UNIVERSITA' DEGLI STUDI DI PADOVA

FACOLTA' DI SCIENZE STATISTISTICHE

TESI DI LAUREA IN STATISTICA E TECNOLOGIE INFORMATICHE

CONCETTI DEL MODELLO RELAZIONALE AD OGGETTI

Relatore: Prof. MASSIMO MELUCCI

Laureanda: SABRINA ZARDO

ANNO ACCEDEMICO 2003-04

Indice

Introduzione

Il modello relazionale fu introdotto nel 1970 da E. Codd. Si basa sul concetto di relazione, o tabella, composta da righe (record) e da colonne (campi). Tutte le righe hanno la medesima struttura e ogni colonna ammette tipi semplici e predefiniti. Ogni tabella è indipendente dalle altre, le relazioni tra le tabelle sono costituite da campi comuni.

La struttura di questi sistemi ha permesso la realizzazione efficace ed efficiente di applicazioni di tipo gestionale in cui i dati manipolati sono semplici, raccolti in tabelle; negli RDBMS il linguaggio di interrogazione (SQL) è semplice, dichiarativo e standard, e tale caratteristica rende gli applicativi portabili su diverse piattaforme, quali Oracle, SQL Server, DB2, Sybase, Informix.

Gli sviluppi tecnologici hanno inoltre ottimizzato le prestazioni dei sistemi relazionali, ottenendo un buon livello di affidabilità, una buona gestione dell'accesso concorrente, architettura client-server e la gestione delle transazioni; i sistemi sono in grado di supportare un gran numero di utenti eseguendo il controllo sugli accessi, e abilitando o disabilitando l'accesso alle informazioni e alle operazioni eseguibili nella base dati tramite schemi e grant.

Il modello relazionale risulta, però, poco adatto a rappresentare informazioni complesse, come la gestione di sistemi CAD con figure geometriche assemblate insieme, o matrici per le applicazioni CAM per definire i movimenti di macchinari a controllo numerico, ecc. Le associazioni tra entità vengono modellate per valore e, nel caso di associazioni complesse, questa situazione richiede la creazione di parecchie tabelle/colonne fittizie.

Il linguaggio standard di definizione e interrogazione, SQL-92, racchiude in sé poche strutture dati; non comprende strutture annidate quali array e insiemi, e non è in grado di fornire tutti i possibili dati che potrebbero servire ad una applicazione per poter "mappare" gli oggetti del mondo reale nel modo più diretto possibile; assieme alle strutture complesse, nasce poi l'esigenza di gestire operazioni complesse per la necessità di trattare tali strutture, e di eseguire delle operazioni proprie per il tipo di dato e calcolare i tipi necessari all'utente. Le operazioni complesse inoltre necessitano di transazioni di lunga durata per garantire la consistenza dei dati.

Lei basi di dati orientate agli oggetti si propongono come una possibile soluzione alle limitazioni dei sistemi relazionali nella rappresentazione dei dati, sfruttando la ricchezza

delle strutture fornite dal paradigma object-oriented, e nella gestione di grandi quantità di dati, caratteristica tipica dei RDBMS.

La possibilità di modellare strutture complesse conduce alla costruzione e alla gestione di una architettura dati molto vicina allo schema concettuale del progetto; le fasi di sviluppo e di gestione si uniscono concentrando lo studio su un unico modello; si riducono così i problemi di comunicabilità derivanti dal passaggio dall'analisi, alla progettazione e alla programmazione.

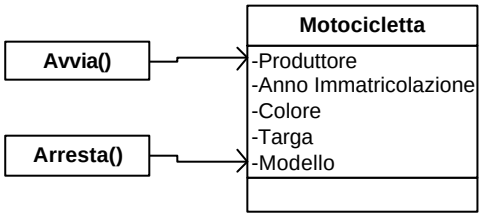
Il sistema OODBMS si presta a progetti che hanno necessità di tempi di sviluppo brevi, e di condivisione di informazione complessa, sviluppando sistemi intelligenti. Si sono imposti in nicchie di mercato non adeguatamente supportate dai sistemi relazionali. D'altra parte, questi sistemi non hanno avuto il successo di mercato sperato in quanto carenti in funzionalità rispetto ai sistemi relazionali, mentre questi ultimi hanno sviluppato funzionalità caratteristiche dei sistemi ad oggetti.

Capitolo 1
Il paradigma Object Oriented

1.1 Oggetti ed entità

L'object-Orientation è sempre più diffuso in ambito software engineering e nei linguaggi di programmazione, perchè espone una metodologia che permette di affrontare il problema da risolvere con un approccio verso la realtà, iniziando dall'osservazione degli oggetti reali, strutturando il problema in **oggetti**, che sono componenti elementari, entità rilevanti che descrivono il problema, e identificando per ogni entità proprietà e relazioni.

Oggetto



E' possibile definire l'approccio a oggetti secondo queste caratteristiche:

1. *Ogni cosa è un oggetto*. L'oggetto pensato come variabile particolare: memorizza l'informazione, risponde alle richieste; sugli oggetti possono essere sviluppate delle operazioni.
2. Un programma è un *insieme di oggetti che interagiscono* con l'invio di messaggi. Si può pensare all'invio di messaggi come esecuzioni di richieste attraverso le chiamate ai metodi dei vari oggetti coinvolti.
3. Ogni *oggetto ha la sua memoria*, e può a sua volta essere composto da altri oggetti. E' possibile creare nuovi tipi di oggetti componendoli con oggetti esistenti.
4. Ogni *oggetto ha un tipo*. Ogni oggetto è una istanza di una classe, la cui classe è sinonimo di tipo.
5. *Tutti gli oggetti di un particolare tipo possono ricevere la stessa richiesta*. Un

oggetto di tipo cerchio è anche un oggetto di tipo forma, pertanto tutti i messaggi per l'oggetto di tipo forma possono essere interpretati anche dall'oggetto di tipo cerchio.

L'oggetto viene definito attraverso lo **stato**, il **comportamento** e l'**identificativo**. Lo stato è rappresentato dai valori degli attributi dell'oggetto, la cui manipolazione viene eseguita tramite i **metodi** (che ne definiscono il comportamento) e ogni oggetto può essere distinto in modo univoco da tutti gli altri oggetti tramite un **OID** (Object Identifier).

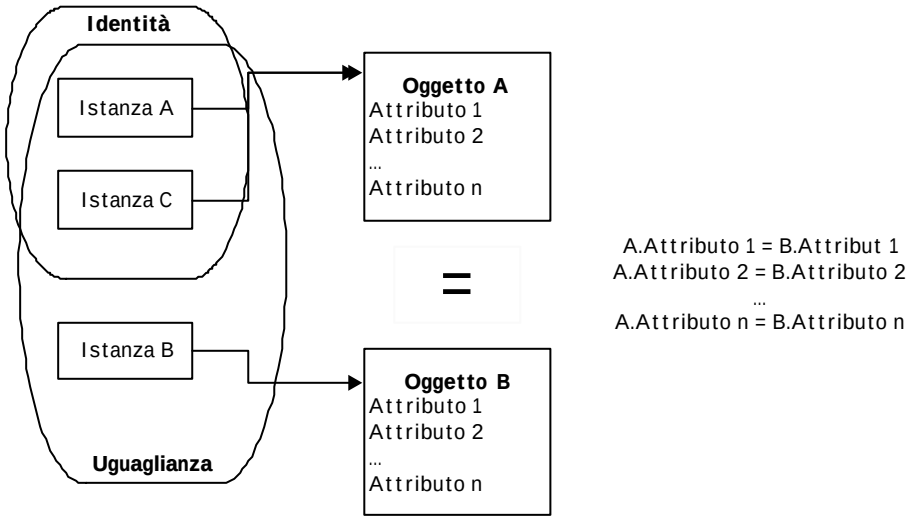
Non tutte le entità vengono rappresentate come oggetti; possono essere utilizzati anche i valori che si autoidentificano e non hanno OID. Tutte le entità primitive (interi, caratteri, ecc.) ricadono all'interno di questo gruppo e sono rappresentate da valori; le entità composte sono oggetti.

L'uguaglianza tra oggetti può presentare due situazioni diverse:

1. uguaglianza per **identità**, dove due oggetti sono identici se hanno lo stesso identificatore;
2. uguaglianza per **valore**,se due oggetti hanno valori uguali per tutti gli attributi.

L'uguaglianza per identità implica l'uguaglianza per valore, ma non viceversa.

Uguaglianza



Se un oggetto, oltre a comprendere tipi primitivi include ulteriori oggetti, viene definito **oggetto complesso**.

L'interfaccia stabilisce le richieste che possono essere fatte ad un particolare oggetto e comprende il codice necessario a soddisfarle, chiamata **implementazione**.

