Basi di dati orientate agli oggetti

Giovanna Guerrini

DISI - Università di Genova
via Dodecaneso 35
16146 Genova - Italy

guerrini@disi.unige.it

SOMMARIO

- Introduzione
- Modelli dei dati ad oggetti
  - Concetti di base (oggetti ed identificatori, oggetti complessi, incapsulazione, classi, ereditarietà)
  - Modello dei dati dello standard ODMG (e linguaggio di definizione dei dati ODL)
- Linguaggi di interrogazione
  - Accessi agli oggetti
  - Il linguaggio di interrogazione dello standard ODMG (OQL)

SOMMARIO

- Versioni ed evoluzione
- Cenni ad aspetti architetturali
- Cenni ad altri OODBMS
- Conclusioni

+

- *ObjectStore*

INTRODUZIONE

- le prime e più rilevanti applicazioni dei DBMS sono state in campo finanziario ed amministrativo
- questo ha influenzato l'organizzazione e l'utilizzo dei dati nei DBMS
- innovazioni hardware hanno aperto il mercato a nuove applicazioni che richiedono strumenti software adeguati

Esempi di tali applicazioni sono:

- (Iper)testi, multimedia
- Progettazione: CAD/CAM, CASE
- Computer integrated manufacturing
- Sistemi esperti/basi di conoscenza
- Office automation
- Reti intelligenti (telecomunicazioni)
- Sistemi di supporto delle decisioni
- Sistemi informativi geografici/cartografici

Tali nuove applicazioni possono essere caratterizzate come *data-intensive programming in the large*

data intensive un programma data-intensive produce e/o richiede grandi quantità di dati

programming in the large programmi molto grandi e complessi, progettati e sviluppati da molti programmatori (software engineering)

Sistemi software molto grandi e complessi che richiedono di gestire grandi quantità di dati

Nuove applicazioni: Requisiti

- Condivisione dei dati
- Persistenza dei dati
- Grandi quantità di dati
- Affidabilità dei dati

- Dati strutturati (tipi complessi)
- Semantica dei dati
- Modellazione del comportamento
- Versioni e long-transaction
- Interoperabilità

I DBMS tradizionali soddisfano solo i primi quattro requisiti

Evoluzione dei DBMS - Alcune direzioni

- DBMS orientati ad oggetti
  (DBMS + programmazione orientata ad oggetti)
- DBMS attivi
  (DMBS + comportamento reattivo - AI)
- DBMS deduttivi
  (DBMS + programmazione logica)
- DBMS relazionali estesi (o object-relational)

INTRODUZIONE

DBMS Orientati ad oggetti

- Gli OODBMS rappresentano una delle più promettenti evoluzioni nell'area basi di dati
- Molti sistemi già disponibili come prodotti commerciali
- Object-orientation sempre più diffuso in ambito software engineering e linguaggi di programmazione: vantaggio di *unicità del paradigma*

INTRODUZIONE

L'approccio ad oggetti nello sviluppo del software

L'object-orientation è una tecnologia chiave per architetture software avanzate e piattaforme di sviluppo di applicazioni

- Richiede maggior tempo di progettazione iniziale
- Riduce significativamente la dimensione del codice
- Richiede minor tempo totale e meno sviluppatori

INTRODUZIONE

Unicità del paradigma

- Nel tradizionale ciclo di vita del software si devono superare diverse barriere ognuna delle quali comporta problemi di comunicabilità
  dal dominio del problema all'analisi (es. DFD), alla programmazione (es. C) alle basi di dati (es. ER+relazionali)
- Nel ciclo di vita del software orientato ad oggetti le varie fasi si basano su unico modello
  – non si deve progettare separatamente la struttura della base di dati
  – non si hanno problemi di comunicazione tra DBMS e linguaggio di programmazione

INTRODUZIONE

Integrazione di sistemi eterogenei

- Un requisito importante è che le nuove applicazioni possano interagire con le applicazioni esistenti e accedere ai dati gestiti da tali applicazioni
- La scelta di uno specifico linguaggio o DBMS dipende dai requisiti correnti dell'applicazione e dalla tecnologia disponibile, che variano nel tempo $\Rightarrow$ sistemi eterogenei
- Il paradigma ad oggetti, grazie all'incapsulazione, permette di risolvere problemi di integrazione

INTRODUZIONE

OODBMS - Definizione

Un OODBMS è un sistema con le funzionalità e le caratteristiche di:

- un linguaggio di programmazione ad oggetti
- un DBMS

Il progetto di un OODBMS richiede l'integrazione della tecnologia delle basi di dati con la tecnologia object-oriented

INTRODUZIONE

Funzionalità di un OODBMS

- object identity
- oggetti complessi
- incapsulazione
- ereditarietà
- overloading e late binding
- completezza computazionale
- estensibilità
- persistenza
- condivisione
- sicurezza
- affidabilità
- linguaggio di interrogazione
- efficienza

INTRODUZIONE

A chi è adatto un OODBMS?

Organizzazioni che:

- hanno necessità di tempi di sviluppo brevi
- adottano programmazione ad oggetti
- hanno necessità di condividere informazione complessa
- sviluppano sistemi intelligenti

INTRODUZIONE

Prodotti e prototipi

- ObjectStore(Object Design)
- GemStone (Serviologic)
- O2 (Ardent Software)
- POET (POET Software)
- Jasmine (Computer Associates)
- Orion (MCC) /Itasca
- Ontos (Ontologic)
- Objectivity/DB (Objectivity)

...

- Iris/OpenODB(Hewlett-Packard)
- Versant (Versant Technology)
- Vision (Innovative Systems)
- G-Base (Graphael)
- Statice (Symbolics)
- Trellis (Digital)
- Zitgeist (Texas Instr.)
- Matisse (Matisse Software)

INTRODUZIONE

Cenni storici

1986-1989 lancio dei primi linguaggi ad oggetti con persistenza
(sistemi stand-alone, non adottano piattaforme standard industriali)
es: G-Base, GemStone, Vbase

1990 primi OODBMS con funzionalità complete
(architettura client/server, piattaforme comuni)
es: Ontos, ObjectStore, Objectivity, Versant, Itasca, O_2, Zeitgeist

1991 nasce ODMG - necessità di uno standard

1993,1997 ODMG-93 e ODMG 2.0

1999 ODMG 3.0 - object data management

MODELLO DEI DATI AD OGGETTI

Concetti base

- Oggetti ed identificatori di oggetti
- Oggetti complessi
- Incapsulazione
- Classi
- Ereditarietà

OGGETTI ED IDENTIFICATORI

Identità

- Ogni entità del mondo reale è modellata come un **oggetto**
- Ogni oggetto ha un identificatore (OID) che lo distingue da tutti gli altri oggetti nella base di dati
- L'identificatore è immutabile ed indipendente dal valore dell'oggetto
  l'oggetto può essere visto come coppia (OID, valore)
- Gli OID in genere non sono visibili agli utenti

OGGETTI ED IDENTIFICATORI

OID e chiavi - Differenze

- Una chiave (valore di uno o più attributi) può essere modificata, l'OID è invece immutabile
- Usando le chiavi, il programmatore deve:
  - selezionare gli attributi da utilizzare come chiave
  - assicurare l'unicità dei valori di chiave

  Gli OID sono gestiti completamente dal sistema

OGGETTI ED IDENTIFICATORI

OID e chiavi - Differenze

- Le chiavi sono uniche all'interno di una relazione
  Gli OID sono unici all'interno dell'intero sistema

- Poichè gli oggetti sono riferiti in modo uniforme è più semplice gestire collezioni di oggetti eterogenei (es. insieme di persone e di dipartimenti)

OGGETTI ED IDENTIFICATORI

OID e chiavi - Differenze

- Rappresentazione di associazioni tra entità

  Modello relazionale value-based
  le associazioni tra entità sono rappresentate tramite chiavi esterne – join

  Modello ad oggetti identity-based
  tramite riferimenti tra oggetti (puntatori) – navigazione (direzionale) del riferimento

OGGETTI ED IDENTIFICATORI

Identità e uguaglianza - Differenze

Il concetto di OID introduce due tipi di uguaglianza tra oggetti:

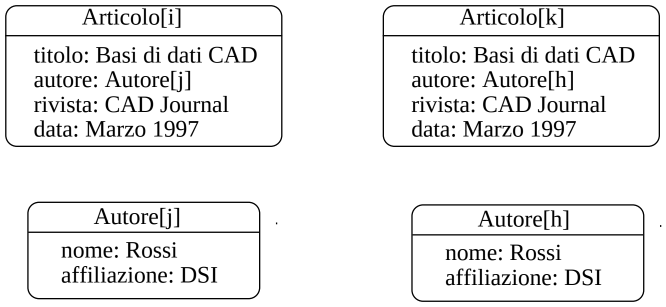
identità se i due oggetti hanno lo stesso OID

uguaglianza per valore se i due oggetti hanno lo stesso valore

- **deep** gli attributi corrispondenti sono uguali (per valore)
- **shallow** gli attributi corrispondenti sono identici

OGGETTI ED IDENTIFICATORI

Esempio di oggetti uguali ma non identici



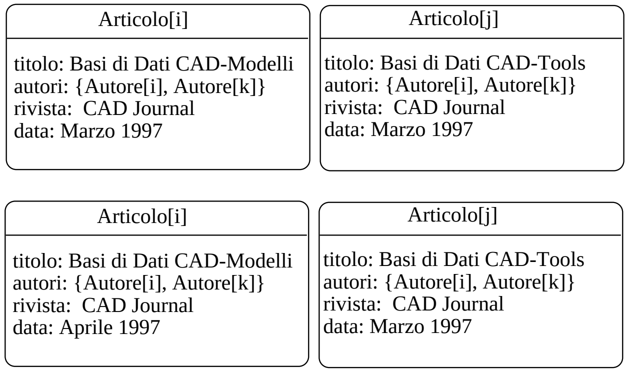
Condivisione di oggetti

- Gli OID permettono la *condivisione* (sharing) di oggetti
- Nel caso di oggetti condivisi lo stato delle componenti è unico
- I cambiamenti alle componenti sono visibili da tutti gli oggetti che li riferiscono

Condivisione di oggetti

(a)

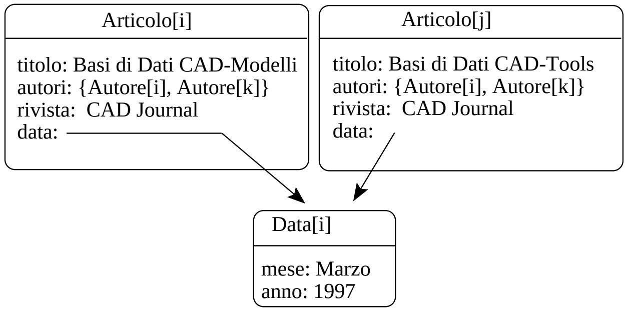
1. 2.



