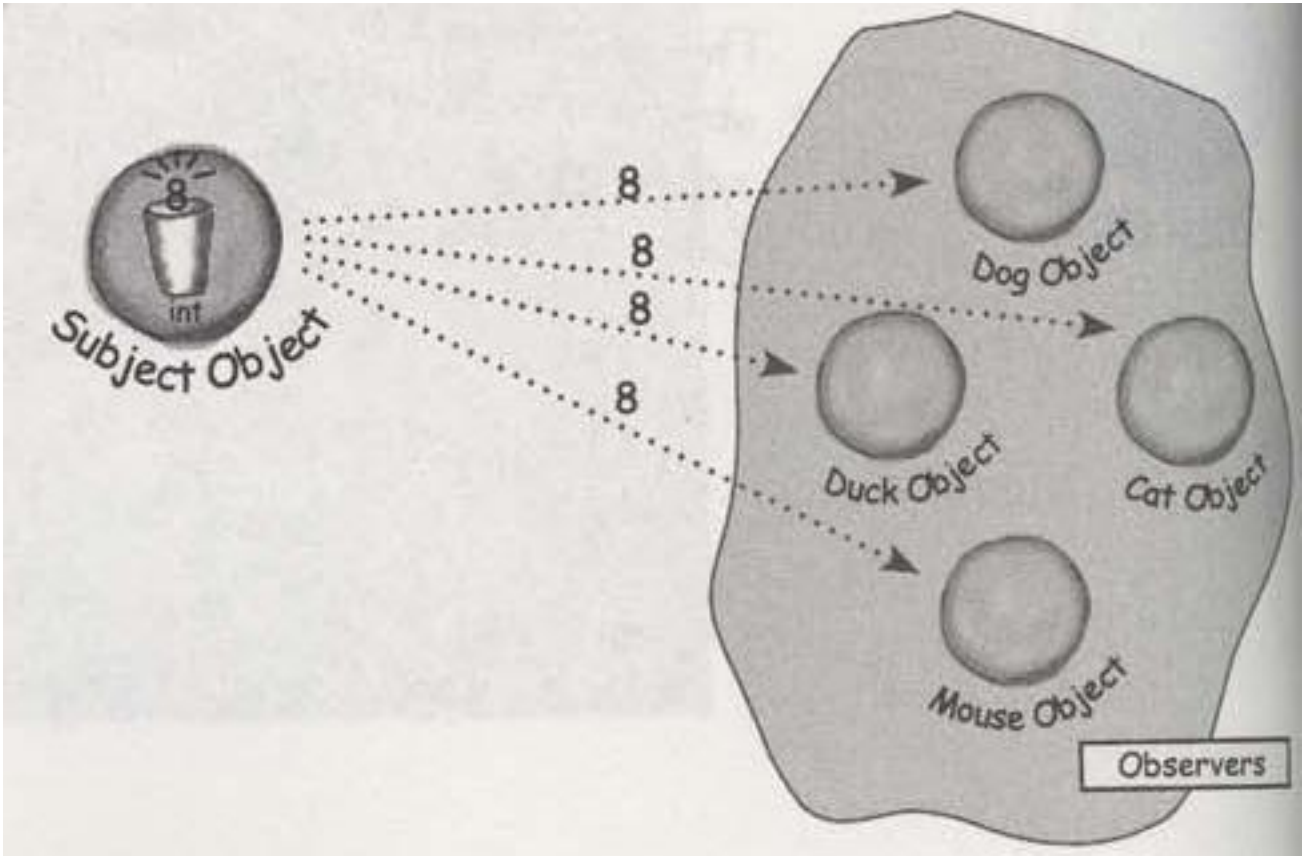
Ako funguje Observer? Príklad

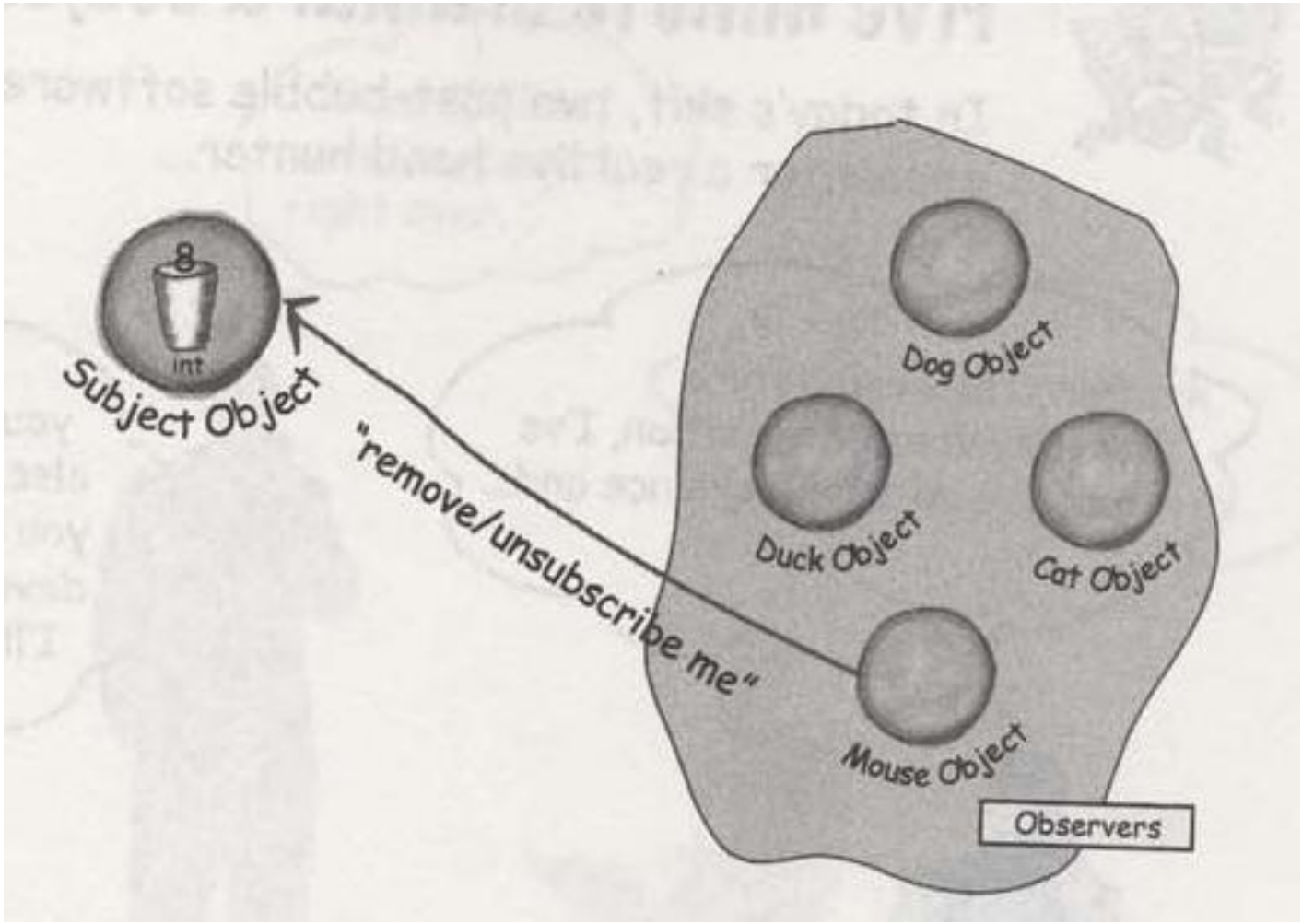
- Vydavateľ začne vydávať časopis
- Zákazník sa prihlási k odberu (zaplatí si predplatné)
- Vždy, keď vydavateľ vydá nové číslo, zašle ho automaticky všetkým predplatiteľom
- Keď zákazník nechce ďalej časopis odoberať, odhlási sa. Vydavateľ ho vyradí zo zoznamu predplatiteľov a prestane mu časopis zasielať
- Ostatní predplatitelia časopis dostávajú aj naďalej

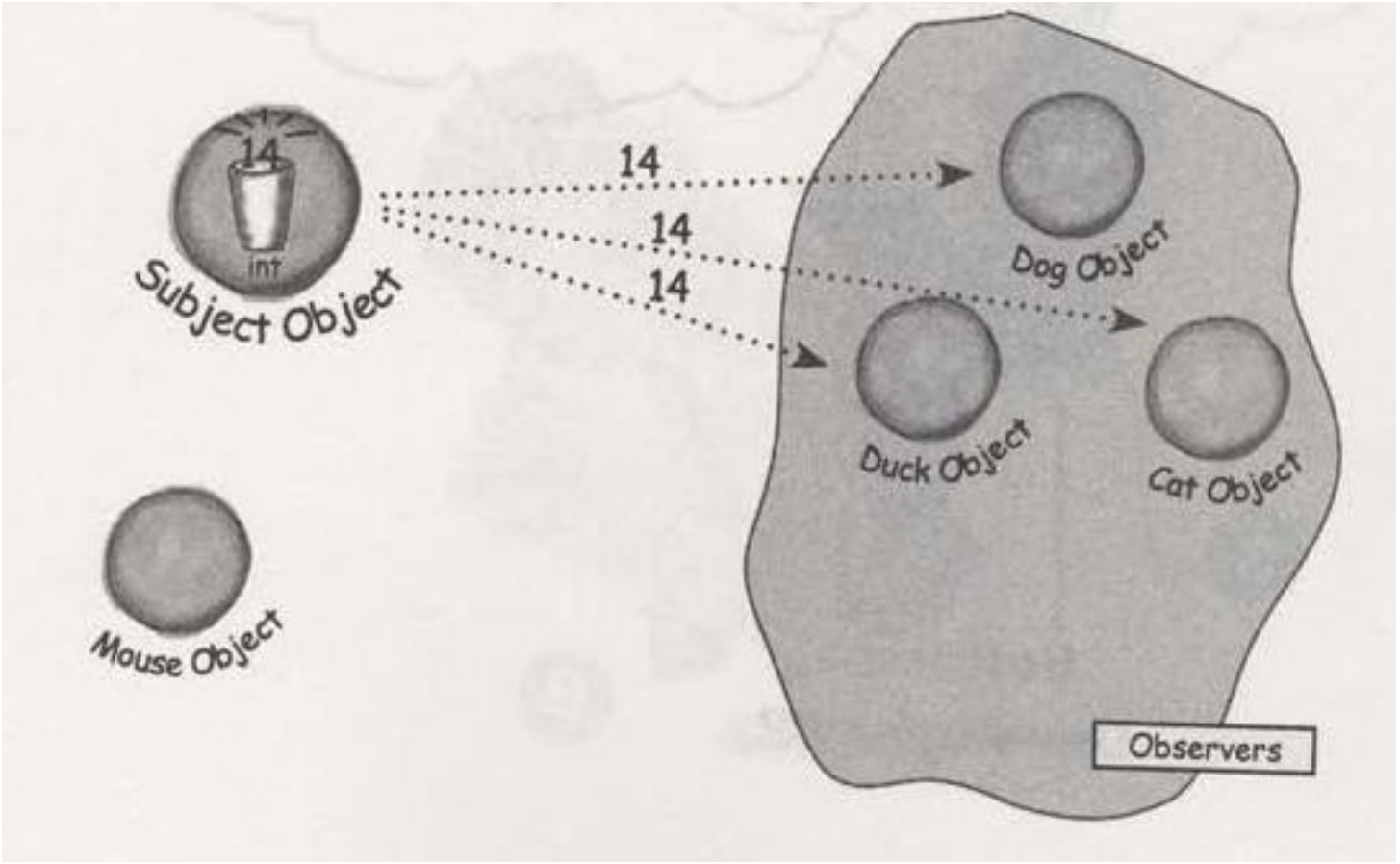

The observers have subscribed to (registered with) the Subject to receive updates when the Subject's data changes.
When data in the Subject changes, the observers are notified.
Subject object manages some bit of data.
2
int
Subject Object
2
2
2
Dog Object
Cat Object
Mouse Object
Observer Objects
New data values are communicated to the observers in some form when they change.
Duck Object
This object isn't an observer, so it doesn't get notified when the Subject's data changes.


"register/subscribe me"
2
int
Subject Object
Duck Object
Dog Object
Cat Object
Mouse Object
Observers


2
int
Subject Object
Dog Object
Duck Object
Cat Object
Mouse Object
Observers

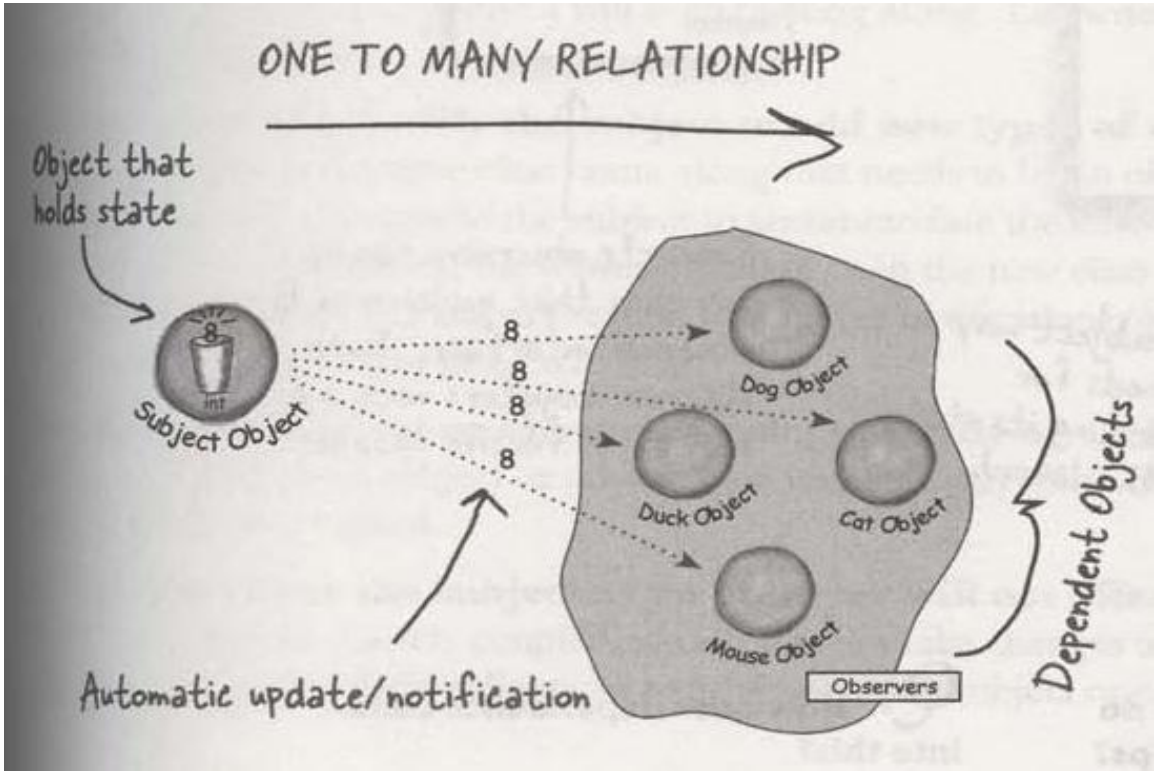

8
int
Subject Object
8
8
8
8
Dog Object
Duck Object
Cat Object
Mouse Object
Observers