1.2 Istanziazione

La dichiarazione di una variabile stabilisce a quale oggetto può referenziare. La creazione dell'oggetto non coincide con la dichiarazione, ma avviene in modo esplicito a seguito della dichiarazione, fornendo le specifiche di tipo e tutti gli argomenti necessari per la sua costruzione. In questo modo il sistema viene messo in grado di allocare lo spazio che serve per memorizzare gli attributi dell'oggetto, essendo in possesso di tutte le informazioni necessarie allo scopo.

L'operazione di creazione degli oggetti viene indicata con il termine di **istanziazione:** gli oggetti vengono memorizzati all'interno dell'area di memoria di sistema, nota con il nome di *heap,* e in risposta all'operazione di creazione viene restituito il **riferimento all'oggetto** (**OID**), indicatore utilizzato per riferirsi all'oggetto creato.

Gli attributi presenti all'interno di un oggetto sono variabili dell'istanza, poiché per ogni oggetto definito è presente una copia distinta degli attributi.

Gli oggetti non vengono mai cancellati esplicitamente: nel momento in cui non servono più è sufficiente non referenziarli, vengono così acquisiti dal **garbage collector**, che agendo indipendentemente rispetto all'esecuzione e tenendo traccia di tutti i riferimenti agli oggetti, li rimuove dallo spazio allocato nello heap. La cancellazione dell'oggetto avviene quando il garbage collector decide di intervenire, ad esempio quando il sistema non riesce a trovare sufficiente spazio libero per allocare un'ulteriore oggetto e sorge quindi la necessità di recuperarne.

1.3 Tipi, Classi, Attributi e Metodi

I **metodi** definiscono il comportamento degli oggetti di una certa classe. La programmazione orientata agli oggetti distingue nettamente la nozione di cosa va fatto da quella di come lo si fa. Il **cosa** è descritto sottoforma di insieme di metodi e dalla relativa semantica. E' la combinazione di metodi, dati e semantica che permette all'utilizzatore di

sapere cosa accade quando i metodi specificati vengono invocati su di un oggetto.

La dichiarazione di un metodo è composta da due parti: l'intestazione del metodo (**signature**) e il corpo del metodo (**body**). L'intestazione del metodo è costituita dal nome del metodo e dalla lista (anche vuota) dei tipi di parametri, sia di input che di output. Il corpo del metodo è costituito invece dalle istruzioni.

Due metodi possono avere lo stesso nome, lo stesso insieme di parametri e sollevare le stesse eccezioni, ma non possono essere ritenuti equivalenti se dotati di semantica differente.

Il **come** effettuare qualcosa per un oggetto è definito dalla sua classe di appartenenza, la quale fornisce le implementazioni dei metodi che l'oggetto supporta.

L'idea che tutti gli oggetti, anche se unici, possono essere raggruppati in **classi** in quanto possessori di caratteristiche e comportamenti comuni, fu sviluppato nel primo linguaggio object-oriented Simula-67, che introdusse appunto il termine "classe" nei programmi.

Oggetti identici appartengono a "classi di oggetti", dove l'identità non coinvolge lo stato dell'oggetto.

Ogni oggetto ha una classe che ne definisce i dati e il comportamento.

La classe è particolare tipo di dato che descrive un insieme di oggetti con caratteristiche e comportamenti identici ed è composta da questi tre elementi:

- gli **attributi**, variabili che contengono i dati associati alla classe e ai suoi oggetti, con lo scopo di memorizzare i risultati dei calcoli effettuati all'interno della classe;
- i **metodi**, che contengono il codice eseguibile della classe e sono costituiti da istruzioni (il modo in cui vengono invocati, e quindi il modo in cui le istruzioni al loro interno vengono eseguite, è quello che dirige l'esecuzione del programma);
- la **classe** e le **interfacce**, che possono essere a loro volta membri di altre classi o interfacce.

I membri di una classe possono avere diversi livelli di visibilità o accessibilità: possono essere definiti attributi che possono essere letti e modificati; oppure è possibile limitare l'accesso dei membri a porzioni di codice della classe stessa o di altre classi correlate.

Ogni oggetto è una istanza di una classe. Quando viene invocato un metodo su un oggetto, si esamina la classe a cui esso appartiene per cercare il codice da eseguire.

Un **attributo** può essere inizializzato al momento della sua dichiarazione, assegnandogli un valore del tipo corrispondente; tale valore può essere semplice

(costante), oppure costituito da un altro attributo, o da un'invocazione di metodo o da una espressione che comprenda entrambi. L'unica necessità è che l'inizializzatore restituisca lo stesso tipo di attributo; nel caso in cui non avvenga alcuna inizializzazione, il valore di default iniziale assegnato all'attributo dipende dal tipo.

1.4 Incapsulazione: Interfaccia e implementazione

L'introduzione delle classi ha permesso di evidenziare nuovi aspetti sulla visibilità dei dati trattati. Generalmente gli oggetti non operano direttamente sui dati di altri oggetti, anche se una classe rende i suoi attributi pubblici, cioè accessibili a tutti. Le classi progettate correttamente, nascondono i loro dati in modo che questi possano essere modificati esclusivamente accedendovi con i metodi della classe stessa.

Le classi **definiscono la struttura degli oggetti** con meccanismi con i quali questi possono essere costruiti a partire dalla definizione, e i **metodi**, insiemi di parti di codice eseguibile, che costituiscono il nucleo della computazione e che manipolano i dati memorizzati all'interno degli oggetti.

Uno dei punti di forza della tecnologia orientata agli oggetti è il supporto che essa fornisce nell'**incapsulamento** e nell'occultamento dei dati (data hiding): l'oggetto è nascosto mentre sono visibili solo le operazioni. Il controllo dell'accesso ai dati all'interno delle classi viene gestito nella dichiarazione di classe e interfaccia. All'interno di un metodo è possibile riferirsi direttamente agli attributi e ai metodi della classe; il riferimento alla variabile è diretto, non è necessario ricorrere a un riferimento dell'oggetto.

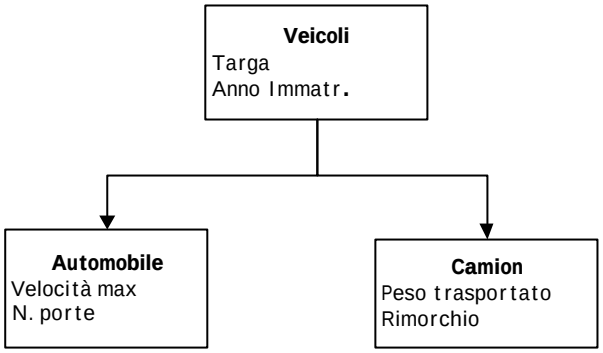
Nella progettazione di una classe si ha a volte la necessità di dichiarare i metodi che un oggetto deve supportare senza fornirne l'implementazione. Questa tipologia di classe è detta **classe astratta**, e contiene solo le dichiarazioni dei metodi supportati, definendo solo l'interfaccia. In questa tipologia di classe non è possibile includere l'implementazione. La classe astratta viene utilizzata per implementare l'overriding.

1.5 Estensione delle classi

L'**ereditarietà** permette di definire una classe da una esistente. Questa nuova classe viene chiamata *sottoclasse*, riceve attributi, metodi e messaggi alla classe generatrice,

definita *superclasse*. La sottoclasse può sviluppare ulteriori attributi e metodi; se la superclasse modifica i suoi metodi e attributi, anche la sottoclasse viene coinvolta in tali operazioni.

Ereditarietà



La classe descrive una entità della realtà nei tipi e nei comportamenti. Vi sono casi in cui i tipi sono simili, e differiscono per poche caratteristiche aggiuntive o per alcuni comportamenti leggermente diversi. Tramite l'ereditarietà è possibile rappresentare questo concetto, e la similarità tra i tipi viene espressa introducendo i tipi base e i tipi derivati. Un **tipo base** contiene tutte le caratteristiche e i comportamenti che sono condivisi con i suoi **tipi derivati.** Si crea un tipo base per rappresentare il nucleo di una idea di un oggetto nel sistema e da esso si derivano gli altri tipi per esprimere in modi differenti il concetto che si vuole realizzare.

Un tipo definito tramite l'ereditarietà, oltre a contenere tutti i membri del tipo base, sia quelli privati che quelli nascosti, eredita anche l'interfaccia della classe base. Questo vuol dire che tutte le chiamate alla classe base possono essere inoltrate anche alla classe derivata; quindi la classe derivata è dello stesso tipo della classe base.

Si parla di ereditarietà **multipla** quando una classe ha più superclassi; ereditarietà **semplice** quando la classe possiede una sola superclasse.

L'aspetto essenziale dell'ereditarietà è la relazione che si viene a creare tra le classi, considerando il case che una sottoclasse può essere a sua volta superclasse, si parla di **gerarchia di ereditarietà**.