Condivisione di oggetti

(b)

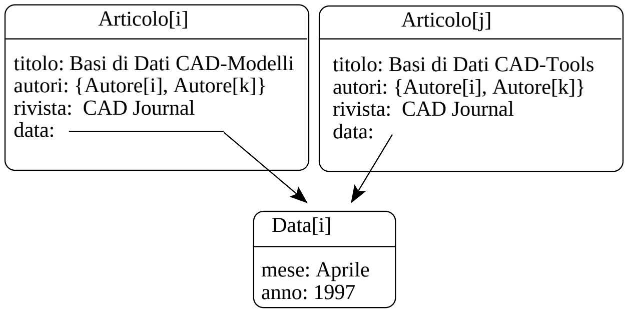
1.



Condivisione di oggetti

(b)

2.



OGGETTI ED IDENTIFICATORI

OID - possibili approcci all'implementazione

- coppia <class-id, identificatore dell'oggetto nella classe>
- identificatore numerico che non fa riferimento all'identificatore della classe
- identificatore del record (RID) in cui è memorizzato l'oggetto
- può anche contenere informazioni sulla collocazione dell'oggetto in basi di dati distribuite

OGGETTI COMPLESSI

Ogni applicazione richiede tipi di dato specifici

Nella maggioranza delle applicazioni, i dati sono oggetti con strutture complesse

- figure geometriche di base e vettori (applicazioni CAD)
- matrici (applicazioni CAM - movimenti dei bracci dei robot)
- un articolo ha un titolo, una lista di autori, un insieme di parole chiave, una lista di capitoli, ognuno con un titolo e una lista di sezioni ...

OGGETTI COMPLESSI

- Non è possibile sviluppare un DBMS che fornisca tutti i possibili tipi di dato che potrebbero servire in un'applicazione
- Gli oggetti del mondo reale devono poter essere "mappati" in oggetti della base di dati nel modo più diretto possibile
- La soluzione è quella di fornire agli utenti dei "building blocks" con cui costruire i tipi di dato necessari

OGGETTI COMPLESSI

Gli OODBMS forniscono:

- tipi di dato strutturati
- oggetti complessi
- tipi di dato (ADT) specifici dell'applicazione
- tipi di dato non strutturati - es. binary large objects (Blobs)

OGGETTI COMPLESSI

Assemblati a partire da *oggetti atomici* mediante *costruttori*

- Oggetti atomici true, false, 25, "this is an atom"

- Costruttori

  tuple [fname: John, lname: Doe]

  set John, Susan, Mary

  array <1:25, 2:20, 3:21>

  list [25, 20, 21]

possono a loro volta essere componenti di altri oggetti

OGGETTI COMPLESSI

Oggetti e valori

Molti sistemi non richiedono che ogni entità sia rappresentata come oggetto, ma distinguono tra valori (o letterali) e oggetti

differenze:

- – i valori sono astrazioni universalmente conosciute (stesso significato per ogni utente)
  – gli oggetti hanno un significato che dipende dall'applicazione

OGGETTI COMPLESSI

Oggetti e valori

- – i valori sono *built-in* nel sistema
  – gli oggetti devono essere definiti

- – l'informazione rappresentata da un valore è se stesso
  – l'informazione di un oggetto sono le relazioni con altri oggetti e valori

- – i valori sono usati per descrivere altre entità
  – gli oggetti sono le entità che devono essere descritte

OGGETTI COMPLESSI

Oggetti e valori

- esempi tipici di valori:

  interi, reali, booleani, caratteri, stringhe

- in alcuni sistemi si hanno anche valori strutturati

  insiemi, liste, tuple, array

  valori strutturati possono contenere (riferimenti ad) oggetti come componenti

OGGETTI COMPLESSI

Vantaggi di un modello che fornisce oggetti complessi

- rappresentazione diretta di oggetti strutturati quali testi, mappe, disegni CAD senza bisogno di decomporli in unità più piccole (tuple o record)
- ricerca e ricomposizione più veloce

OGGETTI COMPLESSI

- Molti sistemi permettono di memorizzare valori di grandi dimensioni non strutturati e non interpretati dal sistema
- **BLOB** (*binary large object*): valori di grandi dimensioni come *bitmap* di immagini o lunghe stringhe di testo
- Non strutturati: il DBMS non conosce la loro struttura, ma è l'applicazione che li usa che sa come interpretarli

INCAPSULAZIONE

In un OODBMS i dati e le operazioni su di essi sono incapsulati in un'unica struttura (l'oggetto)
idea proviene dagli Abstract Data Types (interfaccia + implementazione)

- un oggetto rappresentate un `Impiegato`
- interfaccia (visibile):

  operazione `aggiorna_stip(incr)`
- implementazione = stato + codice operazioni (non visibile)

  attributi `nome:Rossi, stipendio:2,000, skills:{CAD,DB}`

INCAPSULAZIONE

Componenti di un oggetto

Un oggetto consiste quindi di:

- un **OID**, o identificatore
- uno **stato**, o valore, costituito dai valori per un certo numero di *attributi*, o campi
  tali campi possono contenere riferimenti ad altri oggetti
- un **comportamento** costituito da un insieme di *metodi* o operazioni

INCAPSULAZIONE

Metodi

La definizione di un metodo consiste di due componenti:

segnatura specifica il nome del metodo, il nome (e i tipi) degli argomenti, ed eventualmente il tipo del risultato

body consiste di codice scritto in qualche linguaggio di programmazione (eventualmente esteso)

- ObjectStore: C++ o Java
- O2: CO2 (estensione del C)

INCAPSULAZIONE

Metodi: messaggi e implementazioni

- l'implementazione (body) delle operazioni è nascosta, cioè non è visibile dall'esterno
- l'interfaccia di un oggetto è l'insieme delle segnature delle operazioni
  - definisce i *messaggi* cui l'oggetto risponde
  - descrive interazione dell'oggetto con il mondo esterno

INCAPSULAZIONE

Metodi

- I dati e le operazioni sono progettati insieme e sono memorizzati nello stesso sistema ⇒ maggiore *indipendenza logica dei dati*
- Un metodo è invocato mandando un messaggio ad un oggetto
- Mandando lo stesso messaggio a due oggetti di due classi differenti questi possono esibire comporatamenti differenti (*overloading*)

INCAPSULAZIONE

Completezza computazionale

- Viene fornito un linguaggio *general purpose* per scrivere i metodi
- L'intera applicazione può quindi essere completamente scritta in termini di oggetti
- Nei sistemi relazionali, SQL doveva essere utilizzato insieme ad un linguaggio di programmazione (*impedance mismatch*)

INCAPSULAZIONE

Estensibilità logica

- La maggioranza degli OODBM fornisce ricche librerie di classi quali date, valute, figure geometriche
- L'utente può estendere l'insieme di tipi
- Non c'è differenza tra i tipi forniti dal sistema e quelli aggiunti dall'utente

INCAPSULAZIONE

Incapsulazione e basi di dati

- Gli OODBMS non sempre forzano l'incapsulazione stretta
- In alcuni casi, le applicazioni di basi di dati devono semplicemente scrivere e leggere i valori degli attributi di un oggetto
- Se si forza l'incapsulazione stretta, questo può essere fatto solo tramite metodi *accessor* (`getAttr`) e *mutator* (`setAttr`) $\Rightarrow$ si è forzati a scrivere molti metodi banali

INCAPSULAZIONE

Incapsulazione e basi di dati

Diversi approcci:

- attributi possono essere acceduti (letti e scritti) direttamente – es. Orion
- si forza incapsulazione stretta – es. GemStone
- si permette di specificare quali attributi possono essere acceduti direttamente e quali no (attributi pubblici e privati) – es. ODMG, O2, ObjectStore

CLASSI

Istanziazione

- L'istanziazione è un meccanismo che permette di "riutilizzare" la stessa definizione per generare oggetti simili

  il concetto di **classe** è la base per l'istanziazione
- Una classe specifica lo scopo delle sue istanze specificando:
  - una struttura, cioè un insieme di attributi
  - un insieme di messaggi che definiscono l'interfaccia esterna degli oggetti
  - un insieme di metodi che sono invocati da tali messaggi

CLASSI

Tipo, classe, interfaccia - Differenze

- Nel modello ad oggetti sono presenti diversi concetti legati alla descrizione delle caratteristinche di un insieme di oggetti:

  tipo, classe, interfaccia

- La separazione tra tali concetti è piuttosto confusa e le differenze con cui i termini vengono utilizzati varia da sistema a sistema

CLASSI

Tipo, classe, interfaccia - Differenze

Tipo:

- è un concetto principalmente legato ai linguaggi di programmazione

- fornisce la specifica di un insieme di oggetti o valori (operazioni invocabili su di essi)

- è utilizzato a tempo di compilazione per controllare la correttezza dei programmi

CLASSI

Differenze tra il concetto di classe e quello di tipo

Classe:

- fornisce l'implementazione (stato + implementazione delle operazioni) per un insieme di oggetti dello stesso tipo

- fornisce primitive per la creazione di oggetti

- è "first class object"

CLASSI

Tipo, classe, interfaccia - Differenze

Interfaccia:

- fornisce la specifica del comportamento esterno di un imsieme di oggetti

- può essere implementata da una classe

- non può essere instanziata direttamente

CLASSI

Tipo, classe, interfaccia - Differenze

Classe astratta:

- ulteriore concetto presente in alcuni linguaggi/sistemi
- classe che non può essere istanziata direttamente, ma solo specializzata
- può contenere metodi astratti (solo segnatura senza implementazione)

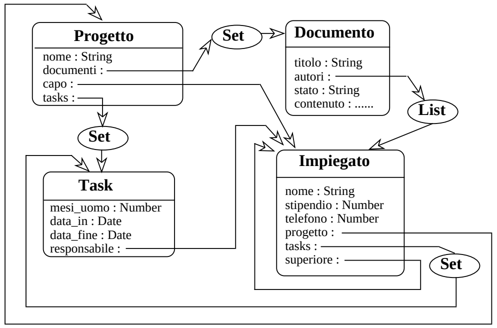
CLASSI

Gerarchia di aggregazione

- Per ogni attributo viene in genere specificato un *dominio*, che specifica il tipo dei possibili valori dell'attributo
- La definizione di una classe include la specifica dei domini degli attributi
- Se una classe C è il dominio di un attributo A di una classe C', si dice che c'è una relazione di aggregazione (o *clientship*) tra C' e C