8
int
Subject Object
"remove/unsubscribe me"
Dog Object
Duck Object
Cat Object
Mouse Object
Observers

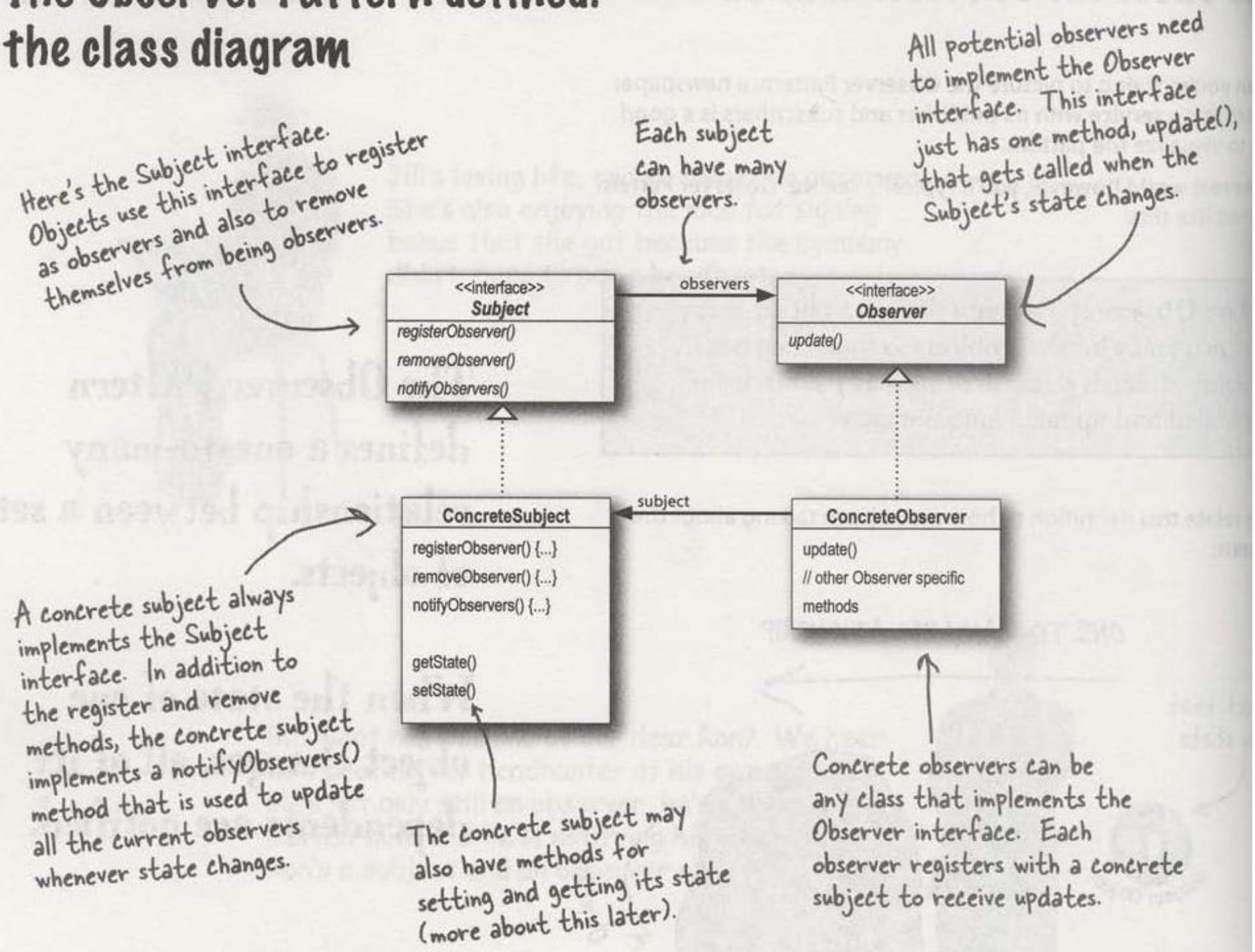

14
int
Subject Object
14
14
14
Dog Object
Duck Object
Cat Object
Mouse Object
Observers

Vzor Observer: Definícia

Vzor Observer definuje závislosť s kardinalitou 1:N medzi objektmi. Pre túto závislosť platí, že ak objekt na strane 1 zmení svoj stav, sú o tom informované všetky objekty na strane N.



The Observer Pattern defined: the class diagram



Observer ako implementácia princípu Loose coupling

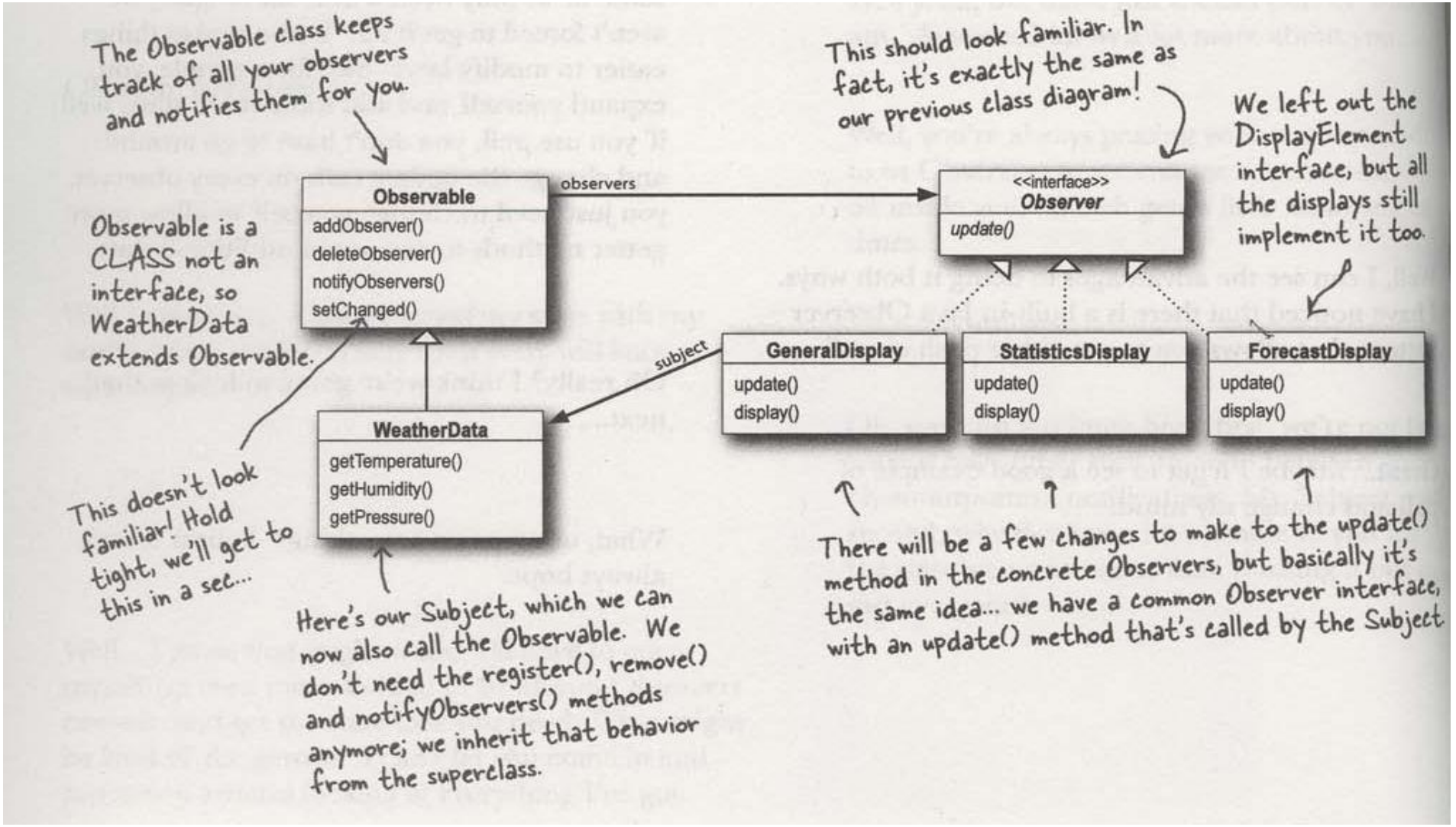
- Jediné, čo subject vie o observeroch je to, že implementujú nejaké konkrétne rozhranie
- Kedykoľvek máme možnosť pridať ďalšieho observera (bez zásahu do kódu subjectu)
- Subjekty a observery môžeme znovu použiť (reuse) nezávisle od seba (nízke zaťaženie)
- Zmeny subjektu alebo observera zostávajú izolované (nemajú dôsledky pre iné objekty)
- Rozšírenie aplikácie o ďalšieho observera je veľmi jednoduché

Návrh WeatherStation

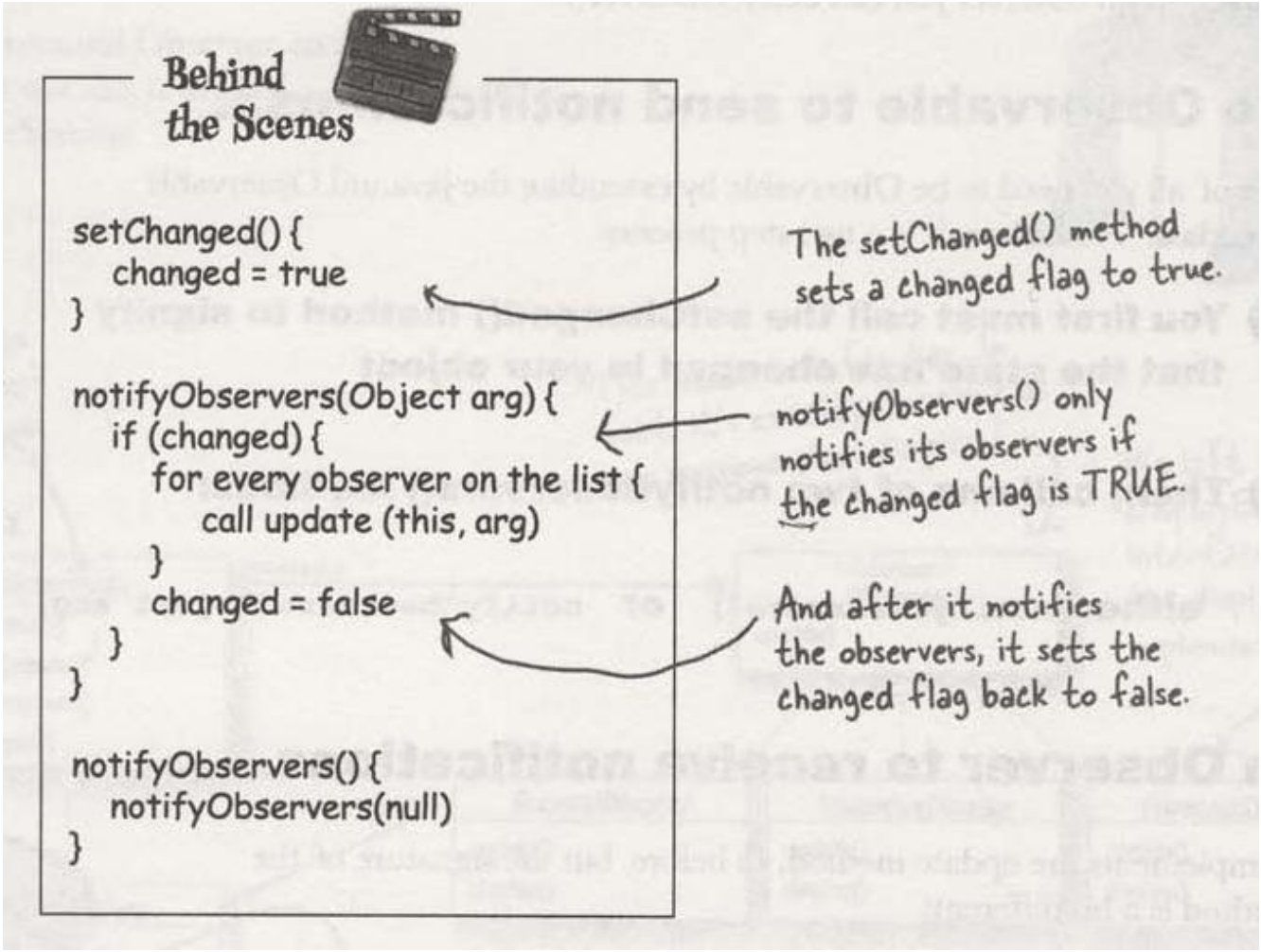
Použité princípy návrhu

- Loose coupling
- Oddelenie kódu podliehajúceho zmenám od statického kódu

Vstavaná podpora vzoru Observer v prostredí Java



SetChanged


Behind the Scenes
setChanged() {
changed = true
}
notifyObservers(Object arg) {
if (changed) {
for every observer on the list {
call update (this, arg)
}
changed = false
}
}
notifyObservers() {
notifyObservers(null)
}
The setChanged() method sets a changed flag to true.
notifyObservers() only notifies its observers if the changed flag is TRUE.
And after it notifies the observers, it sets the changed flag back to false.