Le classi derivate possono essere differenziate dalla classe base secondo due tipologie:

- nella classe derivata viene aggiunto almeno un metodo in più rispetto alla classe

base: **estensione della classe**;

- almeno un metodo della classe base viene modificato: l'**overriding** del metodo. La classe estesa definisce un metodo con la stessa signature e lo stesso tipo di ritorno di un metodo della superclasse.

Quando si estende una classe per aggiungervi nuove funzionalità, viene creata la relazione comunemente nota relazione IsA: l'estensione crea un nuovo tipo di oggetto che "è un" tipo simile all'originale. La relazione IsA è diversa dalla relzione HaS "ha un", nella quale un oggetto ne utilizza un altro per poter memorizzare parte dello stato o per effettuare un certo compito, esso cioè "ha un" riferimento all'altro oggetto.

Un metodo può essere ridefinito solo se è accessibile. Se il metodo non è accessibile, non può essere ereditato, e se non è ereditato, non può essere ridefinito.

Il **polimorfismo** nel paradigma ad oggetti, sfrutta l'estensione delle classi per poter assegnare all'oggetto il metodo appropriato, identificabile solo a runtime. L'oggetto può appartenere sia alla sua classe vera e propria, che ad una qualsiasi altra classe estensione della precedente.

Il meccanismo di estensione di una classe viene utilizzato per ottenere una specializzazione, nei casi in cui la classe estesa definisca un nuovo comportamento e quindi costituisca una versione più specializzata della sua superclasse. L'estensione di una classe può quindi comportare unicamente una modifica alle implementazioni dei metodi ereditati per renderli eventualmente più efficienti.

Per una classe è possibile disporre di metodi statici, noti come **metodi di classe**. Questi metodi implementano operazioni tipiche della classe stessa, operano sugli attributi statici e non su istanze specifiche della classe; qando vengono invocati, non riferiscono ad un un oggetto specifico su cui operare, come invece agiscono i metodi normali.

Le classi sono dotate di costruttori, ovvero blocchi di istruzioni utilizzabili per inizializzare un oggetto prima che restituisca un riferimento ad esso. I costruttori hanno lo stesso nome della classe che inizializzano.

I motivi che spingono a fornire costruttori specializzati sono i seguenti:

- alcune classi possono non avere uno stato iniziale ragionevole se prive di parametri;

- fornire uno stato iniziale è al tempo stesso ragionevole e conveniente nel momento in cui si costruiscono certi tipi di oggetti.

- costruire un oggetto è una operazione costosa, quindi si vuole che gli oggetti siano dotati di uno stato iniziale corretto subito dopo la creazione.

- un costruttore che non è pubblico permette di limitare il numero di coloro che

possono creare oggetti attraverso di esso.

Se in una classe non si forniscono costruttori di alcun tipo, il sistema utilizza un costruttore di tipo standard che non fa nulla: costruttore di default.

Un'altra forma di costruttori è il costruttore copia, che richiede come argomento un oggetto del tipo corrente e ne costruisce uno nuovo che ne è una copia.

1.6 Overriding, Overloading, e late binding

Con l'introduzione delle classi estese e del polimorfismo, è possibile definire metodi che manipolano oggetti senza sapere di che tipo sono. In questo modo è possibile gestire codice che non dipende da uno specifico tipo. Il problema che sorge è: come fa il sistema a sapere nel momento della compilazione che codice considerare se non sa che tipo di oggetto dovrà trattare? Le funzioni chiamate generate da un sistema non OOP, sono dette "early binding": significa che il compilatore genera una chiamata ad una specifica funzione ed il linker risolve la chiamata. Il paradigma ad oggetti usa il concetto di "**late binding**". Quando viene richiamato un metodo, il codice non determina la funzione finchè non è a runtime. Il compilatore assicura che il metodo esiste e definisce il tipo dell'argomento di ritorno, ma non sa quale sarà il codice che dovrà eseguire.

Ogni metodo è dotato di una firma, signature, costituita dal nome, dal numero e tipo dei parametri. Due metodi possono avere lo stesso nome nel caso in cui siano dotati di firme diverse, ovvero di un diverso numero o tipo di parametri: **overloading**. Quando si invoca il metodo il compilatore confronta il numero e il tipo degli argomenti per determinare il metodo che corrisponde alla particolare firma.

L'**overriding** di un metodo, invece significa sostituirne l'implementazione in modo tale che nel momento in cui il metodo viene invocato su di un oggetto della sottoclasse, ne venga invocata la versione effettivamente definita dalla sottoclasse. Quando si stanno ridefinendo dei metodi, le firme e i tipi di dato restituiti devono essere gli stessi del metodo corrispondente nella superclasse.

Capitolo 2
Basi di Dati a Oggetti

2.1 Modello base

Una base dati Object Oriented è un sistema con funzionalità e caratteristiche di un linguaggio di programmazione ad oggetti calato della base dati, che integra la tecnologia delle basi dati con il paradigma ad oggetti, offrendo la possibilità di rendere persistenti i dati che gestisce, fornendo funzionalità di accesso e ricerca degli stessi , permettendo così di utilizzare un unico modello object oriented per strutturare le informazioni. Questo modello, composto da strutture complesse di dato, viene rappresentato all'interno della base dati. Viene superato il problema dell'impedence mismatch, presente nei sistemi relazionali, costituito dalla differenza concettuale tra i modelli di rappresentazione dei dati usati dall'applicazione e dal modello memorizzato nella base dati. Questa limitazione comporta l'implementazione di algoritmi per convertire i dati tra un modello e l'altro.

La prima generazione di Object DBMS fu sviluppata tramite linguaggi di programmazione ad oggetti persistenti e realizzava solo alcune caratterisitiche della base dati, senza la funzionalità dell'interrogazione, e incompatibile con i modelli relazionali.

Gli OODBMS della seconda generazione, oltre ad offrire un maggior numero di caratteristiche, sono provvisti del supporto all'interrogazione dei dati.

Gli OODBMS riconoscono tipi di dato strutturato, oggetti complessi, tipi di dato astratto (ADT) specifici dell'applicazione, e tipi di dato non strutturato (Blobs); modellano direttamente oggetti complessi e le loro associazioni, hanno buone prestazioni per applicazioni navigazionali, limitato supporto per concorrenza, parallelismo e distribuzione. Forniscono semplici modelli transazionali, con limitate funzionalità nel controllo dell'accesso e coprono un mercato di nicchia che richiede accessi navigazionali efficienti.

2.2 Oggetti e valori

Le entità nel sistema sono rappresentate come oggetto, o come valore. I valori sono l'entità standard conosciute da ogni utente e sono built-in nel sistema; gli esempi tipici di

valori sono gli interi, reali, booleani, caratteri e stringhe. Alcuni sistemi possiedono poi **valori strutturati**: insiemi, liste, tuple, array; i valori strutturati possono contenere oggetti come componenti.

La struttura dell'oggetto dipende dall'applicazione e deve essere definita.

L'informazione è rappresentata da un valore, se è se stessa; l'informazione di un oggetto invece è rappresentata dalle relazioni con gli altri oggetti e valori. I valori sono utilizzati per descrivere altre entità, mentre gli oggetti sono le entità che devono essere descritte.

Un oggetto consiste di un identificatore, uno stato o valore, costituito per un certo numero di attributi (campi) che possono a sua volta contenere riferimenti ad altri oggetti, e di un comportamento come insieme di metodi o operazioni.

L'accesso ai componenti di un oggetto avviene con la sintassi:

oggetto.attributo
oggetto.metodo

2.3 Oggetti complessi

Gli oggetti complessi sono generati a partire da oggetti atomici mediante costruttori e possono a loro volta essere componenti di altri oggetti. I vantaggi di un modello che fornisce oggetti complessi sono evidenti nella rappresentazione diretta di oggetti strutturati quali mappe, disegni CAD, ecc., strutture complesse che in questa gestione non necessitano di essere decomposti in unità più piccole, tuple o record. La ricerca e la gestione di tali struttura risulta inoltre più veloce.

Nella categoria degli oggetti complessi appartengono anche i tipi di dati di grandi dimensioni, non strutturati, che il sistema non è capace di interpretare, si pensi all'informazione multimediale. Un esempio è dato dai campi di tipo BLOB (bynary large object) che permettono di memorizzare valori di grandi dimensioni, quali immagini di tipo bitmap o lunghe stringhe di testo. Sono definiti non strutturati perché al sistema non è nota la struttura, ma è l'applicazione che li gestisce che è in grado di interpretarli.