CLASSI

Gerarchia di aggregazione



CLASSI

Gerarchia di aggregazione

- La gerarchia di aggregazione può contenere cicli
- Il dominio di un attributo può cioè essere la classe stessa
- Nell'esempio, l'attributo `superiore` della classe `Impiegato` ha come valori oggetti della stessa classe `Impiegato`
- I cicli possono ovviamente avere lunghezza arbitraria (cioè anche maggiore di uno)

CLASSI

Estensione

- Oltre ad essere un template, la classe in alcuni sistemi denota anche la collezione delle sue istanze (*estensione*)
- Questo aspetto è importante perchè la classe diventa la base su cui sono formulate le interrogazioni
- Le interrogazioni sono significative solo se applicate a collezioni di oggetti

CLASSI

Estensione

- Quando la classe non ha funzione estensionale, le interrogazioni sono applicate a collezioni (insiemi)
- Possono esserci più insiemi che contengono istanze della stessa classe
- L'estensione automatica ha il vantaggio della semplicità
- L'estensione gestita dall'utente ha il vantaggio della flessibilità

CLASSI

Estensione

Possibili approcci:

- In alcuni sistemi (es. ObjectStore, GemStone e O2) la classe definisce solo la specifica e l'implementazione degli oggetti

  Le interrogazioni e gli indici sono definiti su collezioni di oggetti
- In altri (es. Orion) la classe definisce la specifica e l'implementazione, ed inoltre mantiene l'estensione (insieme delle istanze)

CLASSI

Persistenza degli oggetti

- Persistenza degli oggetti significa:
  - come gli oggetti sono inseriti nella base di dati
  - come gli oggetti sono rimossi dalla base di dati
- Nei sistemi relazionali esistono comandi espliciti per inserire e rimuovere i dati nella/dalla base di dati (INSERT, DELETE)

Persistenza degli oggetti

Inserimento degli oggetti - Approcci:

1. *persistenza automatica*

   ogni oggetto diventa automaticamente persistente quando viene creato
   non c'è bisogno di un comando di inserimento esplicito

2. *radici di persistenza*
   un oggetto è persistente se:

   - ha un nome esterno, oppure
   - è raggiungibile da un oggetto persistente

Persistenza degli oggetti

Cancellazione degli oggetti - Approcci:

1. tramite un comando di cancellazione esplicito (es. Orion, Iris)

2. dal sistema quando non è più riferito da altri oggetti (es. GemStone, O2)

L'approccio 2. assicura l'integrità referenziale, ma necessita di un meccanismo di garbage collection

Persistenza degli oggetti

- Molti sistemi permettono di avere istanze persistenti e transienti di una stessa classe

- Le applicazioni accedono gli oggetti in modo uniforme, indipendentemente dal fatto che siano transienti o persistenti

- Si dice che la persistenza è una proprietà *ortogonale* (rispetto al tipo) degli oggetti

Persistenza degli oggetti

- Nei sistemi con persistenza per radice:

  – gli oggetti creati sono transienti

  – per renderli persistenti bisogna assegnare loro un nome o collegarli ad una radice

- Nei sistemi con persistenza automatica

  sono disponibili due varianti dell'operazione di *new* e un'operazione per rendere persistente un oggetto transiente

CLASSI

Estensione

- Nei sistemi con estensione non automatica l'utente mantiene generalmente l'estensione della classe
- esempio: classe `Persona`
  - si associa un nome esterno (es. `Persone`) ad un oggetto di tipo `Set(Persona)`
  - dopo ogni *new* su `Persona` si inserisce l'oggetto in `Persone`
  - per cancellare un oggetto, lo si rimuove da `Persone`

CLASSI

Attributi e metodi di classe

Caratterizzano la classe, intesa come un oggetto

Non si applicano alle istanze della classe, ma alla classe stessa

Attributi e metodi *statici* di Java

Esempio:

```
class Persona (nome, stipendio, eta´)
class-attribute max-stipendio
class-method trova_il_piu´_ricco () -> Persona
```


CLASSI

Metodi di classe - costruttori

- Esempio comune di metodi di classe: *costruttori*
- Metodi invocati al momento della creazione di un oggetto
  il body consiste nell'inizializzazione degli attributi
- Non hanno tipo di ritorno ed il nome coincide con quello della classe
- È possibile definire più costruttori per ogni classe
  (ovviamente con numero di argomenti diverso)

CLASSI

Migrazione degli oggetti

- In quasi tutti i sistemi il legame classe-oggetto è fisso: un oggetto rimane istanza della classe in cui è stato creato
- Poichè gli oggetti rappresentano entità del mondo reale dovrebbero essere in grado di rispecchiare l'evoluzione di tali entità

  Joe è studente
  Joe diventa assistente
  Joe diventa professore

CLASSI

Migrazione degli oggetti

La migrazione degli oggetti può essere pericolosa per l'integrità (comporta problemi semantici)

esempio:

- oggetto `c1` di tipo `Corso` si riferisce a oggetto corrispondente a Joe tramite attributo `studenti`
- se Joe non è più studente tale riferimento è scorretto ad es. non posso più accedere attributo `media` di Joe

EREDITARIETÀ

- L'**ereditarietà** è un importante meccanismo di riutilizzo del codice
- Permette ad una classe, detta *sottoclasse*, di essere definita a partire dalla definizione di una classe già esistente, detta *superclasse*
- La superclasse eredita attributi, messaggi e metodi dalla superclasse
  Può introdurre attributi, messaggi e metodi addizionali
  Può ridefinire (override) attributi, messaggi e metodi ereditati (con alcune restrizioni)

EREDITARIETÀ

Esempio

- Si considerino i seguenti tipi di oggetti

```
Camion                        Bus
n_licenza: String             n_licenza: String
produttore: String            produttore: String
ultima_revisione: Date        nro_posti: int
peso: int                     ultima_revisione: Date
valore(): int                 prox_revisione(): Date
prox_revisione(): Date
```


EREDITARIETÀ

Esempio - Continua

- Nel modello relazionale sono necessarie due tabelle e tre procedure
- Con l'approccio ad oggetti `Camion` e `Bus` sono riconosciuti essere veicoli
- Si introduce quindi una nuova classe `Veicolo` e le classi `Camion` e `Bus` sono definite come specializzazione di `Veicolo` è necessario definire solo le caratteristiche aggiuntive delle classi

EREDITARIETÀ

Esempio - Continua

```
        Veicolo
   n_licenza: String
   produttore: String
   ultima_revisione: Date
   prox_revisione(): Date


Camion: Veicolo      Bus: Veicolo
peso: int            nro_posti: int
valore(): int
```


EREDITARIETÀ

Ereditarietà - Vantaggi

- Evita ridondanza di codice
- Fornisce un potente meccanismo di progettazione

  le classi possono essere raffinate in più passi
- Permette una rappresentazione dello schema della basi di dati più concisa e meglio organizzata

EREDITARIETÀ

Ereditarietà - Sostituibilità

- Un'istanza di una sottoclasse può essere utilizzata ovunque ci si aspetti un'istanza della superclasse

  ad una variabile di tipo `Persona` può essere assegnato oggetto istanza della classe `Impiegato`
- Ogni variabile ha quindi
  - un *tipo statico*: tipo di cui è dichiarata
  - un *tipo dinamico*: classe più specifica dell'oggetto cui la variabile è istanziata

EREDITARIETÀ

Overriding

- Consideriamo i seguenti tipi di oggetti:

  `bitmap`, `window`, `impiegato` (record)

  e un'applicazione che debba visualizzare oggetti di tali tipi
- In un sistema convenzionale bisogna scrivere tre procedure

  `display_bitmap`, `display_window`, `display_impiegato`

EREDITARIETÀ

Overriding

```
for x in X do
    begin
      case of type(x)
         persona:  display_bitmap(x);
         figura:  display_window(x);
         grafo:  display_impiegato(x);
      end;
    end;
```


EREDITARIETÀ

Overriding

Nell'approccio ad oggetti:

- si definisce una classe generale (astratta) `Screen_Object` con tre sottoclassi: `bitmap`, `window`, `impiegato`
- si definisce un'operazione `display`
- in ogni sottoclasse si ridefinisce opportunamente l'operazione `display`

```
for x in X do x.display()
```


EREDITARIETÀ

Overloading

- Una conseguenza dell'overriding è che allo stesso nome di operazione corrispondono differenti implementazioni
- Nell'esempio l'operazione `display()` ha almeno tre implementazioni differenti in `bitmap`, `window`, `impiegato`
- L'overloading si può avere anche in assenza di ereditarietà (es. operazione =)

EREDITARIETÀ

Late binding

- L'overriding implica l'utilizzo del **late binding**
- Il metodo da utilizzare per rispondere ad un messaggio non può cioè essere deciso a compile time
- Un oggetto risponde ad un messaggio eseguendo il metodo più specifico, che non è necessariamente noto a compile time

EREDITARIETÀ

Estensibilità delle applicazioni

- È possibile costruire applicazioni che possono essere estese senza modificarne il codice
- Modifiche allo schema (aggiunta o cancellazione di una classe) non impattano il codice applicativo

  Se si devono visualizzare anche informazioni relative a studenti, basta definire una nuova classe `studente` come sottoclasse di `Screen_Object`
- Non si deve modificare nè ricompilare il codice che effettua la visualizzazione degli oggetti

EREDITARIETÀ

Method lookup (dispatching)

- È l'operazione effettuata dal sistema per determinare il metodo da eseguire per rispondere ad un messaggio
- Si determina la classe più specifica cui l'oggetto ricevente appartiene (il suo tipo dinamico)
- Si determina la superclasse più specifica di tale classe che fornisca un'implementazione per il metodo invocato (risalendo la gerarchia di ereditarietà)

EREDITARIETÀ

Method lookup (dispatching)

- Per ottimizzare questo processo i sistemi fanno generalmente uso di strutture ad-hoc (es. *dispatch tables*)
- I metodi per cui la selezione del codice da eseguire dipende solo dall'oggetto che riceve il messaggio sono detti metodi *selfish*

  l'oggetto ricevente è detto *distinguished parameter*

EREDITARIETÀ

Method lookup (dispatching)

- Quasi tutti i linguaggi di programmazione ad oggetti (Smalltalk, C++, Java) e tutti gli OODBMS sono basati sul dispatching singolo
- Per alcune operazioni (es. metodi binari) sembrerebbe più ragionevole tenere conto dei tipi dinamici di tutti gli argomenti ⇒ *dispatching multiplo* (*multimetodi*)