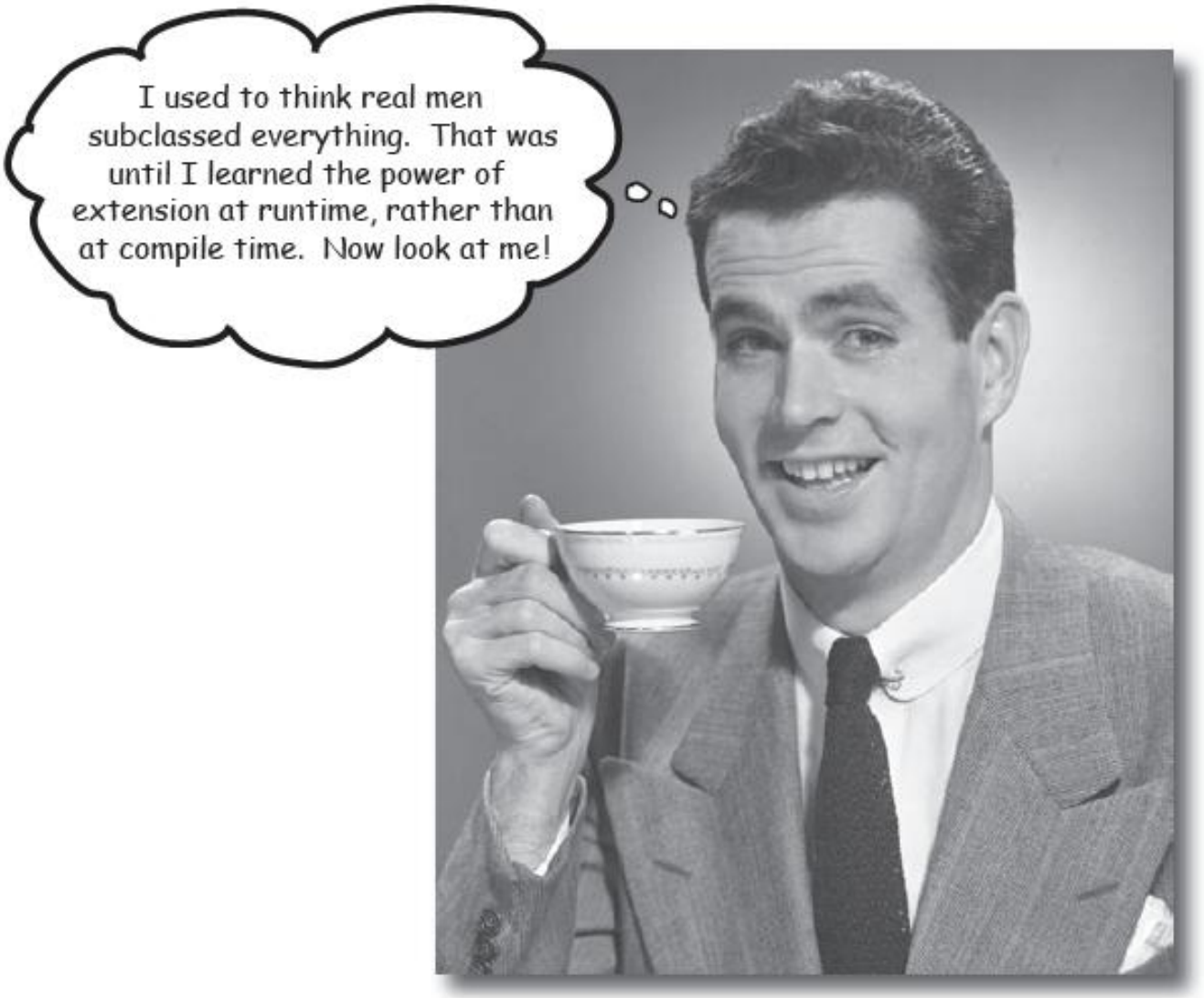
Príklad knižnica

- Prípad užitia: zasielanie noviniek
- Knihovník môže do systému zadať novinky
- Novinky sa zobrazia:

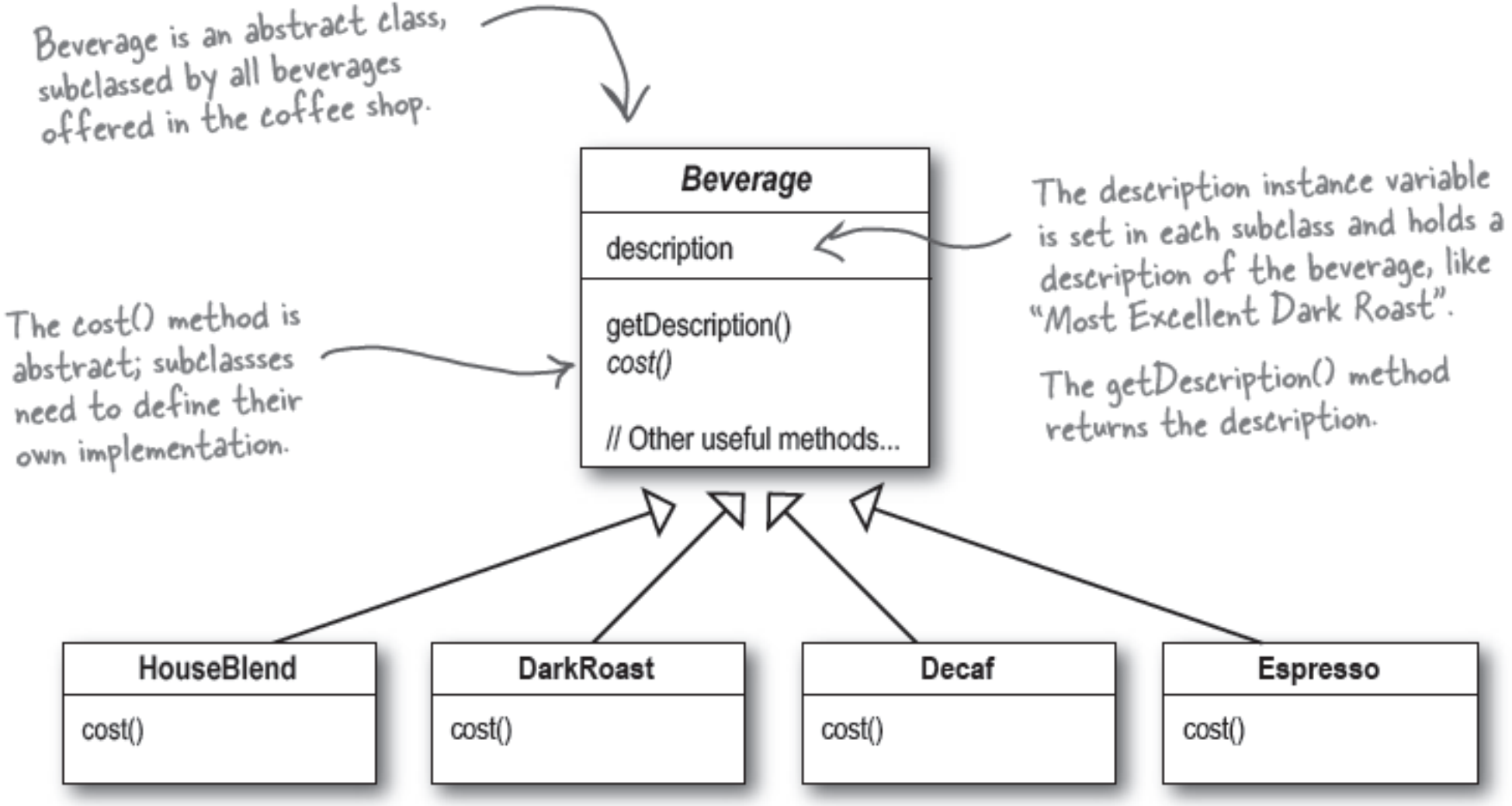
–užívateľom, ktorí sú práve v knižnici a listujú v katalógu knižnice (sú on-line)

–užívateľom, ktorí sa prihlásili k odoberaniu noviniek prostredníctvom emailu

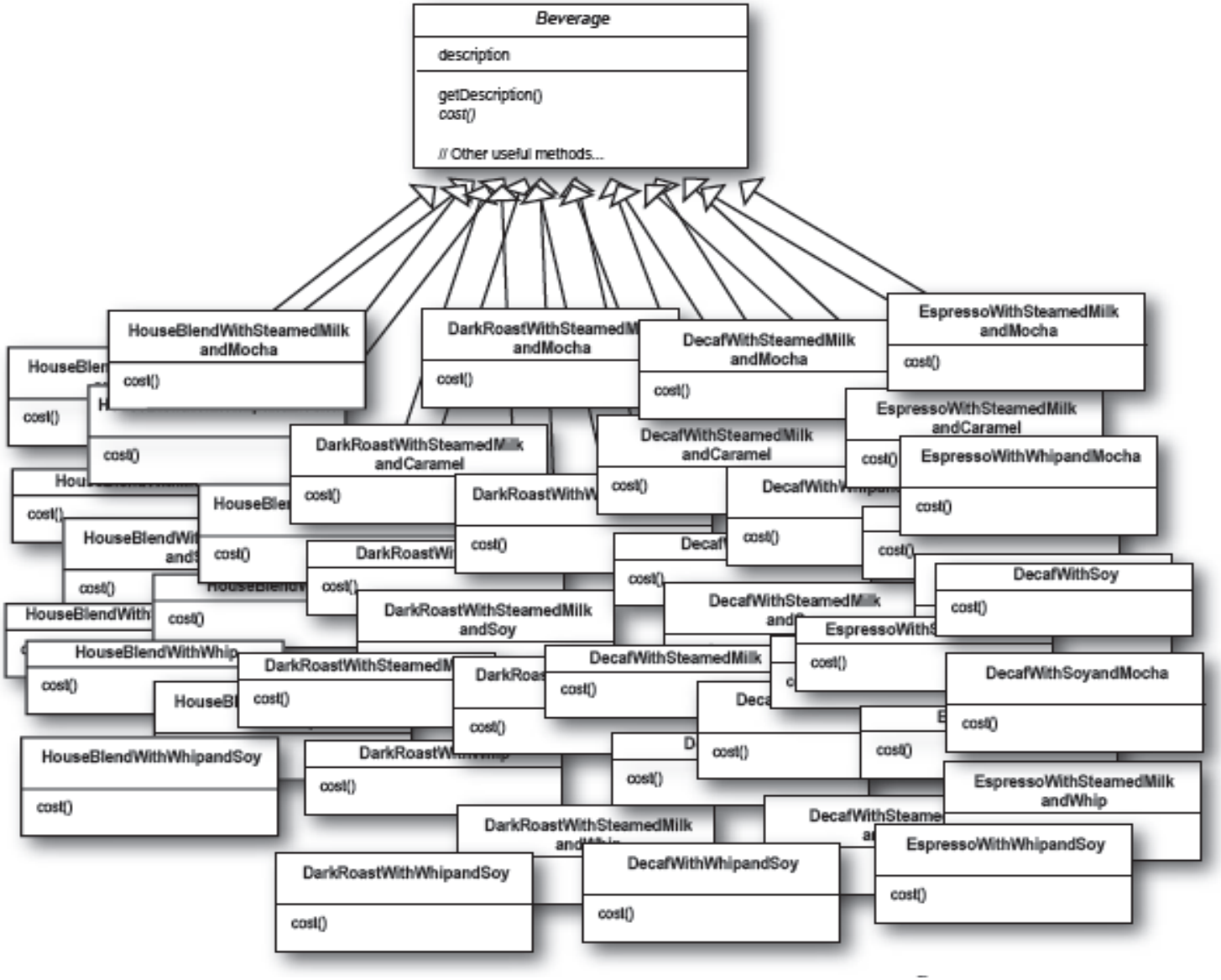
Decorato


I used to think real men subclassed everything. That was until I learned the power of extension at runtime, rather than at compile time. Now look at me!

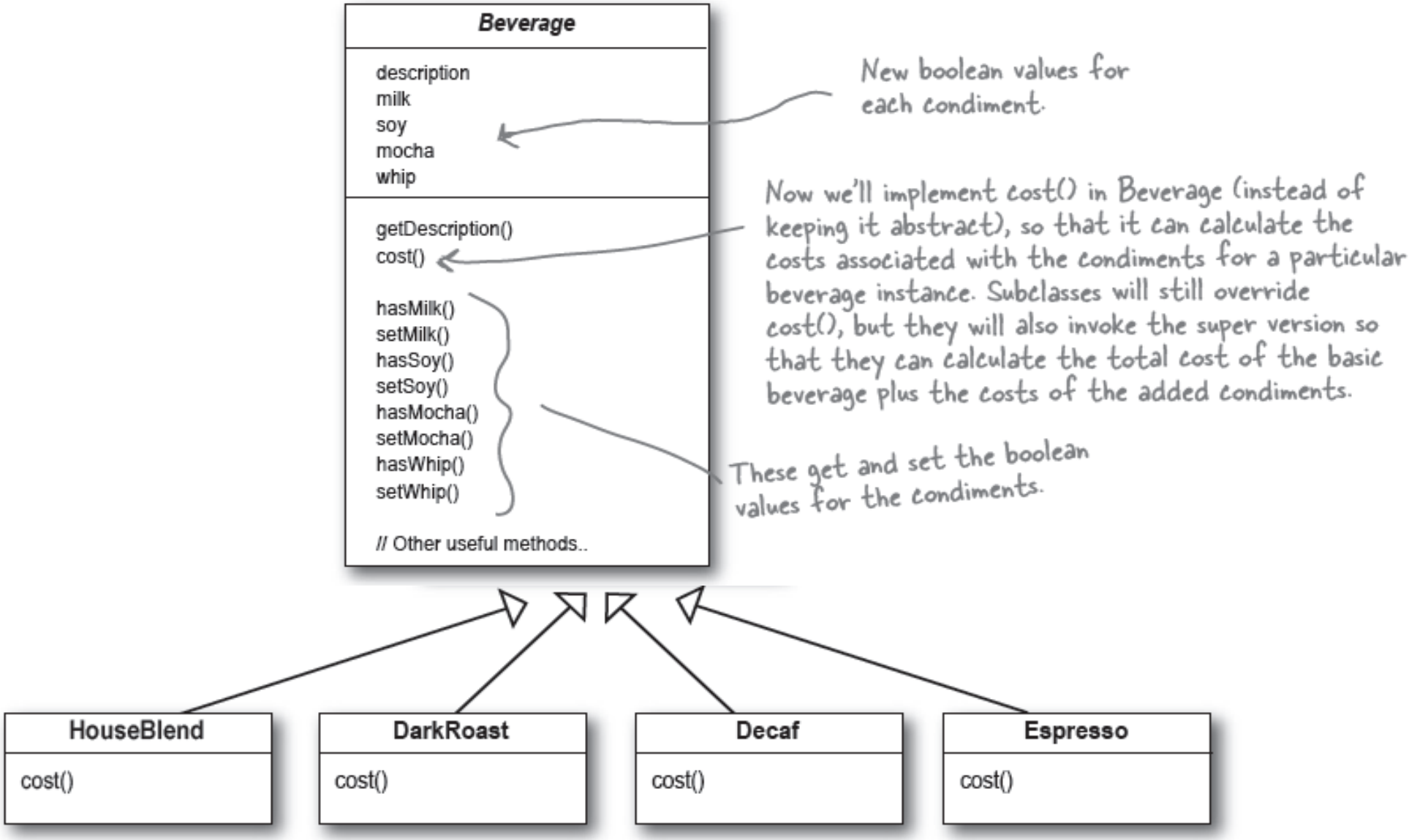
Príklad: Starbuzz Coffie



Koľko stojí káva s cukrom a mliekom?
Riešenie 1.



O niečo lepšie riešenie



Skúste si to!