2.4 Object Identifier Definition

Ogni oggetto possiede un identificatore (OID) che lo distingue da tutti gli altri all'interno della base dati. Tale identificatore è univoco, immutabile e indipendente dal valore dell'oggetto. L'oggetto può essere definito come coppia (OID, valore). Gli OID, inoltre, sono trasparenti all'utente, non sono visibili. Le differenze tra OID e chiavi del sistema relazionale sono:

- univocità, l'OID è univoco nel sistema, la chiave è univoca nella relazione;
- l'OID è immutabile perché indipendente dal valore dell'oggetto, mentre la chiave fa parte del valore dell'oggetto essendo composta da uno o più attributi;
- l'OID è gestito dal sistema, la chiave è definita dall'utente ed è l'utente che ne assicura l'univocità.
- Il concetto di OID introduce due tipi di uguaglianza tra oggetti:
- identità, quando due oggetti possiedono lo stesso OID;
- uguaglianza, quando due oggetti possiedono lo stesso stato (gli attributi hanno lo stesso valore).

Utilizzando lo stesso OID, è possibile condividere gli oggetti, in tal caso lo stato di entrambi gli oggetti è uguale e i cambiamenti alle componenti di un oggetto si ripercuotono anche su tutti gli altri oggetti che li riferiscono.

2.5 Incapsulazione

Ogni metodo ha sempre un parametro implicito che corrisponde all'oggetto sul quale il metodo viene invocato. L'implementazione delle operazioni è nascosta, non visibile dall'esterno.

L'interfaccia di un oggetto è l'insieme delle signature delle operazioni, definisce a quali richieste l'oggetto risponde e le interazione dell'oggetto con il mondo esterno.

La progettazione comune di dati e operazioni all'interno dello stesso sistema, risolve il problema dell'impedence mismatch presente in SQL, dove per avere una completa gestione di tutte le eventuali problematiche da risolvere, è necessario avere almeno due linguaggi di programmazione: uno per gestire i dati, l'altro per gestire il codice. Con OODBMS il linguaggio utilizzabile è unico, permette di definire operazioni mediante i metodi associati agli oggetti, l'intera applicazione può essere sviluppata in termini di oggetti.

Un metodo viene invocato inviando un messaggio ad un oggetto; è possibile che due oggetti di due classi differenti possano esibire comportamenti differenti allo stesso messaggio, overloading.

Per quanto riguarda l'incapsulazione, possiamo distinguere l'incapsulazione stretta dove i valori degli attributi di un oggetto possono essere letti e scritti solo tramite i metodi accessor (getAttribute) e mutator (setAttribute); e l'incapsulazione non stretta nella quale l'accesso ai valori degli attributi è possibile anche con modalità diretta.

Il metodo **accessor** restituisce il valore associato ad un attributo di un oggetto, mentre il metodo **mutator** modifica il valore dell'attributo.

L'istanziazione è un meccanismo che permette di generare oggetti. Un insieme di oggetti simili appartengono alla stessa classe.

I costruttori sono i metodi invocati al momento della creazione di un oggetto. Il corpo esegue l'inizializzazione degli attributi, non hanno tipo di ritorno ed il nome coincide con quello della classe.

2.6 Classi e tipi

Una classe specifica lo scopo delle sue istanze definendo:

- una struttura, un insieme di attributi;
- un insieme di messaggi, interfaccia esterna degli oggetti;
- un insieme di metodi, invocati dai messaggi.

La classe definisce le primitive per la creazione di oggetti, e l'interfaccia stabilisce le specifiche del comportamento esterno di un insieme di oggetti.

Per ogni attributo viene specificato un dominio, cioè l'intervallo dei possibili valori dell'attributo. La definizione di una classe include anche la specifica del dominio degli attributi. Se la classe C è il dominio dell'attributo A di una classe C', si dice che c'è una relazione di aggregazione (o clientship) tra C' e C. La gerarchia di aggregazione può contenere cicli, un dominio di un attributo può essere la classe stessa. I cicli possono avere lunghezza arbitraria.

Le classi supportano sempre un metodo di creazione (new) per creare gli oggetti, il metodo di creazione è un metodo di classe.

Gli attributi e i metodi di classe caratterizzano la classe intesa come un oggetto, non si applicano alle istanze della classe, ma alla classe stessa.

Tramite l'ereditarietà è possibile riutilizzare il codice, la sottoclasse viene definita a partire dalla definizione di una classe già esistente, superclasse. La sottoclasse eredita attributi, messaggi e metodi dalla superclasse, e può introdurre attributi, messaggi e metodi addizionali. Può ridefinire attributi, messaggi e metodi ereditati, override.

Esempio:

Camion	Bus
N_licenza: string	N_licenza: string
Produttore: string	Produttore: string
Ultima_revisione:date	Ultima_revisione: date
Peso: int	Prox_revisione(): date
Valore(): int	
Prox_revisione(): date	

Nel modello relazionale per definire lo schema richiesto sono necessarie due tabelle e tre procedure. Con l'approccio ad oggetti, camion e bus sono riconosciuti veicoli. Si introduce quindi una nuova classe veicolo e le classi camion e bus sono definite come specializzazione di veicolo. E' necessario pertanto definire solo le caratteristiche aggiuntive delle classi.

Veicolo	
N_licenza:	string
Produttore:	string
Ultima_revisione:	date
Prox_revisione():	date
Camion:	**Veicolo**
Peso:	Int
Valore():	Int
Bus :	**Veicolo**
Nro_posti:	int

L'ereditarietà evita la ridondanza di codice e fornisce un potente meccanismo di progettazione. Le classi possono essere definite in più passi e si ottiene una

rappresentazione dello schema della basi di dati più concisa e meglio organizzata.

Un'istanza di una sottoclasse può essere utilizzata ovunque ci si aspetti un'istanza della superclasse, ad una variabile di tipo *Veicolo* può essere assegnato un oggetto istanza della classe *Camion*.

Ogni variabile ha un tipo statico che è il tipo di cui è dichiarata, e un tipo dinamico, classe più specifica dell'oggetto cui la variabile è istanziata.

Tramite l'overriding è possibile definire metodi che possono essere applicati ad oggetti di tipo differente. Ad esempio: si vuole visualizzare un bitmap, un window, un record impiegato. Nel sistema convenzionale è necessario scrivere tre procedure distinte una per ogni tipo diverso. Nella gestione ad oggetti si definisce una classe generale (astratta) Screen Object con tre sottoclasse: bitmap, window, impiegato. Si definisce il metodo display e in ogni sottoclasse si implementa opportunamente tale metodo.

Overloading e overriding permettono di gestire operazioni con lo stesso nome ma implementazioni differenti. L'overloading può essere applicato anche in assenza di ereditarietà. L'overriding implica l'utilizzo del late binding, in quanto il metodo con cui rispondere al messaggio non può essere deciso a compile time, ma deve essere scelto solo run-time.

Il metodo lookup, dispatching, è l'operazione effettuata dal sistema per determinare il metodo da eseguire per rispondere ad un messaggio: determina la classe più specifica cui l'oggetto ricevente appartiene, considera il suo tipo dinamico, dopodiché determina la superclasse più specifica di tale classe che fornisca una implementazione per il metodo invocato, risalendo alla gerarchia di ereditarietà.

Esempio:

Classe	Persona
Metodo	Aggiorna_Stip()
Sottoclasse	Manager
Metodo (overriding)	Aggiorna_Stip()

Nel codice è possibile definire l'istanza P di persona

A run time, P può essere una istanza della sottoclasse manager

- il tipo dinamico di P è manager
- si sceglie l'implementazione di aggiorna_stip, viene eseguita quella implementata nella sottoclasse manager.

E' possibile costruire applicazioni che possono essere estese senza modificarne il codice: le modifiche allo schema (aggiunta o cancellazione di una classe) non impattano il codice applicativo. Riprendendo l'esempio della visualizzazione, se è necessario visualizzare informazioni relative al tipo *studenti*, è sufficienta definire una nuova classe *studente* come sottoclasse di *screen_object*. Non è necessario modificare o ricompilare il codice che effettua la visualizzazione degli oggetti.

L'ereditarietà multipla permette ad una classe di avere più superclassi; la risoluzione dei conflitti di alternative può essere implicita, basata su un implicito ordinamento delle classi per cui le caratteristiche in conflitto vengono ereditate dalla prima superclasse; esplicita se l'utente specifica da quale superclasse la caratteristica dubbia debba essere ereditata. Un'altra possibilità è quella di impedire conflitti di nome, definire sottoclassi che non hanno caratteristiche con nomi comuni.

L'ereditarietà può essere implementata per i seguenti aspetti:

- gerarchia di specializzazione (subtype hierarchy), consistenza fra le specifiche dei tipo, sostituibilità; comportamento degli oggetti come visto dall'esterno.
- Gerarchia di implementazione, condivisione del codice tra le classi.
- Gerarchia di classificazione, relazioni di inclusione tra collezioni di oggetti.