EREDITARIETÀ

Ridefinizione del dominio degli attributi

- Poichè il dominio rappresenta un vincolo di integrità non è ammissibile lasciarlo ridefinire arbitrariamente nelle sottoclassi

- Sembrerebbe ragionevole lasciar raffinare il dominio sostituendogli un sottotipo del dominio ereditato

  esempio: attributo `coniuge` ha dominio `Persona` in `Persona`, ha dominio `Nobile` in `Nobile`, sottoclasse di `Persona`

  (*ridefinizione covariante del dominio degli attributi*)

EREDITARIETÀ

Ridefinizione del dominio degli attributi

Tale ridefinizione non è però type-safe

esempio:

`setConiuge(Persona c) {coniuge = c}`

metodo di `Persona` non ridefinito in `Nobile`

`X` e `C` variabili di tipo statico `Persona`

`X.setConiuge(C)` è staticamente corretto

se però il tipo dinamico di `X` è `Nobile` e quello di `C` è `Persona` $\Rightarrow$ errore di tipo a run-time

EREDITARIETÀ

Ridefinizione del dominio degli attributi

Approcci:

- non ammettere la ridefinizione del dominio degli attributi

- distinguere tra attributi modificabili/non modificabili

- vedere il dominio come un vincolo di integrità e quindi controllarlo a run-time

In alcuni sistemi si può specificare nella sottoclasse un dominio diverso per un attributo ereditato – non è però ridefinizione ma *hiding*

EREDITARIETÀ

Ridefinizione della segnatura delle operazioni

- Per garantire la type-safety l'unica ridefinizione possibile è
  - specializzare i tipi degli argomenti di output (*covariante*)
  - generalizzare i tipi degli argomenti di input (*controvariante*)

- La ridefinizione controvariante dei tipi degli argomenti di input è contro-intuitiva

EREDITARIETÀ

Ridefinizione della segnatura delle operazioni

La ridefinizione covariante dei tipi degli argomenti di input non è type-safe

<u>esempio:</u>

`setConiuge(Persona c) {coniuge = c}`

metodo di `Persona`, ridefinito in `Nobile` in

`setConiuge(Nobile c) {coniuge = c}`

`X` e `C` variabili di tipo statico `Persona`

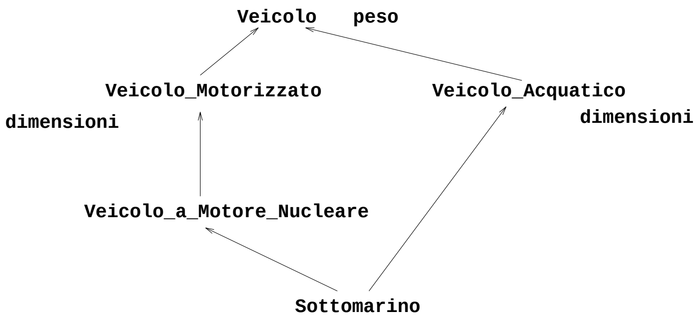
`X.setConiuge(C)` è staticamente corretto

se però il tipo dinamico di `X` è `Nobile` e quello di `C` è `Persona` ⇒ errore di tipo a run-time

EREDITARIETÀ

Ereditarietà multipla

Permette ad una classe di avere più superclassi



EREDITARIETÀ

Ereditarietà multipla

Risoluzione dei conflitti - alternative:

- *implicita basata su un ordinamento delle superclassi*

  le caratteristiche per cui vi è conflitto sono ereditate dalla prima superclasse

  se ad esempio `Sottomarino` è definito come
  `: Veicolo_a_Motore_Nucleare, Veicolo_Acquatico`
  le caratteristiche comuni (es. `dimensioni`) sono ereditate dalla classe `Veicolo_a_Motore_Nucleare`

EREDITARIETÀ

Ereditarietà multipla

Risoluzione dei conflitti - alternative:

- *esplicita*

  l'utente specifica da quali superclassi una caratteristica per cui c'è conflitto debba essere ereditatata
  ad esempio in `Sottomarino`:
  `attribute dimensioni` <u>`from`</u> `Veicolo_Acquatico`

EREDITARIETÀ

Ereditarietà multipla

Risoluzione dei conflitti - alternative:

- *impedire conflitti di nome*

  è possibile definire sottoclasse solo se le superclassi non hanno caratteristiche con nomi comuni

  problema del diamante

  ad esempio attributo `peso` in `Sottomarino`

EREDITARIETÀ

Diversi aspetti dell'ereditarietà

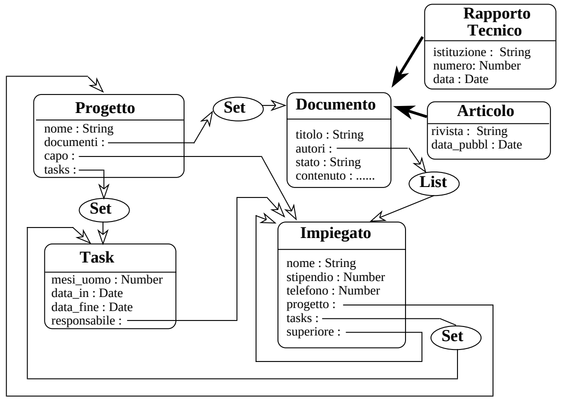
gerarchia di specializzazione (*subtype hierarchy*) consistenza fra le specifiche dei tipi - sostituibilità
comportamento degli oggetti come visto dall'esterno

gerarchia di implementazione condivisione del codice fra le classi - per definire l'implementazione di una classe per *differenza*

gerarchia di classificazione relazioni di inclusione tra collezioni di oggetti

MODELLO DEI DATI AD OGGETTI

Esempio di schema di base di dati



MODELLO DEI DATI AD OGGETTI

Differenze tra i modelli dei vari sistemi

- estensione delle classi
- persistenza
- cancellazione
- oggetti e valori
- accesso agli attributi
- ereditarietà multipla
- dominio degli attributi
- attributi e metodi di classe
- migrazione
- restrizioni su ridefinizione
- oggetti compositi
- associazioni
- istanze eccezionali
- classificazione multipla

MODELLO DEI DATI AD OGGETTI

Oggetti compositi

- La relazione di aggregazione non stabilisce alcuna semantica addizionale tra due oggetti
- Una relazione importante è la relazione **part-of**

  oggetto composito costituito da un insieme di oggetti componenti
- Vincoli di dipendenza ed esclusività

MODELLO DEI DATI AD OGGETTI

Associazioni

- Concetto caratteristico dei modelli semantici (ER)
- Rappresentano legami tra entità del dominio applicativo
- Hanno grado (numero di entità partecipanti all'associazione) e cardinalità (numero minimo e massimo di istanze dell'associazione a cui un'istanza dell'entità può partecipare)
- Possono essere rappresentate come attributi, ma a differenza degli attributi non sono direzionali

MODELLO DEI DATI AD OGGETTI

Istanze eccezionali

oggetti che hanno attributi e/o metodi aggiuntivi rispetto alla propria classe, o che ridefiniscono attributi e/o metodi della propria classe

Classificazione multipla

oggetti che appartengono a più classi più specifiche (es. `Impiegato` e `Studente`)

LO STANDARD ODMG

ODMG (Object Data[base] Management Group)

è uno dei working group di OMG, che consiste dei maggiori produttori di OODBMS (circa il 90% del mercato)

voting members: Object Design, Objectivity, Ontos, O2 Technology, Versant Object Technology

non-voting members: HP, Intellectic, Servio, Itasca, Texas Instruments

ODMG 3.0: JavaSoft, Lucent, POET Software, UniSQL, IBEX, GemStone Systems

LO STANDARD ODMG

OMG (Object Management Group)

associazione privata nata nel 1989 con lo scopo di promuovere l'uso di standard nell'area o-o

Data General, HP, Sun, Canon, American Airlines, Unisys, Philips, Prime, Gold Hill, Soft-Switch, 3 Com $+_{1991}$ AT&T, Digital, NCR, Bull, IBM, Olivetti

ODMG

Scopo del consorzio

- sviluppare una serie di standard per favorire portabilità, riusabilità e interoperabilità degli OODBMS commerciali
- successo dei RDBMS legato all'esistenza di standard, differenze tra i modelli dei vari OODBMS sono un ostacolo alla loro diffusione
- ODMG nel contesto o-o stesso ruolo di SQL in quello relazionale

ODMG

Risultati

- 1993: ODMG-93 standard [R. Cattell, The Object Database Standard: ODMG-93, Morgan-Kaufmann, 1993]
- 1997: ODMG 2.0 standard [R. Cattell et al., The Object Database Standard: ODMG 2.0, Morgan-Kaufmann, 1997]
- 1999: ODMG 3.0 standard [R. Cattell et al., The Object Database Standard: ODMG 3.0, Morgan-Kaufmann, 1999]

ODMG

Componenti

- *Object Model* (modello dei dati ad oggetti)
- *Object Definition Language* (ODL) - la base è l'interface definition language (IDL) di CORBA
- *Object Query Language* (OQL) - linguaggio di interrogazione dichiarativo - la base è SQL
- *Language Bindings*, per C++, Smalltalk, Java

ODMG 3.0 Object Model

è un modello basato sui concetti classici del paradigma o-o:

- le nozioni di base sono quelle di *oggetto* e *letterale*
- gli oggetti e i letterali sono suddivisi in base ai loro *tipi*

  tutti gli elementi di un certo tipo hanno le stesse proprietà e lo stesso comportamento
- i tipi sono organizzati in gerarchie di *ereditarietà*

ODMG 3.0 Object Model

Oggetti e Letterali

- ogni oggetto ha un unico identificatore, un letterale non ha identificatore
- i letterali sono costanti (immutabili) mentre gli oggetti possono essere modificati

ODMG 3.0 Object Model

Oggetti

- atomici
- collection

  (Set, Bag, List, Array, Dictionary)
- strutturati

  (Date, Interval, Time, Timestamp)

ODMG 3.0 Object Model

Letterali

- atomici

  (long, short, unsigned long, unsigned short, float, double, boolean, octet, char, string, enum)
- collection

  (set, bag, list, array, dictionary)
- strutturati

  (date, interval, time, timestamp + struct)

ODMG 3.0 Object Model

Oggetti

- identità: *object identifier* unico e immutabile
- *nomi*: uno o più, significativi per l'utente, unici nel db
- *lifetime*: oggetti transienti o persistenti
- *stato*: proprietà
- *comportamento*: operazioni