Napíšte (stačí pseudokód) metódu cost() pre triedu Beverage a pre triedu DarkRoast.

Aké problémy môžu vzniknúť v súvislosti s týmto riešením?

Open-closed princíp

Triedy by mali byť zatvorené pokiaľ ide o zmeny, avšak otvorené, pokiaľ ide o rozšírenie

Koľko stojí káva s cukrom a mliekom?

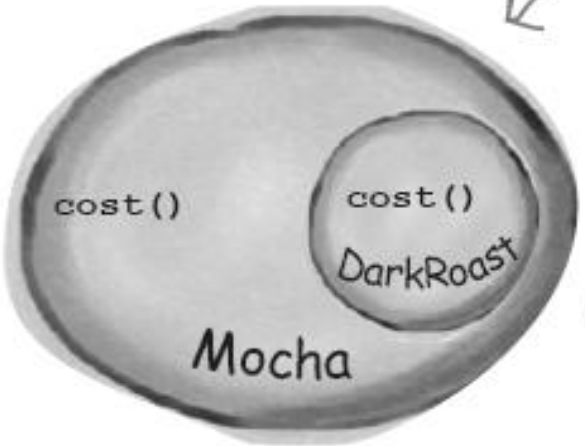
Resp. DarkRoast with Mocha and Whip?

1. **Take a DarkRoast object**
2. **Decorate it with a Mocha object**
3. **Decorate it with a Whip object**
4. **Call the cost() method and rely on delegation to add on the condiment costs**

❶ We start with our DarkRoast object.

Remember that DarkRoast inherits from Beverage and has a cost() method that computes the cost of the drink.

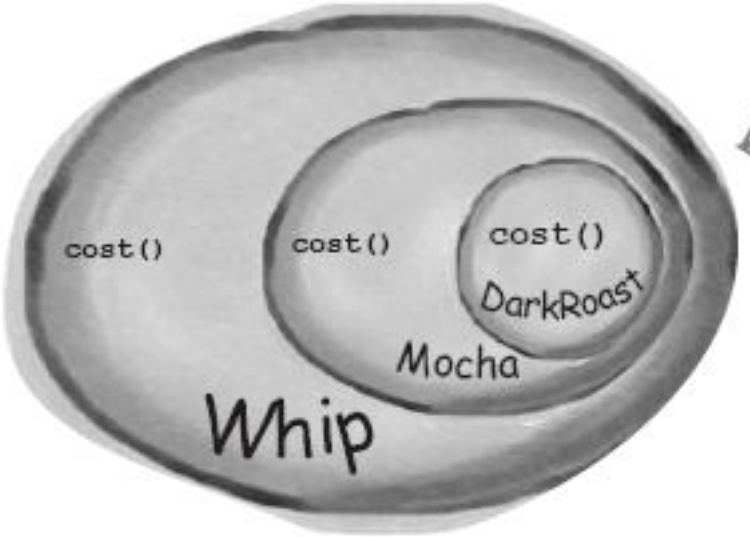
❷ The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.



The Mocha object is a decorator. Its type mirrors the object it is decorating, in this case, a Beverage. (By "mirror", we mean it is the same type..)

So, Mocha has a cost() method too, and through polymorphism we can treat any Beverage wrapped in Mocha as a Beverage, too (because Mocha is a subtype of Beverage).

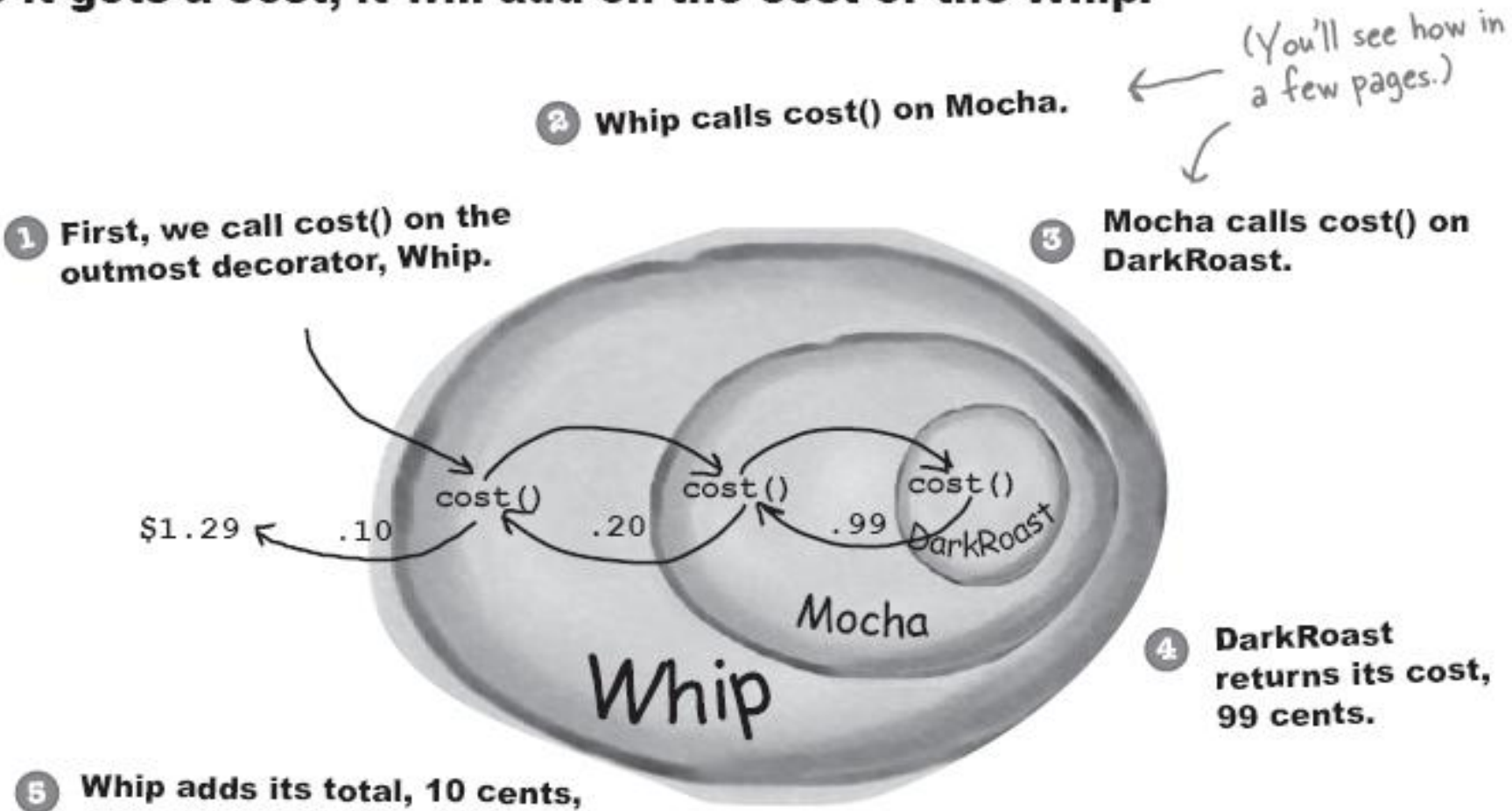
❸ The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.



Whip is a decorator, so it also mirrors DarkRoast's type and includes a cost() method.

So, a DarkRoast wrapped in Mocha and Whip is still a Beverage and we can do anything with it we can do with a DarkRoast, including call its cost() method.

4 **Now it's time to compute the cost for the customer. We do this by calling cost() on the outermost decorator, Whip, and Whip is going to delegate computing the cost to the objects it decorates. Once it gets a cost, it will add on the cost of the Whip.**



Vlastnosti dekorátorov

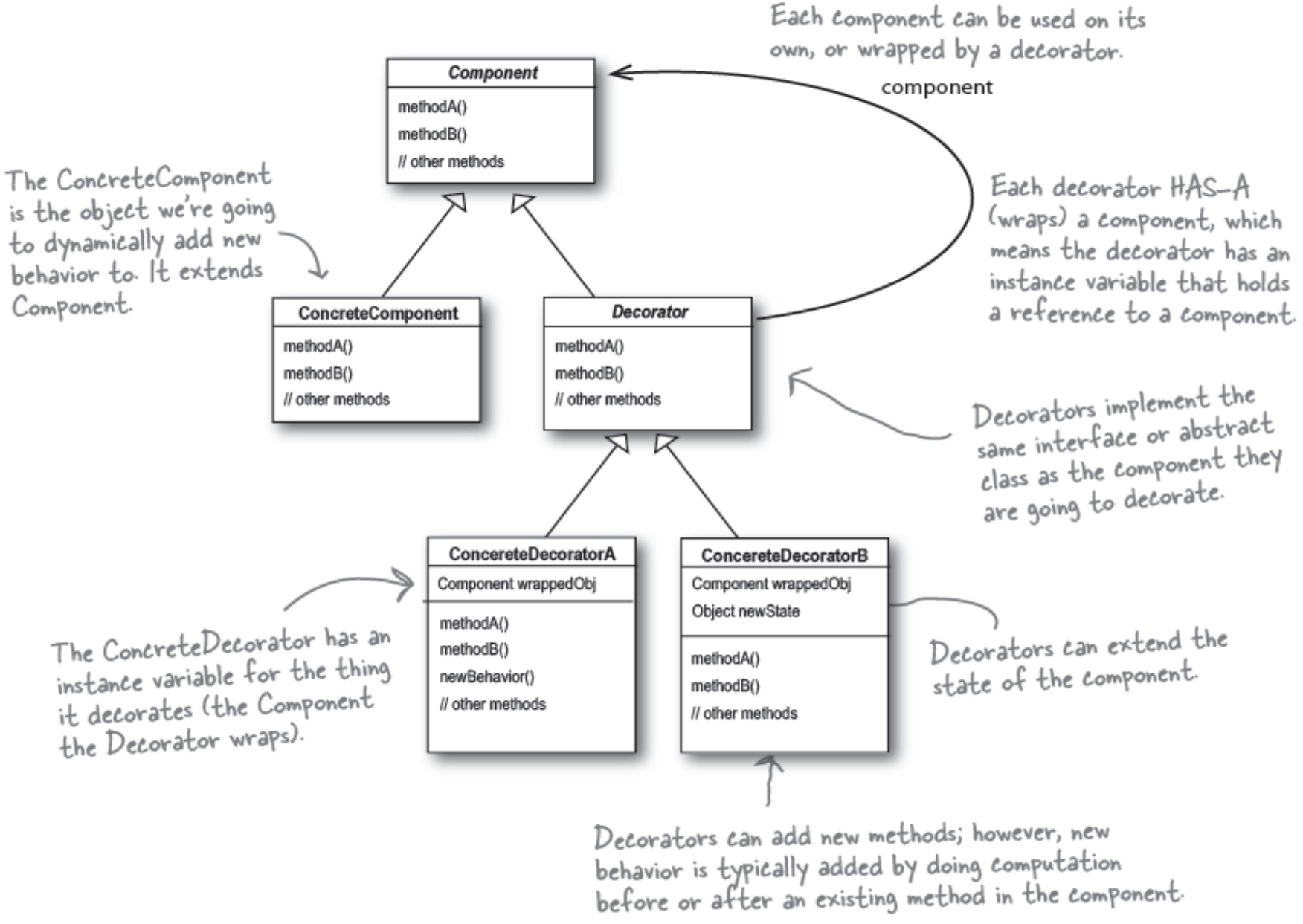
- Dekorátory majú spoločný nadtyp s triedami, ktoré dekorujú (vďaka tomu môžeme predávať dekorovaný objekt namiesto pôvodného)
- Jeden objekt môže byť dekorovaný jedným alebo viacerými dekorátormi
- Dekorátor môže pridávať svoje správanie predtým alebo potom než deleguje riadenie dekorovanému objektu
- Objekty môžu byť dekorované kedykoľvek počas behu programu

Vzor Dekorátor: definícia

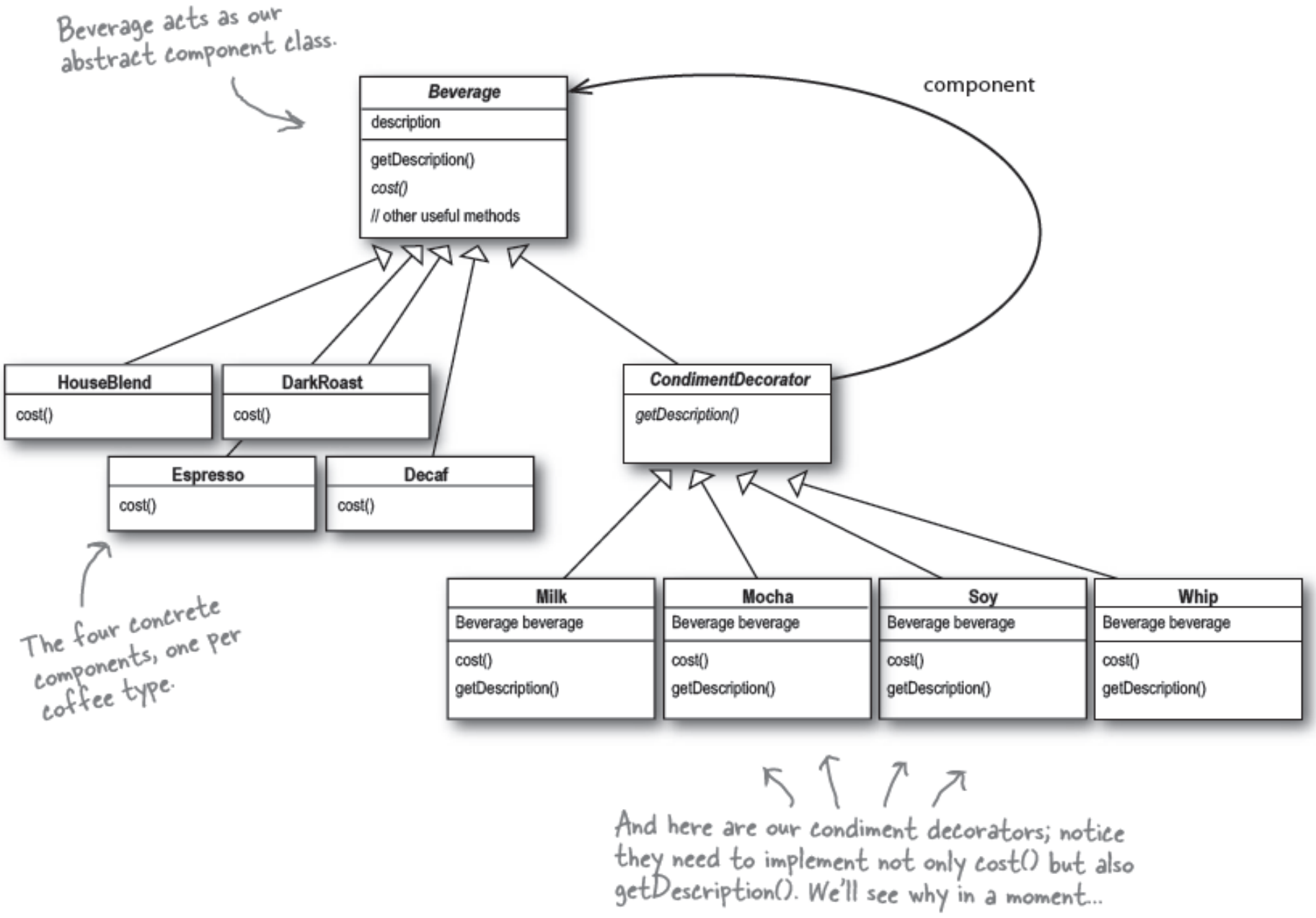
Vzor dekorátor dynamicky pridáva správanie a zodpovednosti iným objektom.