2.7 Accesso agli oggetti

E' possibile distinguere tre tipologie di accesso agli oggetti:

- accesso **navigazionale**, dato l'OID il sistema accede direttamente all'oggetto riferito, è un sistema molto efficiente. Vi è la possibilità di accedere agli oggetti navigando da uno all'altro.
- Accesso **associativo**, attraverso un linguaggio di interrogazione, select….
- Accesso per **nome**, tramite nomi esterni specificati dall'utente MioDoc.titolo.

L'accesso navigazionale è molto importante nelle applicazioni, sfrutta la gerarchia di aggregazione tra gli oggetti e la presenza di riferimenti espliciti (direzionali). Nei sistemi relazionali è estremamente inefficiente perché richiede molte operazioni di join (una per ogni '.').

L'accesso associativo tramite il linguaggio di interrogazione è utile per lavorare su grosse quantità di dati. Avere a disposizione un linguaggio ad alto livello riduce i tempi di sviluppo delle applicazioni. I linguaggi di interrogazione dichiarativi sono alla

base del successo dei sistemi relazionali.

Nell'accesso per nome, i nomi esterni forniscono agli utenti riferimenti semanticamente significativi agli oggetti. Negli oggetti per cui è possibile un accesso diretto, permettono di definire un punto di entrata nella base dati.

Le varie modalità di accesso ai dati non sono esclusive, ma complementari. Si seleziona un insieme di oggetti da una classe (collezione) con una interrogazione dichiarativa, si naviga da un oggetto per visualizzare le sue componenti.

Una delle caratteristiche che distinguono un OODBMS da un Persistent Object System è proprio la presenza di un linguaggio di interrogazione dichiarativo.

2.8 Persistenza

La persistenza è un concetto importante all'interno degli OODBMS. Si parla di persistenza degli oggetti intendendo come gli oggetti vengono inseriti e rimossi dalla base di dati. Si ha la **persistenza automatica** quando un oggetto creato diventa automaticamente persistente oppure quando non è necessario un comando esplicito per l'inserimento dell'oggetto. Si parla di radici di persistenza se gli oggetti creati sono transienti, non persistenti o che esistono solo nella sessione di lavoro, e che per renderli persisteni è necessario assegnare loro un nome o associarli come componente ad un oggetto persistente.

Anche la rimozione degli oggetti dalla base dati può avvenire tramite un comando di cancellazione esplicito, oppure direttamente dal sistema nel momento in cui non è più riferito da altri oggetti. Questo secondo metodo assicura l'integrità referenziale, ma necessita del meccanismo di garbage collector.

Molti sistemi permetto di avere istanze persistenti e transienti di una stessa classe e l'accesso avviene in modo uniforme indipendentemente dalla situazione di persistenza in cui si trovano.

2.9 Linguaggio di definizione e di interrogazione

Il linguaggio ODL è il linguaggio Object Definition Language per la definizione degli schemi a oggetti. ODl descrive i tipi e non le classi, ed è indipendente dal

linguaggio di programmazione prescelto per l'implementazione delle classi. Nell'Object Definition Language è possibile definire solo l'interfaccia di un metodo nell'ambito della definizione di tipo; l'implementazione viene realizzata tramite un linguaggio di programmazione.

Il linguaggio OQL è il linguaggio Object Query Lanaguage per eseguire le interrogazioni agli oggetti. La maggior parte di questi linguaggi sono estensioni dei linguaggi relazionali. La maggiore ricchezza del modello dei dati introduce nuove problematiche, quali la chiusura del linguaggio di interrogazione, la mancanza di base formale (algebra/calcolo ad oggetti), ricerca di nuove tecniche per l'ottimizzazione (metodi e sistemi di indicizzazione specializzate).

2.10 Limiti del modello ad oggetti

Il limite principale degli Object Oriented DBMS è costituito dall'assenza di uno standard definito, presente invece nei relazionali.

I modelli relazionali, grazie a SQL, sono praticamente compatibili tra loro e un'applicazione costruita per Oracle è facilmente portabile in Informix o Sql Server. Una applicazione scritta, invece, per Oject Store difficilmente potrà essere convertita in modo da gestire dati residenti su un'altra base dati ad oggetti, quale GemStone.

Il progetto della base dati è strettamente legato al progetto delle applicazioni, e vi è la mancanza totale di un modello dei dati e di un linguaggio di query standard pienamente accettati.

2.11 Object Management Group

Per definire degli standard nell'area orientata agli oggetti, è nato nel 1989 il gruppo di lavoro Object Management Group. L'Object Database Management Group è uno dei gruppi di lavoro dell'OMG che ha lo scopo di sviluppare una serie di standard per favorire portabilità e interoperabilità degli OODBMS commerciali. Attualmente il gruppo è sciolto in quanto ha completato l'attività assegnatagli e i risultati ottenuti sono riportati nei seguenti documenti:

- ODMG 93 standard, edito 1993;
- ODMG 2.0 standard, nel 1997;
- ODMG 3.0 standard, del 1999.

I documenti pubblicati riassumono il modello dei dati ad oggetti; tracciano le linee guida per un linguaggio di definizione Object Definition Language, dall'*Interface Definition Language* di Corba; un linguaggio di interrogazione Object Query Language, la cui base è SQL, e meccanismi per la costruzione di metodi nei linguaggi C++ e Smalltalk. Questi standard si pongono come obiettivo l'interoperabilità tra i molteplici sistemi delle diverse case produttrici.

2.12 The Object Oriented DB Manifesto

Le caratteristiche principali sugli OODBMS sono definite nel "Manifesto delle basi di dati a oggetti", l'articolo pubblicato nel 1989. In esso vengono elencate una lista di funzionalità per definire e valutare gli OODBMS. Queste funzionalità sono suddivise in obbligatorie (the golden rules), opzionali e scelte aperte (no consensus).

Le golden rules si possono riassumere in:

- *Thou shalt support complex object*. Complessità strutturale, la capacità del sistema di costruire tipo complessi.
- *Thou shalt object identità*. Identità di oggetto, l'identificazione in modo univoco dell'oggetto in base al suo OID;
- *Thou shalt incapsulate thine objects.* Incapsulamento dell'oggetto, definire quindi i metodi, l'interfaccia e attraverso di essi modificare lo stato dell'oggetto.
- *Thou shalt support type or classes*. Tipi e/o classi, la presenza del concetto di tipo per controllare la correttezza dei programmi in compilazione e del concetto di classe come sistema per raccogliere gli oggetti e definirne le implementazioni in esecuzione.
- *Thine classes or types shalt inherit from their ancestors.* Gerarchie di classe o di tipi, capacità di dare complessità semantica all'OODB organizzando le classi tramite gerarchie di generalizzazione e dando loro un tipo più specifico tramite le gerarchie di tipo.
- *Thou shalt not bynd prematurely*. Overrriding, overloading e late binding, per utilizzare implementazioni più specifiche in ciascun oggetto.

- *Thou shalt be computationally complete*. Completezza computazionale del linguaggio in cui si esprimono i metodi.
- *Thou shalt be extensible*. Estensibilità, la capacità di definire nuovi tipi in base alle necessità dell'utente.
- *Thou shalt remember thy data*. Persistenza, capacità di rendere persistenti i propri dati.
- *Thou shalt manage very large databases*. Gestione della memoria secondaria, efficienza nella gestione degli accessi.
- *Thou shalt accept concurrent users*. Concorrenza, gestione degli accessi concorrenti.
- *Thou shalt recover from hardware and software failures*. Recovery, robustezza e gestione degli errori.
- *Thou shalt have a simple way of querying data*. Presenza di un linguaggio di interrogazione.

Le funzionalità quasi obbligatorie si possono riassumere:

- Dati derivati e definizione di viste;
- Funzionalità per database Administrator;
- Vincoli di integrità;
- Funzionalità per la modifica di schemi.

Le proprietà opzionali invece riguardano:

- ereditarietà multipla;
- verifica dei tipi ed inferenza durante la compilazione;
- distribuzione dei dati;
- "Design transactions", gestione di transazioni lunghe e nidificate;
- Gestione delle versioni, presenza di meccanismi espliciti per la gestione delle versioni.

Capitolo 3

Basi di Dati Relazionali a Oggetti

3.1 Modello dati

Gli OODBMS forniscono un supporto efficiente ed efficace per alcune classi di applicazioni su dati complessi, ma senza molti degli positivi dei relazionali.

Le basi di dati relazionali a oggetti nascono dall'esigenza di assicurare le funzionalità dei relazionali rispetto alla gestione di dati tradizionali, estendendo il modello verso la gestione di dati complessi tipica della struttura ad oggetti.

Le innovazioni introdotte nel modello relazionale a oggetti riguardano la gestione di nuove tipologie di dato, i tipi testi, le immagini, gli audio/video, i dati geografici, i tipi di dato definiti dall'utente e i tipi collezione, i metodi per modellare le operazioni sui tipi definiti dall'utente e i nuovi modi per gestire le associazioni.