ODMG 3.0 Object Model

Oggetti: Proprietà

Lo stato di un oggetto è costituito dai valori per un insieme di *proprietà*

- *attributi* dell'oggetto

  i cui valori possono essere oggetti o letterali

- *associazioni* tra l'oggetto ed altri oggetti – simili ad associazioni nel modello ER

  solo associazioni binarie, cioè tra due tipi

  solo i tipi oggetto possono partecipare ad associazioni

ODMG 3.0 Object Model

Oggetti: Associazioni

- un'associazione viene dichiarata definendo una coppia di *traversal path*, uno per ogni direzione di attraversamento dell'associazione
- si possono dichiarare associazioni di tipo one-to-many e many-to-many mediante i tipi collezione set, list, bag
- l'OODBMS è responsabile del mantenimento dell'integrità referenziale per le associazioni

ODMG 3.0 Object Model

Oggetti: Associazioni - Esempio

```
class Professor { ...
  relationship set<Course> teaches
     inverse Course::is_taught_by;...};

class Course {...
  relationship Professor is_taught_by
     inverse Professor::teaches;...};
```

ODMG 3.0 Object Model

Oggetti: Attributi e Associazioni

La scelta di rappresentare un'informazione tramite un attributo con dominio una classe piuttosto che un'associazione tra le due classi non è sempre facile

differenze:

- direzionalità dell'attributo
- l'associazione garantisce integrità referenziale

ODMG 3.0 Object Model

Oggetti: Comportamento

- il comportamento di un oggetto è definito da un insieme di segnature di *operazioni* che possono essere eseguite su o dall'oggetto
- ODMG non si preoccupa dell'implementazione delle operazioni, che verrà specificata in uno dei linguaggi oo per cui esiste il binding
- nozioni usuali di overloading, overriding e late binding

ODMG 3.0 Object Model

Oggetti: Comportamento

- ogni segnatura definisce
  - il nome di un'operazione
  - il nome e il tipo dei suoi argomenti
  - i tipi dei valori restituiti
  - i nomi delle eccezioni che l'operazione può sollevare

ODMG 3.0 Object Model

Oggetti: Metodi - Esempio

```
add_enumeration (in string name,
                 in list<string> elementNames)
                 raises(DuplicateName, InvalidType)
```

ODMG 3.0 Object Model

Classi e Interfacce

- definizione di un object type: *specifica* + una o più *implementazioni*
- specifica: descrizione astratta e indipendente dall'implementazione delle operazioni, eccezioni e proprietà visibili all'utente

ODMG 3.0 Object Model

Classi e Interfacce

- due costrutti per definire la specifica:
  - *interfaccia* specifica solo il comportamento astratto
  - *classe* specifica comportamento e stato astratto
- attributi ed associazioni possono essere dichiarati in un'interfaccia, ma specificano comportamento (accessor methods)

ODMG 3.0 Object Model

Classi: Extent

- nella definizione di una classe può essere specificato il nome dell'extent corrispondente (al più uno)
- tutti gli oggetti istanza di una classe appartengono al suo extent
- l'extent di una classe include gli extent delle sue sottoclassi
- le interrogazioni sono formulate sugli extent

ODMG 3.0 Object Model

Classi: Extent

- in realtà ogni tipo ha un *extent*

  extent di un tipo = insieme delle sue istanze
- solo le classi possono avere extent a cui è assegnato un nome (quindi interrogabili)
- le interfacce non sono direttamente istanziabili

  il loro extent è costituito dall'unione degli extent delle sottoclassi

Classi: Chiavi

- se le istanze di una classe possono essere univocamente identificate dal valore di alcune proprietà è possibile definire tali proprietà come *chiavi*
- l'unicità dei valori di chiave è assicurata all'interno di un extent
- vincolo di integrità semantico, non utilizzato per l'identificazione degli oggetti (OID e nomi)

Ereditarietà

ODMG supporta due relazioni di ereditarietà:

ISA e *EXTENDS*

Gerarchia	*ISA*	*EXTENDS*
Ereditarietà di	comportamento	stato + comportamento
Ereditarietà Multipla	SI	NO
supertipo ↑ **sottotipo**	interfaccia ↑ classe o interfaccia	classe ↑ classe

Ereditarietà

- poichè ISA riguarda solo l'ereditarietà del comportamento, gli attributi e relazioni (accessor method) eventualmente presenti in un'interfaccia non sono ereditati dalle classi che ne ereditano
- tali attributi e relazioni devono però necessariamente essere inclusi nella classe

Ereditarietà

```
interface Student{
   attribute string name;
   attribute string student_id;
   relationship set<Section> takes
      inverse Section::is_taken_by;
   boolean reg_for_course(in Course c, in Section s);
   void drop_course(in Course c);
   short transfer(in Section old_s, in Section new_s);};

class Teaching_Assistant:Student{
   attribute string name;
   attribute string student_id;
   relationship set<Section> takes
      inverse Section::is_taken_by;
   relationship Section assists
      inverse Section::has_TA;};
```

ODMG 3.0 Object Model

Ereditarietà

- non è permessa ridefinizione dei domini degli attributi
- non è permessa ridefinizione della segnatura delle operazioni
- conflitti per ereditarietà multipla del comportamento: si impedisce name overloading

ODMG 3.0: ODL

Definizione di classi e interfacce

header
  {*Lista Attributi*
  Lista Associazioni
  Lista Metodi}

dove:

- *header* può essere
  - di interfaccia, della forma

    `interface` *Nome* [`:`*Lista Supertipi*]

ODMG 3.0: ODL

Definizione di classi e interfacce

- *header* può essere
  - di classe, della forma

    `class` *Nome* [`:`*Lista Supertipi*]
    [`EXTENDS` *Superclasse*]
    [(`extent` *Nome Extent*
    [`key[s]` *Lista Attributi*])]

ODMG 3.0: ODL

Definizione di attributi e relazioni

- ogni attributo nella lista è dichiarato come:

  `attribute` *Dominio* *Nome*

- ogni associazione nella lista è dichiarata come:

  `relationship` *Dominio* *Nome*
  `inverse` *Classe* :: *Nome Inv*

  dove *Dominio* può essere o *Classe*, o una collezione di elementi di *Classe*

ODMG 3.0: ODL

Definizione di metodi

- ogni metodo nella lista è dichiarato come:

  Tipo   *Nome*(*Lista Par*)
  [`raises` *Lista Eccezioni*]

  dove *Lista Par* è una lista di parametri specificati come

  `in` | `out` | `inout` *Tipo Nome*

ODMG 3.0: ODL

Esempio

```
class Impiegato
(   extent impiegati   key nome)
{   attribute string nome;
    attribute unsigned short stipendio;
    attribute unsigned short telefono[4];
    attribute Impiegato superiore;

    relationship Progetto progetto
       inverse Progetto::assegnati;
    relationship Progetto dirige
       inverse Progetto::capo;
    relationship Set<Task> tasks
       inverse Task::partecipanti;

    int premio(); }
```


ODMG 3.0: ODL

Esempio (segue)

```
class Documento
(   extent documenti   key titolo)
{   attribute string titolo;
    attribute List<Impiegato> autori;
    attribute string stato;
    attribute string contenuto; }

class Articolo EXTENDS Documento
(   extent articoli)
{   attribute string rivista;
    attribute date data_pubbli; }
```


ODMG 3.0: ODL

Esempio (segue)

```
class Progetto
(   extent progetti   key nome)
{   attribute string nome;
    attribute Set<Documento> documenti;
    attribute Set<Task> tasks;

    relationship Set<Impiegato> assegnati
       inverse Impiegato::progetto;
    relationship Impiegato capo
       inverse Impiegato::dirige; }
```

ODMG 3.0: ODL

Esempio (segue)

```
class Task
(   extent tasks)
{   attribute unsigned short mesi_uomo;
    attribute date data_in;
    attribute date data_fine;
    attribute Impiegato responsabile;

    relationship Set<Impiegato> partecipanti
       inverse Impiegato::tasks; }
```


ODMG 3.0: ODL

Esempio - 2

```
class Movie (extent Movies key (title, year)) {
  attribute string title;
  attribute short year;
  attribute short length;
  attribute enum Film {color,black&White} type;
  relationship Set<Star> stars
       inverse Star::starredIn;
  relationship Studio ownedBy
       inverse Studio::owns;
  relationship MovieExec producedBy
       inverse MovieExec::produces;
  void changeTypeOrYear();
  short starNbr();
  };
```


ODMG 3.0: ODL

Esempio - 2 (segue)

```
class Star (extent Stars) {
  attribute string name;
  attribute struct {string street, string city} address;
  attribute enum Gend {male, female} gender;
  relationship Set<Movie> starredIn
      inverse Movie::stars;
  relationship Set<Cartoon> voiceIn
      inverse Cartoon::voices;
  void confirmDel();
  };
```


ODMG 3.0: ODL

Esempio - 2 (segue)

```
class Studio (extent Studios keys address, name) {
  attribute string name;
  attribute string address;
  relationship MovieExec president
      inverse MovieExec::manages;
  relationship Set<Movie> owns
      inverse Movie::ownedBy;
  };
```

ODMG 3.0: ODL

Esempio - 2 (segue)

```
class MovieExec (extent MovieExecs) {
  attribute string name;
  attribute struct {string street, string city} address;
  attribute short netWorth;
  relationship Set<Movie> produces
      inverse Movie::producedBy;
  relationship Studio manages
      inverse Studio::president;
  };
```


ODMG 3.0: ODL

Esempio - 2 (segue)

```
class Cartoon extends Movie (extent Cartoons) {
  relationship Set<Star> voices
      inverse Star::voiceIn;
  };
```


PROGETTAZIONE DI SCHEMI AD OGGETTI

- Metodologie di progettazione ad oggetti (es. UML)

- La componente strutturale/statica (es. class diagrams) non è molto diversa dai diagrammi Entità-Relazione

PROGETTAZIONE DI SCHEMI AD OGGETTI

Principali differenze rispetto al relazionale

- Diverse modalità di identificazione (non è necessario introdurre codici se non semanticamente significativi per l'applicazione)

- Possibilità di rappresentare direttamente attributi multivalore e strutturati

- Ereditarietà e metodi

PROGETTAZIONE DI SCHEMI AD OGGETTI

Principali differenze rispetto al relazionale

- Diverse modalità per modellare le associazioni tra dati
  - direzionalità dei riferimenti
  - non tutti gli OODBMS supportano le associazioni
  - mancano comunque associazioni n-arie e associazioni con attributi

    (rappresentate in genere tramite classi)