Dekorátory sú flexibilnou alternatívou dedičnosti

Vzor dekorátor: schéma



Káva s cukrom a mliekom



```
public abstract class Beverage {
    String description = "Unknown Beverage";

    public String getDescription() {
        return description;
    }

    public abstract double cost();
}
```

Beverage is an abstract class with the two methods getDescription() and cost().

getDescription is already implemented for us, but we need to implement cost() in the subclasses.

First, we need to be interchangeable with a Beverage, so we extend the Beverage class.

```
public abstract class CondimentDecorator extends Beverage {
    public abstract String getDescription();
}
```

We're also going to require that the condiment decorators all reimplement the getDescription() method. Again, we'll see why in a sec...

First we extend the Beverage class, since this is a beverage.

```
public class Espresso extends Beverage {

    public Espresso() {
        description = "Espresso";
    }

    public double cost() {
        return 1.99;
    }
}
```

To take care of the description, we set this in the constructor for the class. Remember the description instance variable is inherited from Beverage.

Finally, we need to compute the cost of an Espresso. We don't need to worry about adding in condiments in this class, we just need to return the price of an Espresso: $1.99.

First we extend the Beverage class, since this is a beverage.

```
public class Espresso extends Beverage {

    public Espresso() {
        description = "Espresso";
    }

    public double cost() {
        return 1.99;
    }
}
```

To take care of the description, we set this in the constructor for the class. Remember the description instance variable is inherited from Beverage.

Finally, we need to compute the cost of an Espresso. We don't need to worry about adding in condiments in this class, we just need to return the price of an Espresso: $1.99.

Mocha is a decorator, so we extend CondimentDecorator.

Remember, CondimentDecorator extends Beverage.

```
public class Mocha extends CondimentDecorator {
    Beverage beverage;

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }

    public String getDescription() {
        return beverage.getDescription() + ", Mocha";
    }

    public double cost() {
        return .20 + beverage.cost();
    }
}
```

We're going to instantiate Mocha with a reference to a Beverage using:

(1) An instance variable to hold the beverage we are wrapping.

(2) A way to set this instance variable to the object we are wrapping. Here, we're going to pass the beverage we're wrapping to the decorator's constructor.

We want our description to not only include the beverage – say "Dark Roast" – but also to include each item decorating the beverage, for instance, "Dark Roast, Mocha". So we first delegate to the object we are decorating to get its description, then append ", Mocha" to that description.

Now we need to compute the cost of our beverage with Mocha. First, we delegate the call to the object we're decorating, so that it can compute the cost; then, we add the cost of Mocha to the result.

```
public class StarbuzzCoffee {

    public static void main(String args[]) {
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription()
                + " $" + beverage.cost());

        Beverage beverage2 = new DarkRoast();
        beverage2 = new Mocha(beverage2);
        beverage2 = new Mocha(beverage2);
        beverage2 = new Whip(beverage2);
        System.out.println(beverage2.getDescription()
                + " $" + beverage2.cost());
```

Order up an espresso, no condiments and print its description and cost.

Make a DarkRoast object.

Wrap it with a Mocha.

Wrap it in a second Mocha.

Wrap it in a Whip.

Vzor Singleton

Potrebujeme presne jeden objekt

- Existujú triedy objektov také, že v aplikácii potrebujeme práve jeden objekt danej triedy:
  - Preferences
  - Registry settings
  - Database connection
  - Printer driver
- V niektorých prípadoch by vytvorenie viacerých objektov danej triedy mohlo spôsobiť problémy
  - Napríklad objekty náročné na pamäť a iné systémové zdroje

Prečo návrhový vzor?

- Problém jedného objektu sa zrejme dá riešiť mnohými spôsobmi:
  - Globálne premenné
  - Static variables
  - Singleton sa stal konvenciou a štandardom pre riešenie tohoto problému
  - Výhoda napríklad oproti globálnej premennej:
  - Globálnu premennú musím nainicializovať pri spustení aplikácie. Singleton vtedy, keď ho potrebujem

Problém

- Ako zariadiť, aby nebolo možné vytvoriť viac než jeden objekt nejakej triedy?
- ...skúste niečo vymyslieť!

Ako vytvoríme objekt?

```
new MyObject();
```

Takýchto volaní konštruktora môžeme mať samozrejme ľubovoľne veľa.

Ak je MyObject public class, môže takýto objekt vytvoriť akýkoľvek iný objekt

Čo takáto konštrukcia?

```
public MyClass{
        private MyClass()}
```

Toto je z hľadiska jazyka legálna konštrukcia, avšak zdanlivo nedáva zmysel:

Iba objekty typu MyClass môžu volať konštruktor MyClass. To je nám však na nič, lebo objekt typu MyClass nemá ako vzniknúť

Ešte iná konštrukcia

```
public MyClass{
    private MyClass() {}
    public static MyClass
  getInstace(){
        return new MyClass();
    }
```

Static vlastnosti a metódy

- Static vlastnosti a metódy patria triede. Je možné proužívať ich aj vtedy, keď neexistuje žiaden objekt danej triedy
- Obsahujú dáta a správanie, ktoré sú nezávislé od konkrétneho objektu
- Static members sú inicializované systémom (.NET framework, JVM) pred tým, než sú prvýkrát použité

Static:príklad

```
public class Automobile
{
    public static int NumberOfWheels = 4;
    public static int SizeOfGasTank
    {
        get
        {
            return 15;
        }
    }
    public static void Drive() { }
    public static event EventType
RunOutOfGas;

    //other non-static fields and properties...
}
```

Static trieda

- static môže byť aj trieda. Znamená to, že obsahuje iba static members.
- Nie je možné vytvárať inštancie static tried

Späť k singletonu

```
public MyClass{
    private MyClass() {}
    public static MyClass
  getInstace(){
        return new MyClass();
    }
```

Vytváranie objektov triedy MyClass

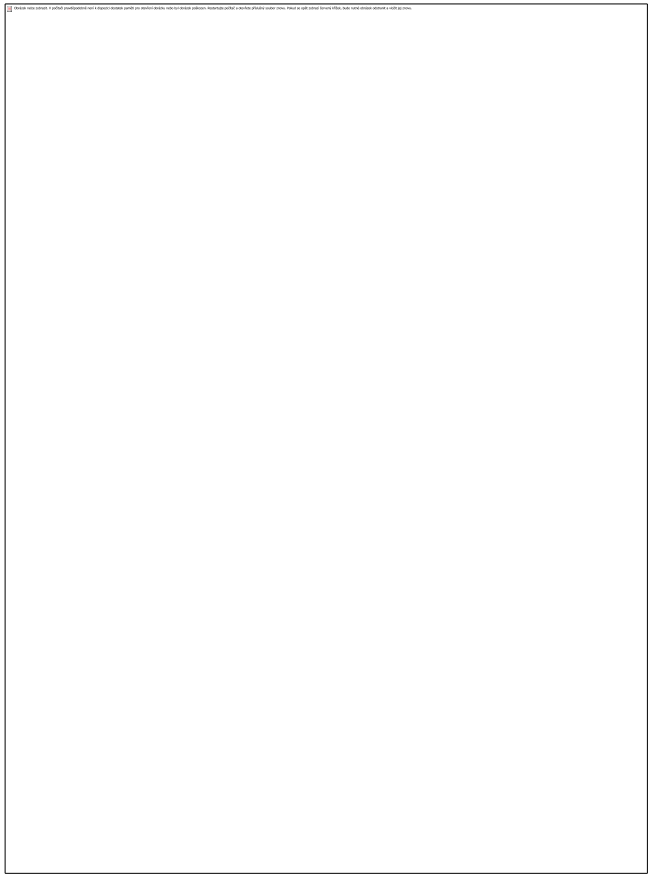
Pri použití uvedenej konštrukcie sa nové objekty triedy MyClass budú vytvárať jednoducho:

```
MyClass.getInstance();
```

Posledná úprava konštrukcie

```
public Singleton{
   private static Singleton UniqueInstance;

   private Singleton() {}

   public static Singleton getInstace(){
       if (UniqueInstance == null) {
          UniqueInstance = new Singleton();
       }
       return Singleton();
   }
   //Dalsie uzitocne metody
}
```

Singleton: class diagram

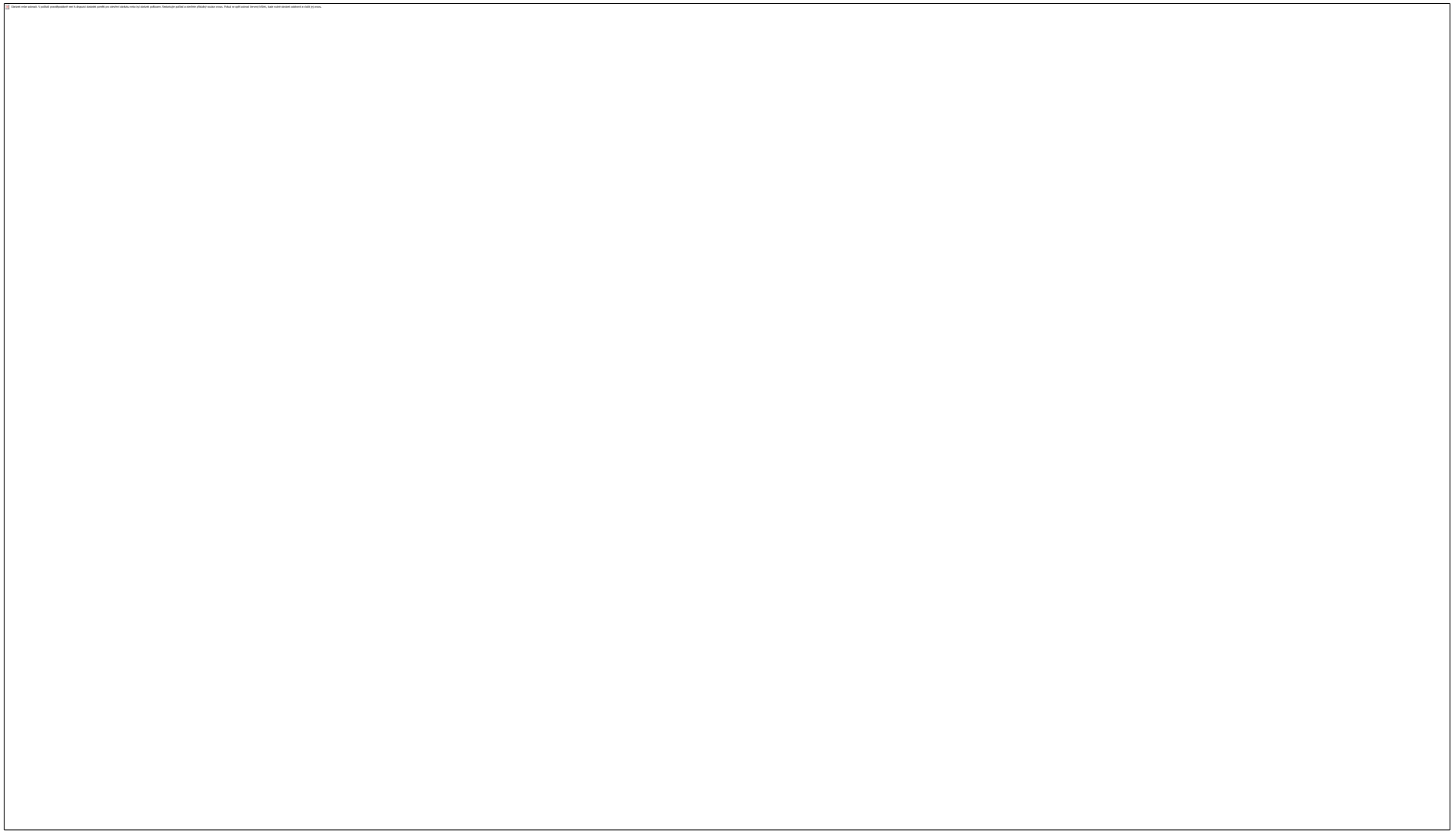


```
class MSSqlConnection {
    private static MSSqlConnection uniqueInstance;
    private string connectieString = @"Data Source=.\SQLEXPRESS;Database...“
    private SqlConnection connectie;

    private MSSqlConnection() { }
    public static MSSqlConnection getInstance()  {
        if (uniqueInstance == null) {
            uniqueInstance = new MSSqlConnection();
        }
        return uniqueInstance;
    }
    public void Pripoj(){
        connectie = new SqlConnection(connectieString);
        connectie.Open();
    }
    public SqlConnection getConnectie()  {
        return connectie;
    }
    public void Odpoj() {
        connectie.Close();
    }
  }
}
```

Prečo chceme iba jedno pripojenie?