La filosofia del ORDBMS per la gestione dei dati è ancora quella relazionale:

- tutti gli accessi ai dati avvengono tramite SQL;
- tutte le entità di interesse sono modellate tramite tabelle.

Attualmente, i maggiori produttori di basi dati relazionali hanno esteso i loro DBMS con caratteristiche object-relational. Tali estensioni presuppongono anche una implementazione ulteriore del linguaggio SQL, e ogni sistema relazionale ha una sua implementazione proprietaria object-relational. Le estensioni differiscono per le funzionalità che supportano, per il modo di realizzare le caratteristiche ad oggetti e, soprattutto, per le modalità di implementazione delle funzionalità aggiuntive in SQL.

La definizione dello standard SQL:1999 è un tentativo di unificazione dell'estensione object-relational del modello relazionale. Al momento della validazione di tale standard, i maggiori produttori di sistemi relazionali avevano già introdotto nel loro prodotto la loro visione di object-relational e, in ogni caso, SQL:1999 non standardizza tutte le funzionalità a oggetti presenti nei RDBMS commerciali. E' presto per stabilire in che modo lo standard impatterà nei sistemi presenti, ma, data l'evoluzione, è probabile che sarà necessario definire un ulteriore standard che medi tra tutte le estensioni proprietarie.

Il modello SQL:1999 comprende due componenti fondamentali: **structured**

user-defined types (UDT) e le **typed tables**.

Lo user-defined-types UDT permette all'utente di definire tipologie di dato più complesse rispetto alle tipologie semplici. Con la definizione dell'UDT viene anche definita la semantica, il comportamento che il tipo deve mantenere se invocato.

Le typed tables sono tabelle le cui righe sono istanze di un particolare UDT al quale la tabella è associata.

La combinazione di questi due nuovi tipi, gli UDT e le typed tables, portano a definire un vero modello ad oggetti tramite SQL.

3.2 Linguaggio di interrogazione

SLQ:1999 è compatibile con SQL 92, la sintassi utilizzata per SQL 92 resta valida anche per SQL:1999. La navigazione tra i riferimenti in SQL:1999 richiede l'utilizzo dell'operatore di dereferenziamento, *deferencing*. Tramite questo operatore è possibile accedere da un soggetto sorgente x ad un attributo A di un oggetto y referenziato in x. Gli attributi OID possono essere esplicitamente utilizzati nelle interrogazioni e in particolare possono essere confrontati tramite l'operatore di uguaglianza con i riferimenti a tuple dello stesso tipo.

3.3 Progettazione di ORDBMS

Per la progettazione dei modelli object-relational non esiste ancora una metodologia consolidata come per i relazionali, e dei tools a supporto dell'attività di progettazione. E' possibile affrontare la progettazione partendo da schemi ER.

Nel caso a oggetti la ristrutturazione dal modello concettuale a quello logico è inesistente: non si eliminano attributi multivalore/compositi e gerarchie di generalizzazione, in quanto vengono trasferiti in gerarchie di tipi. Dallo schema ER, gli attributi compositi diventano tupla oppure ADT. Ogni attributo multivalore viene tradotto in un tipo collezione, array. Ogni entità, se non ha metodi, può essere tradotta in una tabella; invece se ha metodi viene convertita in tipo su cui costruire una tabella.

Le gerarchie di generalizzazione vengono tradotte mediante relazioni di

sottotipo.

Se si progetta il DB partendo da uno schema OO, ogni tipo composito (struct) diventa tipo tupla o tipo complesso.

Ogni tipo multivalore (set, bag, list) diventa tipo collezione.

Dalle classi che non hanno metodi, si creano direttamente le tabelle; i tipi degli attributi vengono trattati nello stesso modo di come è indicato nella conversione dal modello ER. Per le classi che possiedono metodi si crea un opportuno ADT e la tabella associata. Per ogni attributo aggregato è necessario distinguere l'aggregazione semplice da quella complessa. Per la prima si specifica un tipo riferimento con scope uguale alla tabella corrispondente alla classe riferita; per l'aggregazione complessa si specifica un tipo array definito su un tipo riferimento.

3.4 The third generation Database System Manifesto

"The third generation Database System Manifesto" rappresenta la risposta dei sistemi relazionali a oggetti al manifesto degli OODBMS e stabilisce che i DBMS di terza generazione devono essere ottenuti come naturale evoluzione delle basi dati relazionali. I principi base su cui si fonda il contromanifesto possono essere riassunti in tre aspetti fondamentali:

- i sistemi di terza generazione dovranno essere una generalizzazione compatibile con i sistemi di seconda generazione;
- fornire servizi tradizionali di gestione dei dati;
- dovranno permettere la definizione di oggetti complessi e regole, e dovranno essere aperti all'interazione con altri sistemi.

I punti più significativi presenti nel manifesto sono i seguenti:

- *Rich system type*. Deve essere un sistema di tipi ricco, che comprenda costruttori per array, record e insiemi.
- *Inheritance*. Gerarchie di generalizzazione fra tipi con ereditarietà multipla.
- *Function and encapsulation*. Deve includere funzioni, procedure e metodi accompagnate dall'incapsulamento.
- *OID's only if there are no keys*. Assegnamento dell'OID solo nel caso in cui non è disponibile una chiave primaria tra gli attributi definiti dall'utente.
- *Rules and triggers*. Possibilità di inserire regole attive (triggers) e passive

(vincoli di integrità).

- *Non procedural, high level access languages.*
- *Specification techniques for collections.*
- *Updatable views.*
- *Transparency of physical parameters.*
- *Multiple high level languages.*
- *Persistent x for many x's.*
- *Sql is a standard.* Viene posto SQL come il linguaggio di riferimento nei DBMS.
- *Queries and their results are the lo west levest of communication*.

3.5 SQL:1999

3.5.1 Tipi e funzioni di casting

In SQL-92 i tipi di un attributo in una relazione possono essere:

- numerici (interi, reali, ecc)
- carattere (stringhe di lunghezza fissa o variabile, caratteri)
- temporali (date, time, ecc.)
- booleani
- non strutturati (byte, text, blob, cblob)

Per ogni tipo built-in esiste un insieme fisso e predefinito di operazioni che su di esso possono essere eseguite. Queste limitazioni rendono spesso difficile la rappresentazione di dati reali.

In SQL:1999, l'estensione dei tipi ha portato all'introduzione degli *user defined types*.

Uno **user-defined types** è un tipo astratto (ADT), non è presente all'interno del sistema di base dati e non è neppure costruito attraverso un linguaggio di programmazione, ma è definito come nello sviluppo di una applicazione e con esso vengono definiti anche i comportamenti che può assumere.

Si distinguono due tipologie di tipi UDT:

- i tipi **semplici** (o *distinct type*), sono la forma più semplice di estensione del sistema dei tipi fornita da un ORDBMS. E' un tipo di dato che è basato su un

singolo built-in type, ad esempio intero, ma il cui valore non può essere soggetto alle stesse regole del tipo built-in. E' importante quindi definire il comportamento o il dominio di appartenenza di tale valore. Il confronto tra questo tipo e il rispettivo built-in su cui è costruito ovviamente non è più ammissibile perchè devono essere specificate le regole di casting. Il sistema fornisce due metodi di cast abbinati al distinct type: una per convertire dal distinct type al built-in type, e l'altra per il viceversa. L'utilizzo di tali tipi e l'implementazione delle rispettive funzioni di casting, permettono la gestione di confronti costruiti nel rispetto delle regole imposte dall'utente. Non sono definiti invece meccanismi di ereditarietà e/o sottotipo per i tipi semplici.

- I tipi **strutturati** (*structured types*) invece, come indica anche il nome, permettono di avere all'interno della loro definizione strutture di tipi. Ad esempio la rappresentazione dell'indirizzo potrebbe generare un tipo indirizzo composto dai tipi *VIA*, *N_CIVICO*, *INTERNO*, *CAP.*, *CITTA*, *PROVINCIA*. Le istanze dei tipi strutturati non sono oggetti, ma valori. In SQL:1999, l'utente può creare un certo numero di tipi strutturati e usarli come tipi per colonne e variabili SQL, ecc. Inoltre questi tipi possono essere associati a funzioni specifiche definite dall'utente che sono usate per tutti i confronti di istanze di tipo, per le conversioni da un tipo ad un altro, per implementare tutti i possibili comportamenti previsti per il tipo.