ACCESSI AGLI OGGETTI - Diverse modalità

accesso navigazionale dato un OID il sistema accede direttamente (e in modo efficiente) all'oggetto riferito

possibilità di accedere agli oggetti navigando da uno all'altro

es. `X.progetto.capo.stipendio`

accesso associativo attraverso linguaggio di interrogazione

es. `select nome from Impiegato where stipendio > 2000`

accesso per nome tramite nomi esterni specificati dall'utente

es. `MioDoc.titolo`

ACCESSI AGLI OGGETTI - Accesso navigazionale

- l'accesso navigazionale è cruciale in molte applicazioni

  es. attraversamento di grafi

- sfrutta la gerarchia di aggregazione tra gli oggetti e la presenza di riferimenti espliciti (direzionali)

- nei sistemi relazionali è estremamente inefficiente perchè richiede molte operazioni di join

ACCESSI AGLI OGGETTI - Accesso associativo

- i linguaggi di interrogazione sono cruciali per lavorare su grandi quantità di oggetti

- l'avere a disposizione un linguaggio di interrogazione dichiarativo ad alto livello riduce i tempi di sviluppo delle applicazioni

- i linguaggi di interrogazione dichiarativi sono alla base del successo dei DBMS relazionali $\Rightarrow$ più importante caratteristica che gli OODBMS ne hanno ereditato

ACCESSI AGLI OGGETTI - Nomi esterni

- i nomi esterni forniscono agli utenti riferimenti semanticamente significativi agli oggetti
- i nomi esterni permettono di definire *entry point* nella base di dati: oggetti per cui è possibile accesso diretto
- molti OODBMS forniscono dizionari che permettono agli utenti di organizzare "object directories"

ACCESSI AGLI OGGETTI

- le varie modalità di accesso non sono esclusive, ma complementari
- esempio:
  - si seleziona un insieme di oggetti da una classe (o collezione) con un'interrogazione dichiarativa
  - si naviga a partire da ogni oggetto per visualizzare le sue componenti

LINGUAGGI DI INTERROGAZIONE

Caratteristiche principali

- uso di *path expressions*

  ```
  Progetto.capo.nome
  ```

- scope delle interrogazioni: singola classe o gerarchia di ereditarietà
- invocazione di metodi

  ```
  Select all from Veicoli
  where prox_revisione() > 10/11/1999
  ```


LINGUAGGI DI INTERROGAZIONE

- la maggioranza dei linguaggi di interrogazione ad oggetti sono estensioni dei linguaggi relazionali
- la maggiore ricchezza del modello dei dati introduce nuove problematiche

  es. *chiusura* del linguaggio di interrogazione

- mancanza di base formale (algebra/calcolo ad oggetti)
- nuove problematiche per l'ottimizzazione (metodi, tecniche di indicizzazione specializzate)

LINGUAGGI DI INTERROGAZIONE

- lo standard ODMG comprende un linguaggio di interrogazione dichiarativo - OQL - che è stato fortemente influenzato dal linguaggio di interrogazione di O_2

- molti OODBMS ODMG-compliant non implementano (ancora) OQL, o ne implementano solo un sottoinsieme

- una delle caratteristiche che distinguono un OODBMS da un *Persistent Object System* è proprio la presenza di un linguaggio di interrogazione dichiarativo

ODMG 3.0: OQL

- linguaggio di interrogazione SQL-like (select-from-where)

- linguaggio funzionale (gli operatori possono essere composti)

- i risultati di ogni interrogazione hanno un tipo che appartiene al sistema dei tipi di ODMG

- le interrogazioni possono essere annidate

- possibilità di riferire gli oggetti tramite nomi

ODMG 3.0: OQL - Path Expressions

- accesso navigazionale agli oggetti - *dot notation*

- possono coinvolgere: attributi, associazioni, metodi (con late binding)

- `myMovie` nome di oggetto di tipo `Movie`

  `myMovie.title`, `myMovie.stars`, `myMovie.starNbr()`

- possono avere lunghezza arbitraria

  `myMovie.ownedBy.president.address.city`

ODMG 3.0: OQL - SELECT-FROM-WHERE

- **select** seguito da una lista di espressioni

- **from** seguito da una lista di dichiarazioni di variabili

- **where** seguito da un'espressione a valori booleani, in cui possono comparire solo costanti e variabili dichiarate nella clausola **from**

ODMG 3.0: OQL - SELECT-FROM-WHERE

nella clausola **from** ogni variabile è dichiarata specificando

- un'espressione il cui valore ha tipo collezione (set o bag) - tipicamente l'extent di una classe
- opzionalmente la keyword **as**
- il nome della variabile

ODMG 3.0: OQL - Esempi

- determinare l'anno di *Via col vento*

```
select m.year
from Movies m
where m.title = "via col vento"
```

- determinare i nomi degli attori di *Casablanca*

```
select s.name
from Movies m, m.stars s
where m.title = "casablanca"
```


ODMG 3.0: OQL

Eliminazione dei duplicati

- le query viste producono un bag non un set come risultato
- possibilità di eliminare i duplicati con **select distinct**
- esempio determinare i nomi di tutte le star di film Disney

```
select distinct s.name
from Movies m, m.stars s
where m.ownedBy.name = "disney"
```


ODMG 3.0: OQL

Risultati di tipo strutturato

- le espressioni nella clausola **select** non devono necessariamente essere semplici variabili
- in particolare, possono contenere costruttori di tipo (es. `struct`)

Risultati di tipo strutturato

- esempio determinare le coppie di star che vivono allo stesso indirizzo

```
select distinct
  struct(star1: s1, star2: s2)
from Stars s1, Stars s2
where s1.address = s2.address
      and s1 <> s2
```


Sottointerrogazioni

- si può utilizzare un'espressione select-from-where (interrogazione) ovunque può apparire una collezione

- in particolare, le sottointerrogazioni possono apparire anche nella clausola **from**

Sottointerrogazioni

- esempio determinare i nomi delle star di film Disney (alternativa)

```
select distinct s.name
from (select m
      from Movies m
      where m.ownedBy.name = "disney") d,
      d.stars s
```


Ordinamento del risultato

- per rendere il risultato dell'interrogazione una lista, invece che un set o un bag, si può usare la clausola **order by**

- tale clausola, del tutto analoga a quella SQL, è seguita da una lista di espressioni

  (**asc** e **desc** come in SQL)

ODMG 3.0: OQL

Ordinamento del risultato

- esempio determinare i film Disney, ordinati in base alla lunghezza e (a parità di lunghezza) in ordine alfabetico

```
select m
from Movies m
where m.ownedBy.name = "disney"
order by m.length, m.title
```


ODMG 3.0: OQL

Quantificatori

- espressioni per testare se tutti, o almeno uno, i membri di una collezione soddisfano una condizione

for all x **in** S **:** $C(x)$

exists x **in** S **:** $C(x)$

- x variabile, S collezione, $C(x)$ condizione booleana sulla variabile x

ODMG 3.0: OQL

Quantificatori

- esempio determinare i nomi delle star di film Disney (alternativa 3)

```
select s.name
from Stars s
where exists m in s.starredIn:
          m.ownedBy.name = "disney"
```

nomi delle star che appaiono *solo* in film Disney **exists** $\rightarrow$ **forall**

ODMG 3.0: OQL

Espressioni aggregate

- OQL usa le stesse 5 funzioni aggregate di SQL: **avg**, **count**, **sum**, **min**, **max**

- si applicano a tutte le collezioni i cui membri siano di tipo appropriato

- esempio determinare la lunghezza media dei film

```
avg(select m.length from Movies m)
```

ODMG 3.0: OQL

Group-by

- la forma della clausola **group by** in OQL è
  - la keyword **group by**
  - una lista di *partition attributes*, separati da virgole - ognuno di questi ha la forma

    nome di campo : espressione

  la forma è cioè **group by** $f_1 : e_1, \ldots, f_n : e_n$

ODMG 3.0: OQL

Group-by

- il valore ottenuto dal raggruppamento è un insieme di strutture della forma

  $\mathtt{struct}(\mathtt{f}_1 : \mathtt{v}_1, \ldots, \mathtt{f}_n : \mathtt{v}_n, \mathtt{partition} : \mathtt{P})$

  dove P denota il bag di oggetti dell'interrogazione che appartengono al gruppo

- la clausola `select` di un'interrogazione che contiene **group by** può contenere solo i campi $\mathtt{f}_1, \ldots, \mathtt{f}_n$ e `partition`

ODMG 3.0: OQL

Group-by

- esempio determinare la lunghezza totale dei film prodotti da ogni studio in ogni anno

```
select std, yr,
       totLength: sum(select p.m.length
                      from partition p)
from Movies m
group by std: m.studio, yr: m.year
```


ODMG 3.0: OQL

Clausola Having

- come in SQL, per eliminare dei gruppi ottenuti con il raggruppamento, si può usare la clausola **having** seguita da una condizione, che si applica ai campi del gruppo

- esempio determinare la lunghezza totale dei film prodotti da ogni studio in ogni anno, per le coppie studio/anno tali che lo studio ha prodotto almeno un film più lungo di 120 in quell'anno

ODMG 3.0: OQL

Clausola Having

- esempio

```
select std, yr,
       totLength: sum(select p.m.length
                      from partition p)
from Movies m
group by std: m.studio, yr: m.year
having max(select p.m.length
           from partition p) > 120
```


ODMG 3.0: OQL

Operatori insiemistici

- è possibile applicare gli operatori insiemistici **union**, **intersect**, **except** a coppie di espressioni di tipo collezione

- esempio determinare i film con Harrison Ford che non sono stati prodotti da Disney

ODMG 3.0: OQL

Operatori insiemistici

- esempio

```
(select distinct m
 from Movies m, m.stars s
 where s.name = "Harrison Ford")
except
(select distinct m
 from Movies m
 where m.ownedBy.name = "Disney")
```


ODMG 3.0: OQL - Esempi

- determinare i task con almeno 20 mesi uomo il cui responsabile guadagna almeno 2000

```
select t from Tasks t
where t.mesi_uomo > 20 and
      t.responsabile.stipendio > 2000
```

il risultato è di tipo $\mathtt{bag} < \mathtt{Task} >$

ODMG 3.0: OQL - Esempi

- determinare la data di inizio dei task con almeno 20 mesi uomo

```
select distinct t.data_in
from Tasks t
where t.mesi_uomo > 20
```

il risultato è un letterale di tipo $\mathtt{set} < \mathtt{date} >$

ODMG 3.0: OQL - Esempi

- determinare la data di inizio e la data di fine dei task con almeno 20 mesi uomo

```
select distinct
  struct(di: t.data_in, df: t.data_fine)
from Tasks t
where t.mesi_uomo > 20
```

il risultato è un letterale di tipo
$\mathtt{set} < \mathtt{struct}(\mathtt{di} : \mathtt{date}, \mathtt{df} : \mathtt{date}) >$