- Každá aplikácia potrebuje presne jedno pripojenie
- Správa pripojení sa typicky zanedbáva. Bežná aplikácia má otvorené desiatky pripojení
- Počet pripojení k databázovému serveru môže byť obmedzený na strane servera
- Vysoký počet otvorených pripojení môže spôsobiť problémy s výkonnosťou

Potenciálny problém: multithreading



Riešenie problému

Jedným z možných riešení tohoto problému je „proaktívne“ vytvorenie inštancie:

```
public class Singleton {
    private static Singleton UniqueInstance
  = new Singleton();

    private Singleton() {}

public static Singleton getInstance() {
        return UniqueInstance;
    }
}
```

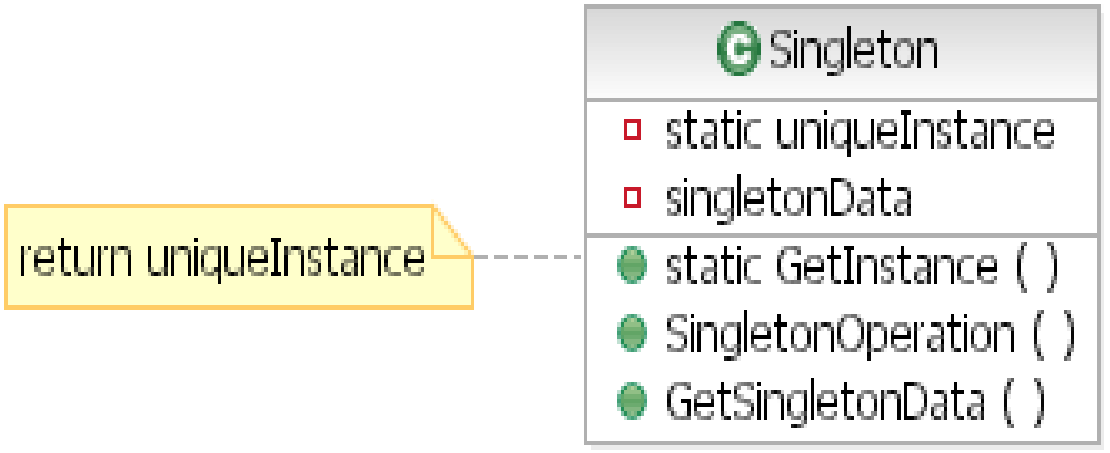
Singleton podľa GoF

- Názov: Singleton
- Klasifikácia: Object creational
- Zámer: Zaistiť, aby trieda mala iba jednu inštanciu a poskytnúť globálny access point k tejto inštancii
- Motivácia
  - Potrebujeme také triedy, ktoré budú mať iba jednu inštanciu
  - Príklad: printer spooler, file system,...
  - Najlepšie je vytvoriť triedu, ktorá sama bude dohliadať na vytváranie inštancií

Singleton podľa GOF II.

- Aplikovateľnosť: Singleton sa používa tam, kde:
  - Musí existovať iba jedna inštancia triedy a táto musí byť prístupná z jedného bodu
  - Potrebujeme jednu inštanciu rozšíriteľnú pomocou dedičnosti. Klienti budú pristupovať k rozšírenej inštancii bez toho, aby modifikovali jej kód (???)

Singleton podľa GoF: Štruktúra



Singleton podľa GoF: Účastníci

- Singleton:
  - definuje operáciu GetInstance(), pomocou ktorej môžu klienti pristupovať k jeho jedinej inštancii
  - môže byť zodpovedný za vytvorenie svojej jedinej inštancie

Singleton podľa GoF: Spolupráca

- Klienti pristupujú k inštancii singletonu výlučne pomocou operácie GetInstance()

Singleton podľa GoF: Dôsledky

- Singleton má nasledujúce výhody
  - Kontrolovaný prístup k jedinej inštancii
  - Redukovaný namespace: Singleton je vylepšním oproti globálnym premenným. Zabraňuje znečisteniu name space globálnymi premennými obsahujúcimi jedinečné inštancie
  - Umožňuje prepísať operácie a reprezentácie pomocou dedičnosti (???)
  - Umožňuje povolenie viacerých inštancií (ak sa rozhodneme, že potrebujeme viac inštancií, je táto úprava jednoduchá a izolovaná)

Singleton podľa GoF: Implementácia a ukážka kódu

- Vysvetlenie a príklady implementácie pomocou jazyka C++

Singleton podľa GoF: Známe použitia

- Smalltalk-80: vzťah medzi triedami a ich meta-triedami (metatriedy obsahujú definície tried)
- InterViews user interface toolkit: Prístup k jedinečným inštanciám tried Session a WidgetKit

Singleton podľa GoF: príbuzné vzory

- Singleton sa využíva pri realizácii niektorých iných vzorov (Abstract Factory, Builder, Prototype)

Technológia ADO.NET

Cieľ prednášky a cvičení

- Zoznámiť sa so základnými princípmi technológie ADO.NET.
- Cieľom nie je naučiť sa všetky technické detaily
- ADO.NET ako príklad riešenia prístupu k perzistentným dátam
- Ukážeme si základy a vyskúšame si ju
- Vyskúšame si nie to najtypickejšie použitie (pomocou wizardu) ale také, pomocou ktorého to čo najlepšie pochopíme (vlastnoručné programovanie)

Študijné materiály

- Dokumentácia ADO.NET “Creating Data Applications by Using Visual Studio:“ ms-help://MS.VSCC.v90/MS.msdnexpress.v90.en/dv_raddata/html/28edce21-220a-484c-b461-a75b0232d293.htm
- Kniha napr. Dalibor Kačmář (?)

Účel technológie ADO.NET

- Problém: Ako z danej aplikácie (C#) pristupovať k perzistentným dátam (databáza, XML, binárny súbor)
- ADO.NET slúži na prepojenie aplikácií s databázou
- ADO.NET poskytuje objekty, do ktorých si môžeme natiahnuť dáta z databázy a pracovať s nimi v aplikácii. Pomocou tých istých objektov môžeme dáta do databázy znova uložiť

Terminológia:

- Hoci ADO.NET pracuje na princípe relačnej databázy, používa nesprávne názvoslovie:
  - Tabel namiesto Relation
  - Column namiesto Attribute
  - Row namiesto Tuple
  - Relation pre vyjadrenie referenčnej integrity (relationship)
- Pretože tieto termíny sú použité v názvoch tried, budeme ich v tejto prednáške používať namiesto správanho názvoslovia relačného modelu, ktorému inak dávame prednosť

História

- Technológia ADO: Active Data Objects
- Veľmi populárna technológia pre prístup k databázam na platforme Windows
- Schopná pracovať s rôznymi databázami

Princípy technológie ADO

- Ado pozná 2 druhy objektov
  - Pre komunikáciu s databázou
  - Pre uchovávanie dát a prácu s nimi
- Aplikácia sa pri inicializácii pripojí k dátovému zdroju a ostáva pripojená až kým sa neuzavrie
- V horšom prípade si vytvára pripojenie pre každý prístup k databáze a neuzatvára ho – databázové servery bývajú preplnené užívateľskými pripojeniami.

Kontext technológie ADO

- Lokálne riešenia:
  - Dáta sú na disku počítača, v ktorom sú spracovávané
  - Dáta sú uložené na „blízkom“ serveri so zaručenou vysokou kvalitou spojenia
  - Výkonnosť a škálovateľnosť nebývajú problematické

Súčasná situácia

- „Klasické“ aplikácie sú stále viac nahradzované riešeniami na báze internetu:
  - Vzdialené servery – kvalita a rýchlosť spojenia nie je garantovaná
  - Aplikácie spracovávajú dáta s rôznych heterogénnych zdrojov:
    - Rôzne platformy (Windows, Unix, Mainframe): interoperabilita
    - Rôzne formáty dát (databázy, XML)
- Tieto aspekty zohľadňuje ADO.NET

Základné prínosy technológie ADO.NET

- Odpojené dáta
- Silná podpora XML
- Riadený kód
- Nový objektový model

Odpojené dáta:

- Spojenie so zdrojom dát sa vytvára iba počas spracovania požiadavky na dáta a prenosu dát
- Databázy: spracovanie dotazu a prenos dát
- Klient má možnosť pracovať s dátami aj bez priameho prístupu k zdroju dát
- Po ukončení spracovania sa spojenie obnoví a zmeny sa premietnu do databázy (zdroja dát)

Silná podpora XML

- XML sa stáva štandardom pre prenos informácií predovšetkým v oblasti internetu a sietí veľkých firiem
- Výhody
  - Formát XML je platformovo nezávislý
  - Pružnosť a univerzálnosť jazyka
  - Podpora XML je plne integrovaná do ADO.NET

Riadený kód

- ADO.NET je napísané v riadenom kóde
- Garbage collection, Caching, systém exceptions

Nový objektový model

- Pôvodný objektový model je postavený na objekte Recordset
- Nový objektový model je silne vylepšený: Dataset

Architektúra ADO.NET

- Dátové komponenty
- Managed provider komponenty

Dátové komponenty:

- Obsahujú dáta z rôznych dátových zdrojov spolu so vzťahmi medzi dátami
- Ich obsah a štruktúra sú univerzálne a nezávislé od typu dátového zdroja
- Patria sem triedy ako DataSet, DataTable, DataRow, DataColumn a DataRelation

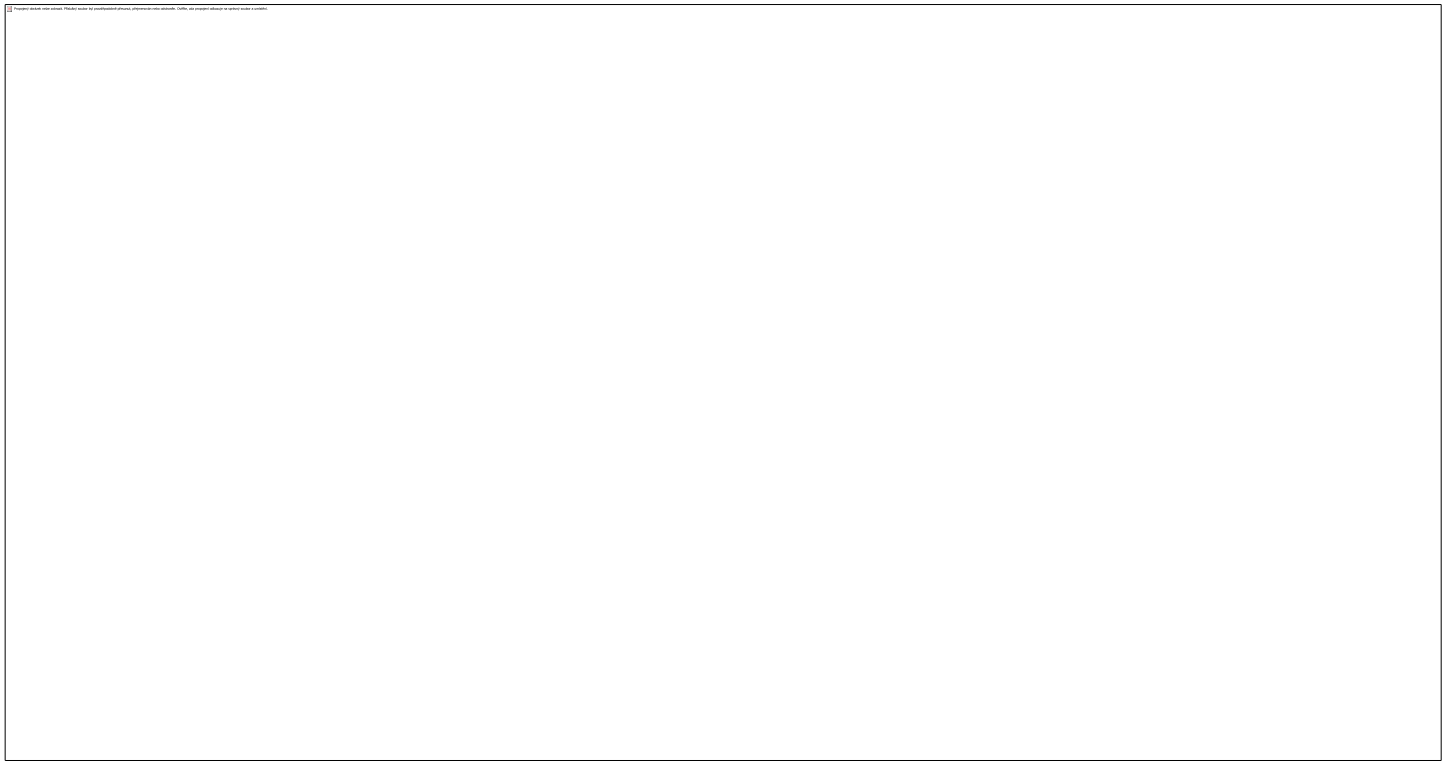
DataSet

- Základný objekt technológie ADO.NET
- Jednoduchá in-memory (relačná) databáza
- Off-line kópia (časti) databázy
- Vlastnosti DataSetu:
  - Úplná nezávislosť na dátovom zdroji
  - Jednotlivé tabuľky môžu byť naplnené z rôznych dátových zdrojov
  - Plne podporuje prácu s XML
  - Podporuje integritné obmedzenia (referenčná integrita) a constraints všeobecne

Managed provider komponenty

- Ich úlohou je sprostredkovať styk s dátovým zdrojom
- Základné funkcie: operácie Select, Update, Insert a Delete
- Dve podskupiny:
  - DataAdapter na prepojenie dátových komponent so zdrojom dát
  - DataReader (na veľmi rýchly, forward-only a read-only prístup k dátam), Command a Connection

Práca s dátami (the data cycle)



Pripojenie k databáze

- Potrebujeme nejaký druh dvojsmernej komunikácie medzi aplikáciou a zdrojom dát
- Objekt SQLConnection
- VisualStudio ponúka DataSource configuration wizard

Objekt SQLConnection

- Vlastnosť ConnectionString: definuje vlastnosti spojenia
- ConnectionString sa skladá z množiny podreťazcov typu:

  Kľúčové slovo = Hodnota;

- Kľúčových slov je zhruba 30

Najdôležitejšie kľúčové slová ConnectionString

- Data Source: meno alebo adresa serveru, ku ktorému sa chceme pripojiť
- Database: databáza, ku ktorej sa chceme pripojiť
- Integrated Security (Yes/No): udáva, či chceme využiť integrovaný systém zabezpečenia OS Windows
- Connect Timeout: ak sa nepodarí nadviazať spojenie do tohto času, nastane chyba
- User ID: užívateľské meno pripojenia
- Password: heslo

ConnectionString: príklad

```
connStr = @"Data Source=.\SQLEXPRESS;
    Database=Kniznica1;Integrated Security=Yes; Connect
    Timeout=30;
```

```
connStr = @"Data Source=.\SQLEXPRESS;
    Database=Kniznica1;Integrated Security=No; Connect
    Timeout=30;User ID=Uzivatel; Password=heslo";
```

Vytvorenie pripojenia

- ConnectionString sa predá ako parameter konštruktoru objektu triedy SQLConnection