Sintassi per la creazione di UDT:

CREATE TYPE <name> [**UNDER** <supertype-name>]
AS <built-in type >
[AS representation]
[[**NOT**] **INSTANTIABLE**]
[**NOT**] **FINAL**
[<specifica tipo di riferimento>]
[<opzioni per le funzioni di cast>]
[lista dei metodi]

Una caratteristica dell'orientamento ad oggetti è la capacità di apportare modifiche interne al codice e ai dati senza che venga richiesto l'aggiornamento delle applicazioni che

utilizzano tali informazioni (incapsulazione). Anche in SQL:1999 tutti i tipi strutturati sono incapsulati, nel senso che sono definiti in modo tale che risulta difficile capire le caratteristiche implementate. Tutti gli accessi ad istanze di tipo strutturato passano attraverso le funzioni definite assieme al tipo.

I tipi di dato strutturato è inoltre composto da un insieme di attributi. Ogni attributo ha un singolo tipo di dato atomico o semplice. L'insieme degli attributi di un tipo strutturato è detto "rappresentazione del tipo". Nella sintassi di creazione tipo, la clausola *AS REPRESENTATION* specifica che tutti gli attributi sono di tipo UDT.

La clausola *[{INSTANTIABLE|NOT INSTANTIABLE}]* definisce se il tipo può essere istanziato, se può avere istanze proprie oppure no. Nel caso fosse *NOT INSTANTIABLE* , si ottiene una classe astratta, utilizzabile per il riuso del codice e per l'overriding.

Oltre ai metodi built-in, SQL permette di definire delle funzioni *user-defined cast*: funzioni di conversione definte dall'utente. L'utilizzo di tali funzioni è vario: può essere necessario convertire l'UDT in un tipo che sia riconoscibile dagli applicativi host, oppure per convertire il dato in un qualsiasi tipo accettato dall'ambiente SQL, ecc.

La sintassi della funzione di CASTING:

CREATE CAST (<source type> **AS** <target type>)
WITH <function signature >
[**AS ASSIGNMENT**]

dove <*function signature*> è la specifica routine di conversione.

Dopo aver creato la funzione, per invocarla è sufficiente richiamare la funzione CAST assegnandogli i parametri in modo appropriato; i parametri in ingresso devono essere stati previsti e trattati all'interno della funzione. Almeno un argomento tra il *source type* e il *target type* deve essere il tipo definito dall'utente.

La sintassi di definizine della <function signature> è la seguente:

FUNCTION <nome> (<source type>) **RETURNS**
<target type>
... codice ...

Se la clausola *AS ASSIGNMENT* è specifiata, il casting è invocato implicitamente quando necessario. Per ogni coppia di tipi può esistere una sola funzione di casting definita dall'utente.

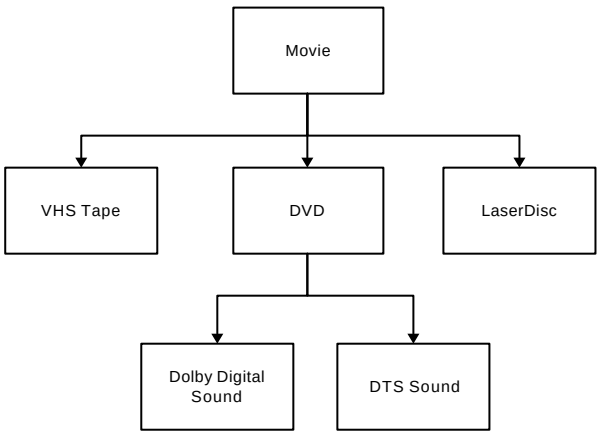
La funzione di casting può essere invocata implicitamente o esplicitamente tramite la sintassi:

CAST (<source type> **AS** <target type>)

La funzione può inoltre essere invocate per il casting tra tipi built-in.

3.5.2 Ereditarietà

La gerarchia in SQL è un insieme di UDT, alcuni dei quali sono specializzazioni di altri tipi.



Nella figura il box “Movie” rappresenta un tipo UDT, mentre i box VHS Tape, DVD e Laser Disc rappresentano UDT con informazioni addizionali rispetto a Movie. Essi vengono definito sottotipi dell’UDT Movie, che a sua volta invece è definito supertipo. L’idea di sottotipo è quella di trattare una struttura che ha una relazione di dipendenza con un’altra, i tipi di dati provengono da una altra struttura. Nella definizione di tipo in SQL, il sottotipo può essere definito utilizzando la clausola UNDER <supertipo>. In questo modo il sottotipo eredita la struttura del supertipo con aggiunta le informazioni specificate nella sua dichiarazione di tipo.

In SQL:1999 non è possibile definire strutture basate direttamente o indirettamente su se stesse, e non è possibile definire strutture di tipo con attributi che non sono definiti o che si devono definire.

Un sottotipo eredita gli attributi, i metodi, ed i vincoli definiti per i suoi supertipi. Il

sottotipo può raffinare il supertipo con nuovi attributi e metodi. Nel sottotipo è anche possibile ridefinire metodi ereditati.

Si distinguono due tipi di ereditarietà, ereditarietà singola ed ereditarietà multipla.

Si parla di **ereditarietà singola** quando un sottotipo eredita attributi e metodi da un supertipo; di **ereditarietà multipla** quando invece un sottotipo può ereditare da più di un supertipo. Questa seconda tipologia di ereditarietà non è supportata per ora in SQL:1999.

Sintassi per la creazione di un sottotipo:

CREATE TYPE <nome tipo>
UNDER <nome superclasse**>**
As <... altri attributi>
[**NOT**] **FINAL**;

Il supertipo deve essere dichiarato con la clausola *NOT FINAL*, se è presente la clausola *FINAL* non possono essere definiti sottotipi.

I metodi sono ereditati dai sottotipi allo stesso modo degli attributi.

E' possibile ridefinire un metodo ereditato, non è possibile ridefinire gli attributi.

Esempio: si consideri il tipo *t_persona* con attributo *ETA* e il metodo *Eta()* che calcola gli anni di età dell'persona, l'oggetto in esame. Si crei il sottotipo *t_insegnante,* il metodo Eta() può essere riscritto per restituire l'anzianità di servizio, overriding.

CREATE TYPE t_insegnante **AS**
UNDER T_PERSONA
    STIPENDIOP DECIMAL (9,2),
    DATA_ASSUNZIONE DATE,
    CORSO T_CORSO,
OVERRIDING METHOD ETA **RETURNS** INTEGER
NOT FINAL;

3.5.3 Cancellazione e modifica di tipi

I tipi possono essere modificati e cancellati. Le sintassi per implementare le due operazioni sono le seguenti:

DROP TYPE <nome tipo> {**CASCADE|RESTRICT**}

esegue la cancellazione del tipo dal base dati;

ALTER TYPE <nome tipo> <operazione_di_modifica>

modifica il tipo esistente.

<operazione_di_modifica> ::= **ADD ATTRIBUTE** <defi.attr.> |
DROP ATTRIBUTE <nome attr.>

3.5.4 Typed tables

Le istanze degli UDT vengono memorizzate in strutture chiamate *Typed Tables*.

Le **Typed Tables** sono tabelle le cui righe memorizzate rappresentano l'istanza del tipo strutturato a cui la tabella è associata. Il nome e il tipo di dato di ciascuna colonna rispecchia la struttura del tipo definito dall'utente.

Le *typed tables* a tutti gli effetti sono tabelle SQL, pertanto l'accesso a tali tabelle avviene con il normale linguaggio SQL (*INSERT, UPDATE, SELECT, DELETE*).

Una caratteristica importante delle typed tables riguarda la colonna di riferimento (*self-referencing column*). Ogni riga della tabella contiene un campo in più che identifica in modo univoco rispetto a tutte le typed table nella base dati le istanze create: ogni istanza è globalmente unica.

Il self-referencing value può essere utilizzato come una sorta di puntatore o di chiave per gestire l'istanza riferita.

Anche le type table partecipano alle gerarchie in modo analogo come gli UDT. Una tabella può essere definita come sottotabella di un'altra, detta invece supertabella. In questo modo viene dichiarata una gerarchia di tabelle e tutte hanno il corrispondente tipo nella gerarchia dei tipi. La tabella nella gerarchia tra tabelle deve trovare lo stesso posto che occupa il corrispondente tipo nella gerarchia dei tipi.

La sintassi per definire una typed tables è la seguente:

CREATE TABLE <nome tabella>

OF <nome UDT>
[**UNDER** <supertable>]
[**REF IS** <nome colonna self-referencing> <**SYSTEM GENERATED | USER GENERATED** | **DERIVED**>]
[<nome colonne> <tipo | **WITH OPTIONS** <lista option>]

Dalla sintassi di creazione tabella, si evince chiaramente che il riferimento al tipo a cui è associata deve essere esplicito.

La clausola [*UNDER* <*supertable*>] serve per specificare la supertabella di riferimento nel caso di gerarchia. Come per i tipi, anche nella gerarchia di tabelle, una sottotabella eredita le colonne della supertabella.

Per quanto riguarda la definizione delle colonne, per una sottotabella non è possibile definire vincoli di chiavi primarie, ma solo vincoli di colonne univoche.