ODMG 3.0: OQL - Esempi

- determinare la data di inizio e il responsabile dei task con almeno 20 mesi uomo

```
select distinct
  struct(di: t.data_in, r: t.responsabile)
from Tasks t
where t.mesi_uomo > 20
```

il risultato è un letterale di tipo
$\mathtt{set} < \mathtt{struct}(\mathtt{di} : \mathtt{date}, \mathtt{r} : \mathtt{Impiegato}) >$

ODMG 3.0: OQL - Esempi

- determinare la data di inizio, i nomi del responsabile e dei partecipanti dei task con almeno 20 mesi uomo

```
select distinct
  struct(di: t.data_in,
         nr: t.responsabile.nome,
         np: (select p.nome
              from t.partecipanti as p))
from Tasks t
where t.mesi_uomo > 20
```

$\mathtt{set} < \mathtt{struct}(\mathtt{di} : \mathtt{date}, \mathtt{nr} : \mathtt{string}, \mathtt{np} : \mathtt{bag} < \mathtt{string} >) >$

ODMG 3.0: OQL - Esempi

- determinare i rapporti tecnici che hanno lo stesso titolo di un articolo

```
select tr
from Rapporti_Tecnici tr, Articoli a
where tr.titolo = a.titolo
```


ODMG 3.0: OQL - Esempi

- determinare il nome ed il premio degli impiegati con stipendio superiore a 2000 e premio superiore a 500

```
select distinct
  struct(di: i.nome, p: i.premio())
from Impiegati i
where i.stipendio > 2000
      and i.premio() > 500
```


ODMG 3.0: OQL - Esempi

- determinare lo stipendio massimo dei responsabili dei task del progetto CAD

```
select max(select t.responsabile.stipendio
           from p.tasks t)
from Progetti p
where p.nome = ´CAD´
```


VERSIONI

- molte delle applicazioni avanzate per cui sono nati gli OODBMS sono di natura *esplorativa*

  ⇒ richiedono versioni di oggetti e alternative

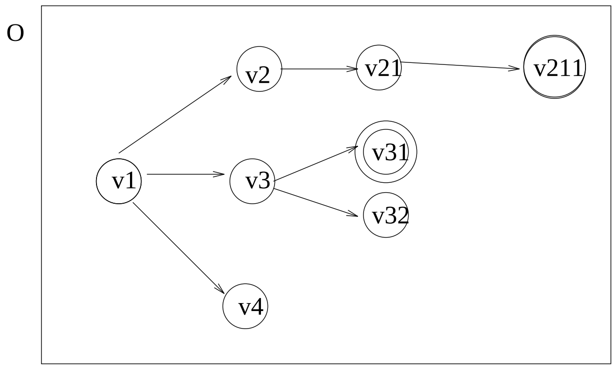
- in un modello con versioni, un oggetto O è una collezione logica di versioni tali che una versione V_i di O è ottenuta da un'altra versione V_k di O

- il fatto che V_i sia derivata da V_k stabilisce una gerarchia di versioni (albero di derivazione)

VERSIONI

- tutte le versioni in una gerarchia di versioni sono oggetti
- la gerarchia di versioni è mantenuta in un oggetto speciale, chiamato oggetto *generico*
- l'oggetto generico memorizza informazioni sulla gerarchia di versioni, ad esempio la versione di default

VERSIONI



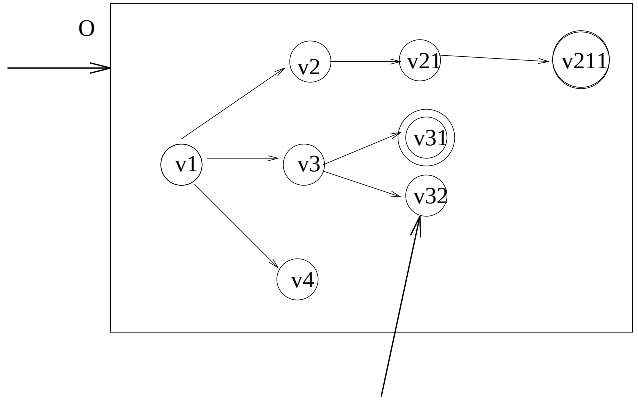
VERSIONI - Riferimenti

specifici (detti anche statici): riferimenti ad una specifica versione di un oggetto O

generici (detti anche dinamici): riferimenti all'oggetto generico associato ad O

è il sistema a selezionare la versione da restituire (in genere l'ultima che è stata generata)

VERSIONI - Riferimenti



VERSIONI

diversi OODBMS supportano meccanismi di versioni

- ORION/Itasca
  versioni transitorie, versioni di lavoro, versioni di consegna
  meccanismo di notifica dei cambiamenti
- Ode due relazioni tra le versioni: temporale e di derivazione
- Versant e O_2 solo nella versione C++

EVOLUZIONE DI SCHEMA

- gli ambienti applicativi degli OODBMS sono caratterizzati dal fatto che le modifiche allo schema sono frequenti
- ad esempio, il modo di classificare gli oggetti e le loro interrelazioni può evolvere durante il progetto di un'applicazione
- la complessità del modello dei dati ad oggetti aumenta le possibili modifiche di schema

EVOLUZIONE DI SCHEMA - Tassonomia

1. modifiche alla definizione di una classe

1.1 modifiche agli attributi

1.1.1 aggiunta di un attributo

1.1.2 cancellazione di un attributo

1.1.3 modifica al nome di un attributo

1.1.4 modifica al dominio di un attributo

1.1.5 modifica della classe da cui viene ereditato un attributo

1.2 modifiche ai metodi

EVOLUZIONE DI SCHEMA - Tassonomia

2 modifiche alla gerarchia di ereditarietà

2.1 modifiche agli archi

2.1.1 rendere una classe S superclasse di una classe C

2.1.2 rimuovere una classe S dalle superclassi di una classe C

2.2 modifiche ai nodi

2.2.1 aggiungere una nuova classe

2.2.2 cancellare una classe esistente

2.2.3 modificare il nome di una classe

EVOLUZIONE DI SCHEMA - Esempio di semantica

1.1.2 cancellazione di un attributo

- se un attributo A viene cancellato da una classe C, è cancellato da tutte le sottoclassi di C
- se esiste una superclasse C' con un attributo con lo stesso nome di A, tale attributo è ereditato e propagato alle sottoclassi di C
- se tale superclasse non esiste, l'attributo è cancellato da tutte le istanze della classe e delle sue sottoclassi

EVOLUZIONE DI SCHEMA - Strategie

un problema importante è come propagare le modifiche dello schema alle istanze delle classi

possibili approcci:

- propagazione immediata
- propagazione differita (lazy): gli oggetti sono modificati solo quando sono acceduti dall'applicazione
- versioni di classe con meccanismo di gestione di eccezioni

EVOLUZIONE

anche a livello di istanza le possibili modalità di evoluzione degli oggetti sono più che nel contesto relazionale (modifica al valore di un attributo)

- migrazioni dell'oggetto in classi diverse
- aggiunta dinamica di classi (nei modelli che supportano classificazione multipla)
- specializzazione a livello di istanza (nei modelli che supportano istanze eccezionali)

TECNICHE DI MEMORIZZAZIONE

un'organizzazione di memorizzazione per gli OODBMS deve fornire un supporto efficiente per

- oggetti con attributi sia atomici che complessi
- oggetti con attributi multivalore
- oggetti con attributi *large objects* (LOB)

non si possono utilizzare organizzazioni basate su file di record con lunghezza fissa

TECNICHE DI MEMORIZZAZIONE

due tipi di pattern di accesso

- accesso basato sull'intero oggetto
  manipolazioni complesse attraverso programmi applicativi

- accesso basato su attributi dell'oggetto
  ad esempio oggetti attraversati durante navigazione

TECNICHE DI MEMORIZZAZIONE

due approcci estremi

- modello diretto
  l'unità di memorizzazione coincide con l'unità semantica

- modello normalizzato
  oggetti decomposti in componenti atomiche memorizzate in file separati

approccio intermedio: si memorizzano nello stesso file componenti accedute contemporaneamente (si deve conoscere a priori pattern d'accesso)

TECNICHE DI MEMORIZZAZIONE

- possibilità di utilizzare *liste di proprietà*

- altro fattore da considerare: gerarchia di ereditarietà
  - unità di memorizzazione è l'intero oggetto (attributi propri della classe e attributi ereditati)
  - memorizzare oggetti separati, collegati tra loro, ciascuno dei quali memorizza le componenti ereditate da una superclasse

TECNICHE DI MEMORIZZAZIONE

Raggruppamento (Clustering)

- partizionare gli oggetti della base di dati e allocare tali partizioni su disco in modo da minimizzare il numero di operazioni di I/O su disco necessarie per il prelevamento

- fattori da considerare: struttura degli oggetti e pattern d'accesso

TECNICHE DI MEMORIZZAZIONE

Raggruppamento (Clustering)

- possibili strategie
  - memorizzare gli oggetti di una stessa classe contigui
  - memorizzazione depth-first della gerarchia di aggregazione: tutti i nodi discendenti di un nodo p sono memorizzati subito dopo p
  - raggruppare istanze che appartengono a una gerarchia di ereditarietà
- strategie statiche/dinamiche

INDICI

- la maggioranza dei join nelle interrogazioni a OODB sono join impliciti (navigazione di relazione di aggregazione)
- è possibile definire tecniche di indicizzazione che pre-calcolano i join impliciti
- esempi di tali tecniche: join indices, nested index, path index, access relations
- si hanno sia indici per identità che per uguaglianza

INDICI

- *nested index*: fornisce associazione diretta tra un oggetto di una classe alla fine di un path e le istanze corrispondenti della classe all'inizio del path
- *path index*: fornisce associazione tra un oggetto alla fine di un path e le istanziazioni del path che terminano in tale oggetto

INDICI

relativamente a gerarchie di ereditarietà

- indici *single-class*: data una gerarchia di ereditarietà con radice la classe C, si mantiene un indice separato per ogni classe nella gerarchia
- indici *class hierarchy*: data una gerarchia di ereditarietà con radice la classe C, si mantiene un indice comune per tutte le classi nella gerarchia

INDICI

- le interrogazioni a OODB possono contenere invocazioni di metodi (usati come attributi derivati o come predicati)

- l'esecuzione di un'interrogazione che contiene l'invocazione di un metodo può richiedere che il metodo sia eseguito su tutte le istanze di una classe