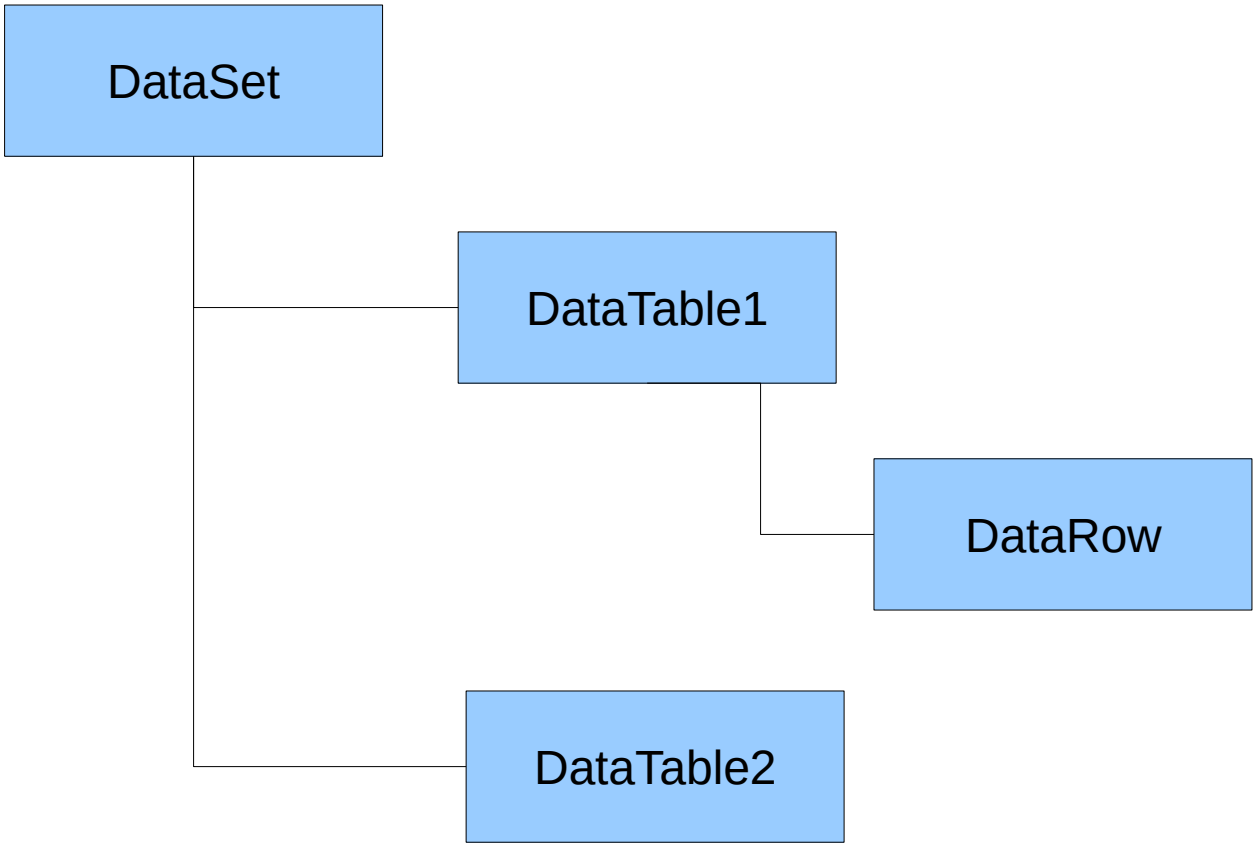
```
string connStr;
SqlConnection spojenie;
spojenie = new SqlConnection(connStr);
```

- Spojenie sa otvorí zavolaním metódy Open

```
spojenie.Open();
```

Príprava aplikácie na príjem dát

- V prípade odpojeného riešenia je potrebné vytvoriť dočasné úložisko dát v aplikácii: objekty triedy DataSet



Vytvorenie datasetu

- Vytvorenie datasetu je jednoduché:

```
DataSet MyDataSet;
MyDataSet = new DataSet();
```

- Aplikácia je teraz pripravená na naplnenie datasetu (t.j. vytvorenie on-line kópie databázy, ktorú budeme používať)
- Iná možnosť je použiť „Dataset wizard“, ktorý je k dispozícii vo Visual Studiu

Natiahnutie dát do aplikácie

- Dáta sa do aplikácie prenesú pomocou SQL dotazu v databáze
- SQL dotaz sa vykoná pomocou triedy SQLDataAdapter
- Trieda SQLDataAdapter má nasledujúce dôležité metódy a vlastnosti

–SelectCommand: SQL príkazu dotazu, ktorý chceme nad databázou vykonať

–Fill(MyDataSet, „MyDataTable“): Okopíruje výsledok dotazu SelectCommand do datasetu MyDataSet do tabuľky MyDataTable

Natiahnutie dát do aplikácie: príklad

```
SqlDataAdapter MyAdapter;
MyAdapter = new SqlDataAdapter();
MyAdapter.SelectCommand = new SqlCommand(
        "select * from Tabulka1", spojenie);
MyAdapter.Fill(MyDataset, "Tabulka2");
MyAdapter.SelectCommand = new SqlCommand(
        "select * from Tabulka2 where X", spojenie);
MyAdapter.Fill(MyDataset, "Tabulka2");
```

Objekty triedy DataTable

- DataSet obsahuje množinu objektov DataTable (pre každú tabuľku jeden)
- K týmto objektom môžeme pristupovať priamo:

```
DataTable MyTabulka;
MyTabulka = MyDataset.Tables["Tabulka1"];
```

DataRow: prístup k riadkom tabuľky

- DataRow objekt reprezentuje jeden riadok tabuľky (DataTable)
- Používa sa na prístup k riadkom tabuľky
  - čítanie, vkladanie, update, vymazávanie
- Vlastnosť Rows objektu DataTable reprezentuje kolekciu všetkých riadkov tabuľky DataTable:

```
public DataRowCollection Rows {get;}
```

- Rows obsahuje objekty typu DataRow
- Typ DataRowCollection je kolekcia so špeciálnymi metódami pre prácu s riadkami tabuľky

DataTable.Select

- Metóda Select umožňuje vybrať podmnožinu riadkov tabuľky
- Vracia pole (array) objektov typu DataRow
- Select(): vráti pole všetkých riadkov tabuľky
- Select(string): vráti pole tých riadkov tabuľky, ktoré spĺňajú podmienku zadanú ako parameter

DataTable.Select: príklad

```
DataTable MyTabulka;
DataRow[] VsetkyRiadky;
DataRow[] Riadky;
VsetkyRiadky = MyTabulka.Select();
Riadky = MyTabulka.Select("Stlpec1='xxx'");
```

DataRow: prístup k jednotlivým hodnotám

```
DataRow MojRiadok;
String x;
x = (string)MojRiadok["Stlpec1"];
x = (string)MojRiadok[0];
```

Pridanie nového riadku

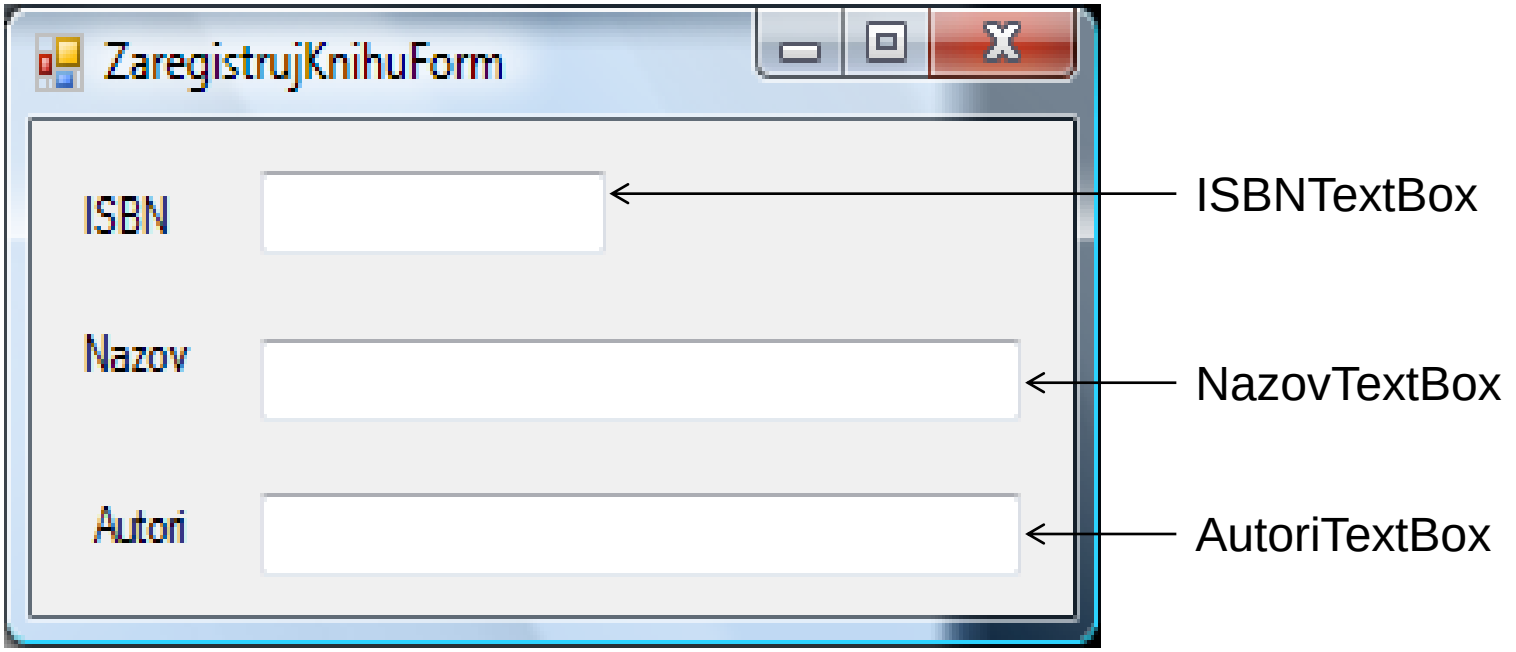
1. DataTable.NewRow(): vytvorí nový riadok. Riadok obsahuje všetky stĺpce tabuľky DataTable. **Pozor:** Tento riadok nie je ešte automaticky súčasťou tabuľky!
2. Nový riadok sa stane súčasťou tebuľky až po pridaní do kolekcie Rows pomocou metódy Add (Rows.Add(*riadok*))

Pridanie nového riadku: príklad

```
row = table.NewRow();
row["id"] = i;
row["item"] = "item " + i.ToString();
table.Rows.Add(row);
```

Zobrazenie dát: príklad

- Úloha: Knižničný systém.



- Po vyplnení čísla ISBN systém vyhľadá v databáze príslušnú knihu a zobrazí jej názov a autorov

Príklad: databáza

- Databáza: Kniznica1
- Tabuľka: Kniha
- Stĺpce:

  ISBN nchar(10) not null;
  Nazov nchar(100);
  Autori nchar(100);

- Primárny kľúč: ISBN

Príklad: použité premenné

```
private DataSet ZoznamKnih;
private DataTable KnihaTbl;
private string connStr;
public SqlDataAdapter KnihaAdapter;
private SqlConnection spojenie;
```

Príklad: inicializácia (konštruktor formulára)

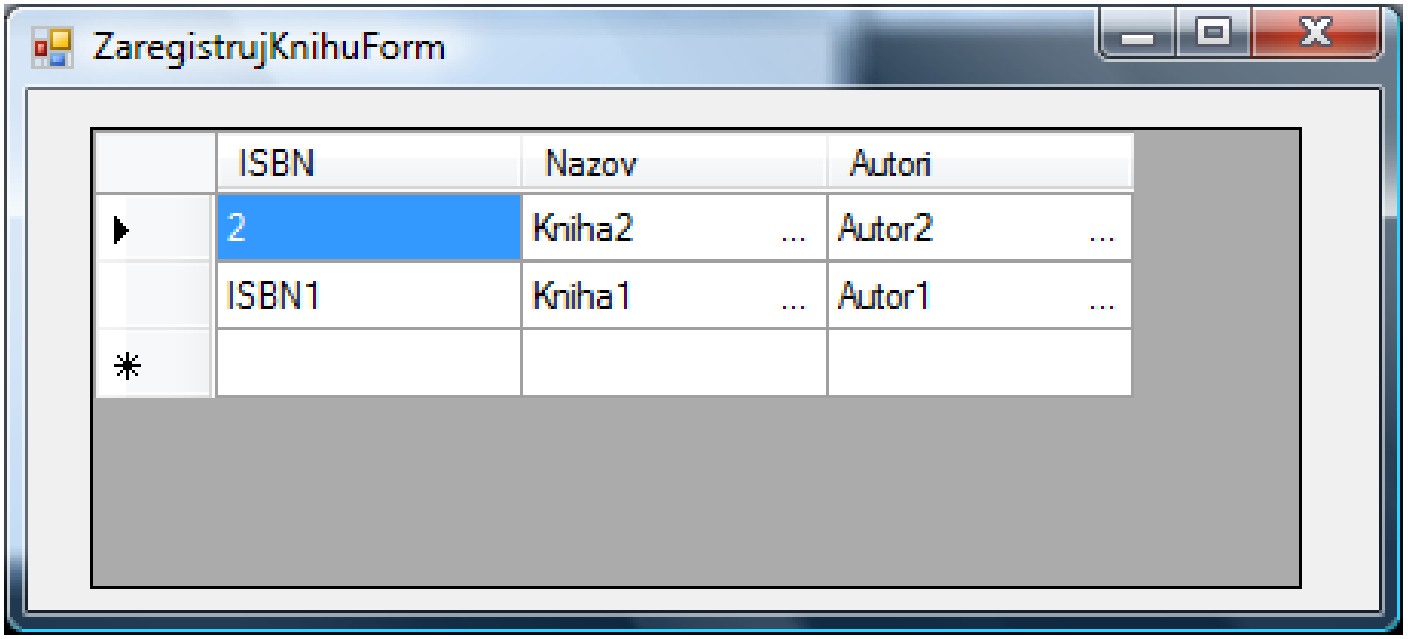
```
public ZaregistrujKnihuForm()
        {
            InitializeComponent();
            connStr = @"Data Source=.\SQLEXPRESS;
           Database=Kniznica1;IntegratedSecurity=Yes;Connect
           Timeout=30";
            spojenie = new SqlConnection(connStr);
            spojenie.Open();
            ZoznamKnih = new DataSet();
            KnihaAdapter = new SqlDataAdapter();
            KnihaAdapter.SelectCommand = new SqlCommand(
                "select * from Kniha", spojenie);
            KnihaAdapter.Fill(ZoznamKnih, "Kniha");
            KnihaTbl = ZoznamKnih.Tables["Kniha"];
            spojenie.Close();
        }
```

Príklad: nájdenie a zobrazenie riadku tabuľky

```
private void ISBNTextBox_Leave(object sender, EventArgs e)
{
    DataRow[] NajdeneKnihy;
    NajdeneKnihy = KnihaTbl.Select("ISBN='"
                                    + ISBNTextBox.Text + "'");

     if (NajdeneKnihy.GetLength(0) > 0)
     {
          NazovTextBox.Text = (string)NajdeneKnihy[0][0];
          AutoriTextBox.Text = (string)NajdeneKnihy[0]["Autori"]
     }
     ....
}
```

DataGridView

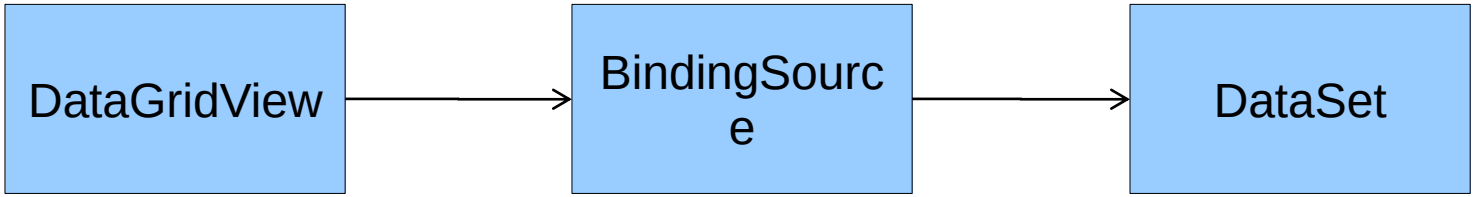
- DataGridView je control špeciálne vytvorený na zobrazovanie objektov typu DataSet.



- Umožňuje zobrazovať, pridávať a meniť dáta tabuľky

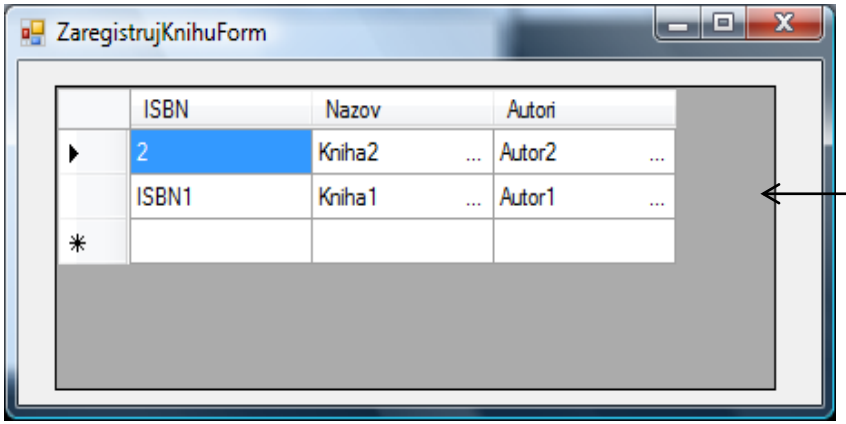
DataGridView: zobrazenie dát

- Dáta sa zobrazia pomocou špeciálneho objektu BindingSource
- BindingSource prepojí prvok užívateľského rozhrania (DataGridView) s dátami (DataSet)




```
DataGridView.BindingSource = MyBindingSource;
MyBindingSource.DataSource = MyDataSet;
MyBindingSource.DataMember = "<nazov tabulky>";
```

DataGridView: príklad



Oproti predchádzajúcemu príkladu pribudne jedna premenná:

```
private BindingSource KnihaBindingSource = new BindingSource();
```

a ďalej:

```
KnihaBindingSource.DataSource = ZoznamVytlackov;
KnihaBindingSource.DataMember = "Kniha";
KnihaDG.DataSource = KnihaBindingSource;
```

Definovanie vzťahov (relationships) medzi tabuľkami datasetu

- Medzi tabuľkami datasetu je možné definovať referenčné vzťahy (relationships)
- Referenčné vzťahy (v ADO.NET) majú kardinalitu 1:N
- Príklad: Faktúra a riadky faktúry
- ADO.NET nazýva vzťahy – veľmi nesprávne - „relation“
- Trieda DataSet má vlastnosť Relations, ktorá obsahuje kolekciu všetkých definovaných relácií

Trieda DataRelation

- Relácie sa realizujú pomocou objektov triedy DataRelation
- Najdôležitejšie vlastnosti:

– **DataSet:** dataset, ku ktorému daná relácia patrí

– **ParentTable:** tabuľka „parent“. Vo vzťahu 1:N je to tabuľka na strane 1

– **ChildTable:** tabuľka „child“ Vo vzťahu 1:N je to tabuľka na strane N

DataRelation: vytvorenie

Existuje niekoľko možných volaní konštruktora. Najbežnejšie má formu:

DataRelation(String, DataColumn, DataColumn)

String: názov relácie
DataColumn: stĺpec (kandidátny kľúč) parent tabuľky
DataColumn: zodpovedajúci stĺpec child tabuľky (cudzí kľúč)

Nakoniec sa ešte takto vytvorená relácia musí pridať k datasetu:

DataSet.Relations.Add(relation)

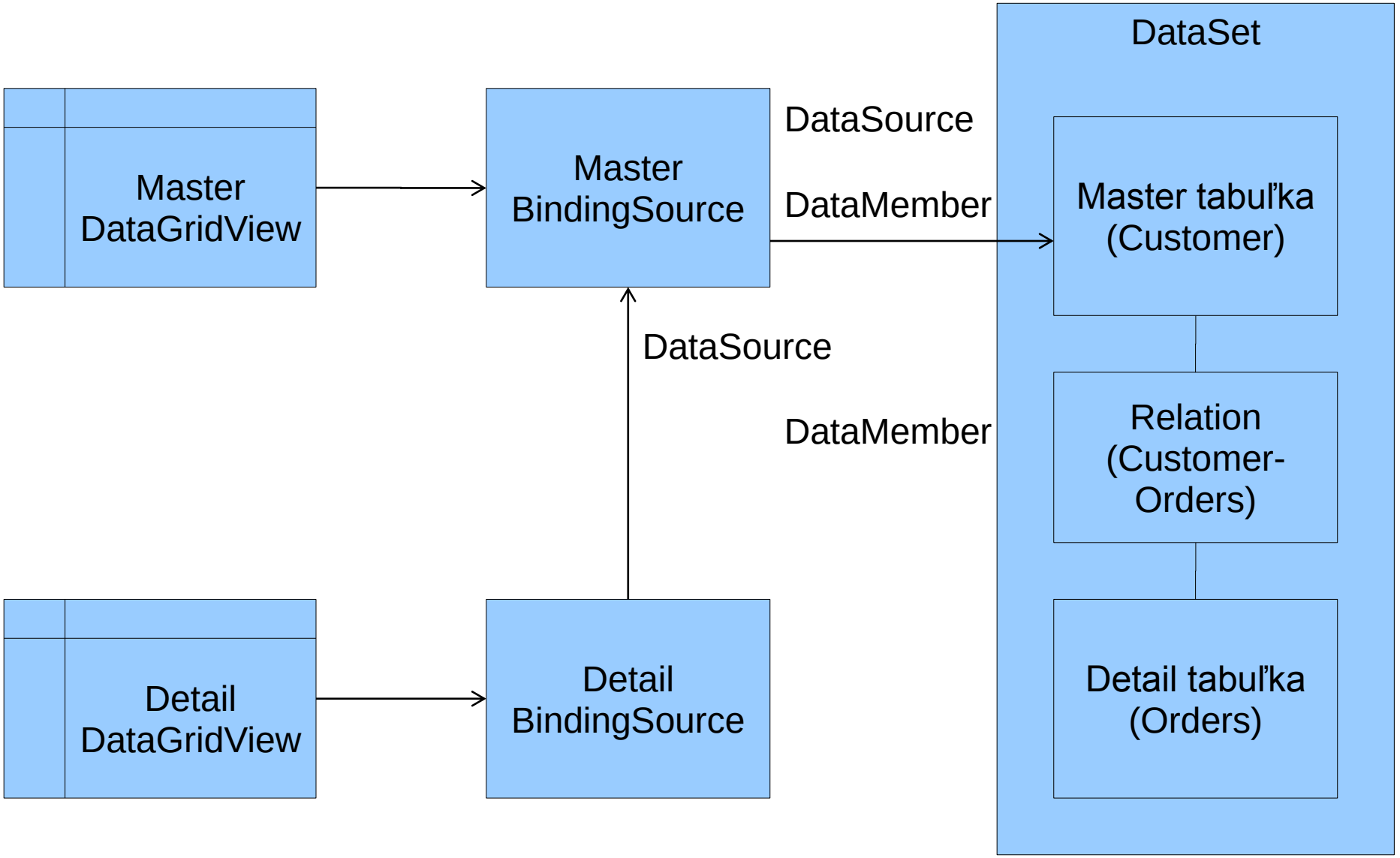
Vytvorenie relácie: príklad

```
private void CreateRelation()
{
    DataColumn parentColumn =
    DataSet1.Tables["Customers"].Columns["CustID"];

    DataColumn childColumn =
    DataSet1.Tables["Orders"].Columns["CustID"];

    relCustOrder = new ("CustomersOrders", parentColumn,
    childColumn);

    DataSet1.Relations.Add(relCustOrder);
 }
```

DataGridView: Master/Detail tabuľky

- DataGridView môžeme použiť aj na zobrazenie tabuliek so vzťahom 1:N (Master/Detail)
- Príklad: Faktúra a riadok faktúry, Zákazník a objednávky zákazníka
- Vytvoria sa dva DataGridView controls. Jeden pre Master tabuľku a jeden pre detail tabuľku

Master/Detail DataGridView



Master/detail data grids

```
masterBindingSource.DataSource = data;
masterBindingSource.DataMember = "Customers";

detailsBindingSource.DataSource = masterBindingSource;
detailsBindingSource.DataMember = "CustomersOrders";

masterDataGridView.DataSource = masterBindingSource

detailsDataGridView.DataSource = detailsBindingSource
```

Uloženie dát do databázy

- Zmeny dát v DataGridViewControl sa automaticky premietnu do Datasetu
- Nepremietnu sa však automaticky do databázy!
- To isté platí pre novovytvorené riadky a vymazané riadky
- Pri ukončení spracovania je treba previesť synchronizáciu dát v DataSete a v databáze
- Použije sa na to metóda DataAdapter.Update

DataAdapter.Update

- Metóda Update vykoná príslušný INSERT, UPDATE alebo DELETE príkaz nad databázou
- Metóde Update treba presne povedať, aký príkaz nad databázou sa má vykonať
- Príkaz pre databázu jej vygeneruje objekt triedy SQLCommandBuilder

SQLCommandBuilder

- Objekty triedy SQLCommandBuilder zinventarizujú zmeny prevedené v datasete a vygenerujú príslušný SQL príkaz
- Metódy:
  - SQLCommandBuilder.GetInsetCommand()
  - SQLCommandBuilder.GetUpdateCommand()
  - SQLCommandBuilder.GetDeleteCommand()

Zmena dát v databáze: príklad

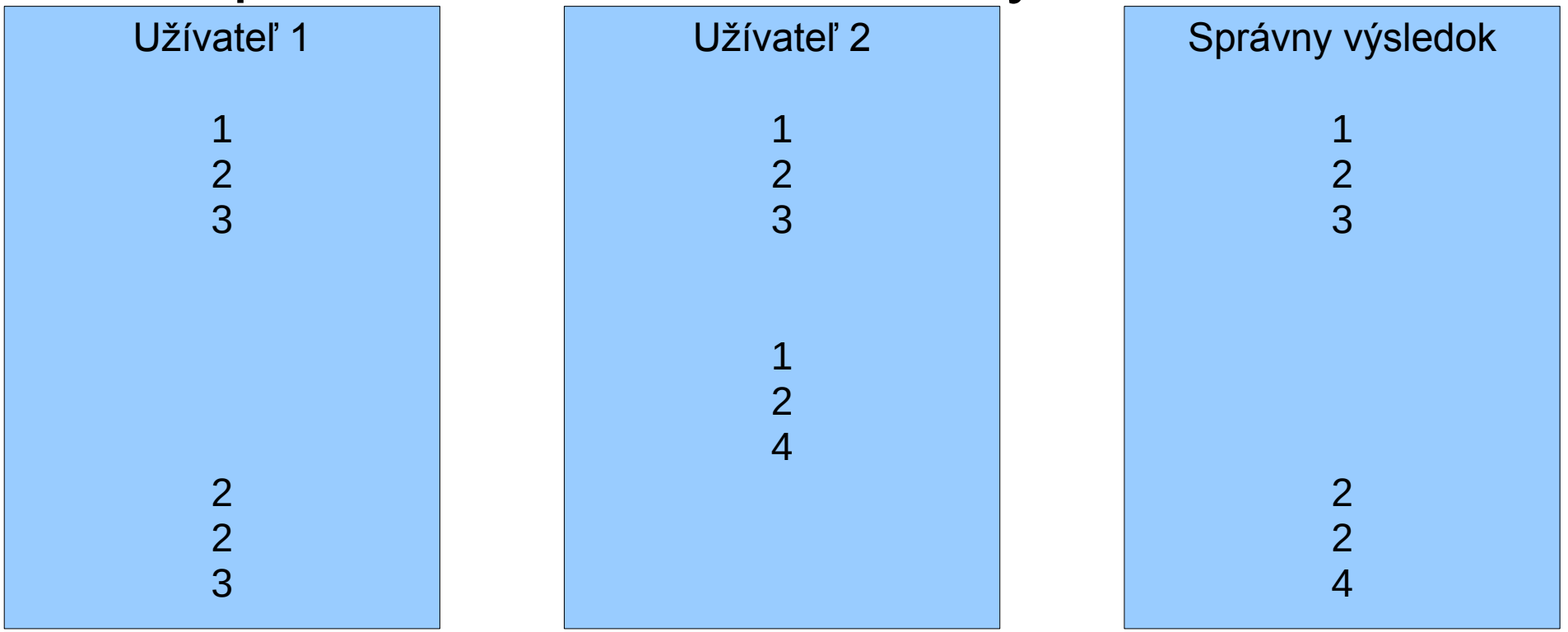
```
SqlDataAdapter adapter = new SqlDataAdapter();

adapter.SelectCommand = new SqlCommand(queryString,
    connection);

SqlCommandBuilder builder = new SqlCommandBuilder(adapter);

connection.Open();

DataSet dataSet = new DataSet();

adapter.Fill(dataSet, tableName);

//code to modify data in DataSet here
    builder.GetUpdateCommand();

//Without the SqlCommandBuilder this line would fail

adapter.Update(dataSet, tableName);
```

Zmena dát v databáze: príklad 2

```
connStr = @"...";

spojenie = new SqlConnection(connStr);

spojenie.Open();

SqlCommandBuilder builder = new
    SqlCommandBuilder(KnihaAdapter);

builder.GetInsertCommand();

KnihaAdapter.Update(ZoznamKnih.Tables["Kniha"]);

builder.GetUpdateCommand();

KnihaAdapter.Update(ZoznamKnih.Tables["Kniha"]);

spojenie.Close();
```

Odpojený prístup: nebezpečenstvo

- Pri odpojenom prístupe k dátam v databáze (a teda mnohoužívateľskom prístupe) vzniká nebezpečenstvo nekonzistentných dát



Užívateľ 1 prepísal zmenu vykonanú užívateľom 2