- una soluzione è quella di *precalcolare* il metodo e memorizzarne i risultati in un indice

ELABORAZIONE DI INTERROGAZIONI

- nei DBMS relazionali le interrogazioni vengono ottimizzate prima di essere eseguite
  - traduzione in formato interno (algebrico)
  - ottimizzazione algebrica
  - generazione dei piani di esecuzione
  - scelta del piano di esecuzione ottimale

ELABORAZIONE DI INTERROGAZIONI

- un'interrogazione su una singola classe è significativamente più potente di un'interrogazione su una singola relazione in una basi di dati relazionale
  - navigazione della gerarchia aggregazione
  - valutazione su gerarchia di ereditarietà

- un'interrogazione su una classe che coinvolge n classi tramite navigazione della gerarchia di aggregazione va vista come un'interrogazione con join di n relazioni

ELABORAZIONE DI INTERROGAZIONI

- si sono principalmente adattate le tecniche sviluppate per l'eborazione di interrogazioni relazionali

- alternative per visitare i nodi in un grafo delle interrogazioni: *forward*, *mixed*, *reverse* traversal

- alternative per accedere alle istanze delle classi: *nested-loop* e *sort-domain*

- principali differenze rispetto al modello relazionale: indici su attributi annidati, attributi multivalore, metodi

TRANSAZIONI

- il controllo della concorrenza e il ripristino sono due delle componenti principali della gestione delle transazioni
- il controllo della concorrenza assicura la correttezza dell'esecuzione concorrente di transazioni
- i meccanismi di ripristino assicurano la consistenza dello stato della base di dati anche a fronte di malfunzionamenti del sistema o errori nell'esecuzione di transazioni
- i sistemi di basi di dati sono visti come collezioni di oggetti che possono essere letti e scritti attraverso transazioni

TRANSAZIONI

- nel contesto ad oggetti il modello di esecuzione utilizzato per il controllo della concorrenza deve essere generalizzato
  - esecuzioni annidate di transazioni e modelli transazionali multilivello (azioni di compensazione)
  - si devono considerare operazioni arbitrarie sugli oggetti (non solo read-write tradizionali)
  - nozioni di metodi *compatibili*: gli effetti di uno non interferiscono con quelli dell'altro

TRANSAZIONI

- operazioni di *check-out*/*check-in* degli oggetti per permettere lunghe sessioni di lavoro senza bloccare gli oggetti
- tecniche di locking a granularità multiple (base di dati, segmento, file, record) sono particolarmente adatte al contesto ad oggetti
- lock per gerarchie di ereditarietà e oggetti complessi

AUTORIZZAZIONE

- ogni base di dati condivisa contiene dati a differenti livelli di sensitività
- il DBMS deve assicurare la privatezza dei dati permettendo ai dati di essere selettivamente condivisi tra molti utenti secondo diverse modalità
- il sottosistema di autorizzazione è la componente del DBMS incaricata di mantenere la privatezza dei dati
- il modello di autorizzazione deve essere consistente con il modello dei dati del DBMS

AUTORIZZAZIONE

- l'unità di controllo dell'accesso deve essere l'oggetto
- diversi livelli di granularità per le autorizzazioni
- autorizzazioni sia esplicite (specificate esplicitamente dall'utente e memorizzate) che implicite (derivate tramite regole) per il gran numero di oggetti in un OODB

AUTORIZZAZIONE

- esempi di operazioni considerate: READ, WRITE, GENERATE, READ DEFINITION (alcune sono applicabili solo a classi non ad istanze)
- necessità di tenere conto di gerarchie di ereditarietà, versioni, oggetti compositi
- accessi durante l'esecuzione di metodi

CENNI AD ALTRI OODBMS

- O_2
- GemStone
- Iris/OpenODB
- Orion/Itasca
- POET
- .... Jasmine (multimedia), Objectivity, Versant, Matisse ...

CENNI AD ALTRI OODBMS - O_2

- distingue oggetti e valori
- persistenza per radice (collezioni gestite dall'utente), no cancellazione esplicita
- linguaggio di interrogazione dichiarativo estensione di SQL ($\sim$ OQL)
- funzionalità di evoluzione, supporto per transazioni

CENNI AD ALTRI OODBMS - GemStone

- il modello dei dati e il linguaggio di accesso e manipolazione (OPAL) sono definiti come estensioni di Smalltalk
- approccio ad oggetti puro: tutto è visto come un oggetto
  (ad es. la definizione di una classe consiste nell'inviare un opportuno messaggio **subclass** ad una classe preesistente)
- gestione manuale dell'estensione (collezioni) e cancellazione via garbage collection
- messaggio di **select** per le interrogazioni, ma linguaggio di interrogazione molto limitato

CENNI AD ALTRI OODBMS - Iris/OpenODB

- origine da modelli semantici, forti influenze di linguaggi funzionali
- funzioni utilizzate per descrivere attributi degli oggetti (proprietà), definire metodi, definire associazioni
  possono essere implementate come stored tables, foreign functions, derived functions (interrogazioni)
- gestione automatica della persistenza, cancellazione esplicita
- linguaggio di interrogazione dichiarativo estensione di SQL

CENNI AD ALTRI OODBMS - Orion/Itasca

- nasce come estensione di Common Lisp
- nozione uniforme di classe, con estensione automatica e cancellazione esplicita
- diverse caratteristiche evolute: oggetti compositi, versioni, evoluzione e versioni di schema, meccanismo di autorizzazione, modello di transazioni
- linguaggio di interrogazione dichiarativo estensione di SQL

CENNI AD ALTRI OODBMS - POET

- come ObjectStore, permette memorizzazione persistente di oggetti Java e C++
- persistenza per raggiungibilità (collezioni) con cancellazione esplicita
- linguaggio di interrogazione dichiarativo, sottoinsieme significativo di OQL (no group by)
- modello di transazioni annidate, possibilità di alcune evoluzioni di schema

DIMENSIONI PER UN CONFRONTO

- architettura generale (piattaforme supportate, strutture di memorizzazione, caching, partizionamento)
- modello dei dati (attributi e associazioni, oggetti e valori, dati multimediali, collezioni e aggregati, oggetti complessi e compositi)
- aspetti procedurali e linguaggi di programmazione (late binding e polimorfismo, incapsulazione, controllo dei tipi)

DIMENSIONI PER UN CONFRONTO

- interrogazioni e linguaggio di interrogazione, ottimizzazione e metodi d'accesso
- evoluzione di schema e versioni
- concorrenza e ripristino - modello transazionale
- sicurezza e controllo degli accessi

DIMENSIONI PER UN CONFRONTO

- sistemi di basi di dati distribuite
- memorizzazione di dati web
- tools
- interfacciamento con DBMS esterni

CONCLUSIONI

- fine anni 80-inizio anni 90 uscita dei primi OODBMS
  - prodotti *rivoluzionari* cioè costruiti *from scratch*
  - fortemente influenzati da linguaggi di programmazione ad oggetti e fortemente contrapposti a DBMS relazionali
  - prodotti da piccole compagnie (non quelle che dominano il mercato dei DBMS)

CONCLUSIONI

- caratteristiche fondamentali:
  - ricchezza di strutture dati
  - classi e tipi definiti dall'utente
  - stretta integrazione con linguaggi di programmazione a oggetti
  - accesso ai dati navigazionale

- gli OODBMS si sono imposti in nicchie di mercato che non trovavano adeguato supporto dai DBMS relazionali (es. applicazioni CAD)

CONCLUSIONI

- tali DBMS non hanno avuto il successo di mercato sperato
  - più carenti per quanto riguarda le funzionalità DBMS dei consolidati prodotti relazionali
  - mancanza o limitatezza di accesso associativo ai dati
  - problema dei legacy system

- nel frattempo, i più diffusi DBMS relazionali (Informix, Sybase, DB2, Oracle) sono stati estesi con caratteristiche ad oggetti ⇒ DBMS **relazionali ad oggetti** o **universali**

- il modello dei dati dello standard SQL:1999 è object-relational

CONCLUSIONI

- gli OODBMS forniscono persistenza a oggetti creati in Java, C++, Smalltalk: estensione di un ambiente di programmazione ad oggetti

- gli ORDBMS (come i relazionali) introducono una API separata (basata su SQL) per lavorare con i dati memorizzati e hanno un loro sistema dei tipi che non è puramente object-oriented

- le architetture degli OODBMS sono tipicamente *client-centric* con il supporto di una cache di oggetti sulla macchina cliente e navigazione efficiente tra oggetti interrelati

- gli ORDBMS hanno tipicamente architettura *server-centric*

CONCLUSIONI

quando/perchè si dovrebbe preferire un OODBMS ad un ORDBMS?

- integrazione *seamless* con linguaggi di programmazione ad oggetti

- si privilegia accesso navigazionale ai dati

- si pensa di dover lavorare prevalentemente con tipi complessi definiti dall'utente, tipicamente non di natura tabellare

CONCLUSIONI

quando/perchè si dovrebbe preferire un OODBMS ad un ORDBMS?

- ci si aspetta di costruire applicazioni in cui è comune (e deve essere efficiente) l'accesso iterativo ad insiemi di oggetti
- si richiedono tools progettati primariamente per programmatori ad oggetti

RIFERIMENTI

- E. Bertino e L.D. Martino "Sistemi di Basi di Dati Orientate agli Oggetti", Addison-Wesley, 1992.
  (modello dei dati ad oggetti, aspetti architetturali, aspetti dinamici)
- E. Bertino, B. Catania, E. Ferrari, G. Guerrini "Sistemi di basi di dati - Concetti e architetture" UTET, Torino, 1997. Capitolo 4.

  E. Bertino e G. Guerrini "Object-Oriented Databases" In J. Webster, editore, *Wiley Encyclopedia of Electrical and Electronics Engineering*. John Wiley & Sons, 1999.

  (modello dei dati ad oggetti)

RIFERIMENTI

- E. Bertino e G. Guerrini "OODBMS History and Concepts" In A. Chaudhri e R. Zicari, editori, *Succeeding with Object Databases*. John Wiley & Sons, 2000.
  (panoramica storica e concetti fondamentali)
- R. Cattel et al. "The Object Database Standard: ODMG 3.0", Morgan-Kaufmann, 1999. (ODMG)
- ObjectStore Java API User Guide. (ObjectStore)

RIFERIMENTI

- J. L. Harrington "Object-Oriented Database Design Clearly Explained". Morgan-Kaufmann, 2000.
  (progettazione)
- D. Barry e J. Duhl "Object Storage Fact Book 4.1 - Object DBMSs", Barry & Associated, Inc., 2000.
  (confronto tra i vari sistemi)