E' possibile inoltre definire vincoli di integrità referenziale come parte della definizione di una typed tables. La tabella a cui l'attributo riferisce può essere un'altra typed tables o anche una ordinaria tabella SQL, ma non una sua sottotabella.

La colonna *self-referencing* non può essere specificata quando viene definita una sottotabella, ma deve essere indicata nella definizione della supertabella. Tale colonna viene poi ereditata da tutte le sottotabelle.

La sintassi per specificare una colonna *self-referencing* e quindi impostare una relazione di integrità referenziale, è la seguente:

REF IS <nome della colonna *self-referencing* > <tipo di referenza>

3.5.5 Operazioni sulle Typed Tables

Le operazioni di manipolazioni dati sulle *Typed Tables* possono essere eseguite con le normali istruzioni SQL: select, insert, dolete, e update, in quanto tabelle a tutti gli effetti.

Nel caso in cui si debbano eseguire interrogazioni o altre funzionalità su gerarchie di tabelle, l'argomento presenta dei risvolti interessanti.

La SELECT eseguita su una supertabella ritorna tutte le istanze che soddisfano le condizioni della clausola WHERE, eseguendo una UNION con tutte le sottotabelle della

gerarchia. Nel caso si volessero solo i dati della supertabella, è necessario specificare l'opzione ONLY nella clausola FROM della SELECT.

FROM <nome tabella>	->	Ricerca nella tabella e nelle sue sottotabelle se presenti
FROM ONLY (<nome tabella>)	->	Ricerca solo nella tabella specificata

Per referenziare gli attributi di tipi strutturati è necessario adottare la notazione **dot**:

<nome campo>**.**<nome tipo attributo>

Per referenziare i metodi dei tipi strutturati la notazione da adottare è la seguente:

<nome campo> **->** <nome metodo()>

L'inserimento nelle gerarchie di tabelle avviene sempre nella tabella specificata dalla operazione di INSERT, mentre la funzionalità di DELETE opera sulla tabella e su tutte le sottotabelle della gerarchia in esame, salvo l'utilizzo della clausola ONLY come per la SELECT.

Per quanto riguarda l'UPDATE, invece, l'aggiornamento si ha su tutte le tabelle sottotabella e supertabelle della tabella indicata senza la clausola ONLY, e sulla tabella specificata e le sue supertabelle con la clausola ONLY.

3.5.6 Incapsulazione e metodi

SQL:1999 incapsula ogni attributo di un tipo strutturato fornendo una coppia di routine interne che vengono invocate quando una applicazione riferisce al tipo. Una routine è definita **observer** o **accessor** ed è usato per recuperare il valore di un attributo; l'altra è detta **mutator** ed è invocata per modificare il valore dell'attributo.

Le funzioni vengono invocate automaticamente alla richiesta dell'attributo o alla modifica del suo valore, questo perché sono referenziate attraverso il nome stesso dell'attributo, che risulta così incapsulato da queste due funzioni.

Un altro metodo predefinito è il **constructor**, termine usato per descrivere il processo di creazione di una istanza. Poiché i tipi UDT sono valori e non oggetti, il processo di "costruzione" di una istanza del tipo UDT è decisamente diverso rispetto al processo di

costruzione di una istanza di un oggetto. Si è visto che la creazione di una istanza di un oggetto necessita l'allocazione delle risorse necessarie per l'oggetto, dopodiché associa agli attributi il valore desiderato. Per quanto riguarda l'UDT, lo scopo del *constructor* è di assegnare il valore specifico per quel tipo, ha un compito di inizializzatore.

SQL:1999 definisce due tipi di metodi: i metodi statici e i metodi di istanze. Un **metodo statico** opera solo sul tipo UDT, mentre un **metodo di istanza** opera sull'istanza del tipo. Questa distinzione si ripercuote sui parametri della definizione e della chiamata al metodo.

I metodi sono funzioni definite dall'utente associate ai tipi e possono essere scritti in

linguaggi proprietari del DBMS o in linguaggi di programmazione standard. La sintassi dipende dal DBMS utilizzato ma è simile a quella delle funzioni. La differenza sostanziale tra metodi e funzioni consiste nel fatto che i metodi hanno un parametro implicito che rappresenta l'oggetto su cui viene invocato e sono specifici del tipo UDT.

La definizione dei metodi in SQL è prevista in due momenti ed entrambi sono richiesti. La prima dichiarazione attesa è nella definizione di tipo strutturato; la seconda dichiarazione è indipendente dalla definizione del UDT, ma risiede nello stesso schema a cui è associato il tipo. Questo secondo modo è la classica dichiarazione di metodo che comprende anche l'implementazione.

Ogni metodo ha sempre almeno un parametro implicito, che è il tipo di dato a cui è associato il metodo, identificato con SELF. Nella chiamata al metodo, non è necessario specificare il tipo di dato in oggetto, perché il metodo per costruzione è associato al tipo dichiarato.

Il metodo potrebbe richiedere ulteriori parametri oltre a quello implicito, in tal caso si applicano le classiche regole del passaggio dei parametri: il tipo di ogni parametro dichiarato nella signature del metodo deve essere compatibile con il tipo di dato corrispondente dichiarato nella definizione del metodo; il parametro di ritorno deve coincidere con il tipo dichiarato; il nome del parametro specificato nella definizione deve coincidere con il nome specificato nella dichiarazione.

Nella signature dei metodi, e non nella definizione, possono essere aggiunte delle clausole opzionali che specificano informazioni aggiuntive sul metodo; queste opzionali caratteristiche sono:

- il nome del linguaggio di programmazione con cui è scritto il metodo, per default LANGUAGE SQL.
- Quando il linguaggio non è SQL, lo stile dei parametri usati per il metodo,

default PARAMETER STYLE SQL.

- Se il metodo non contiene SQL specificare lo statements NO SQL, CONTAINS SQL se contiene SQL ma non accede al database, se legge solo i dati ma non fa aggiornamenti READS SQL DATA, o se esegue aggiornamenti MODIFIES SQL. Il default è CONTAINS SQL.
- Se un metodo è deterministico, cioè se ritorna sempre la stessa risposta in corrispondenza di un set di argomenti (DETERMINISTIC/NOT DETERMINISTIC).
- Se il metodo può ritornare valori nulli CALLED ON NULL INPUT.

Il numero dei parametri nella chiamata ad un metodo deve coincidere con il numero degli argomenti nella definizione del metodo; il tipo del parametro di risposta deve corrispondere al tipo definito.

In SLQ:1999, la sintassi per la creazione dei metodi è la seguente:

CREATE [**INSTANCE** | **STATIC** | **CONSTRUCTOR**] **METHOD** <nome metodo>
(lista parametri)
RETURNS <output data type>
FOR <nome UDT>
<corpo metodo>

I metodi nelle query vengono invocati con la notazione dot.

Nella tabella seguente sono riportate le principali differenze in SQL tra procedure, funzioni e metodi.

Caratteristiche	Procedure	Funzioni	Metodi
Modalità di chiamata	CALL <nome> (<arg>)	<return>=<nome> (<argomenti>)	Dot notation Nome.nomemetodo(<arg>)
Associazione ad un tipo	No	No	Si
Schema di appartenenza	Qualsiasi	Qualsiasi	Schema associato al tipo
Risultato routine	Risolta in compilazione	In compilazione	In compilazione risolti i metodo; risoluzione finale a runtime
Argomenti in Input	Sì	Sì	Sì; oltre all'argomento

			implicito del istanza del tipo che ha invocato il metodo
Parametro in uscita	Sì, può ritornare un insieme di risultati	Sì, solo un valore	Sì, solo un valore

3.5.7 Tipi Array, Row e Tupla

I tipi collezione definiscono dei contenitori per oggetti con struttura simile. Non esiste ancora una standardizzazione sull'insieme di tipi collezione supportati dai vari ORDMS: set, bag, liste, array. L'unico tipo incluso in SQL:1999 è il tipo **array**:

<nome_campo> <tipo> **ARRAY** [<dimensione>]

La dimensione è un valore intero.

Il costruttore:

ARRAY [<valore1>,<valore2>,...]

L'accesso:

<nome_campo>[i] dove i è compreso tra 1 e n.

I metodi definiti negli array sono il casting sul tipo degli elementi e la riduzione della dimensione. Il troncamento del dato genera un'eccezione. Le operazioni di confronto possibili sono uguale e diverso.

Le funzioni definite sono il cancatenamento tra due array e la verifica della cardinalità.

Per i campi di tipo array non possono essere definiti vincoli unique, primary key, foreign key.

I **row type** sono disponibili per rappresentare dei tipi di record senza ricorrere ad un UDT, questi possono essere associati direttamente al tipo.

row(<def_campo1>, ..., <def_campon>)

Esempio:

```
ROW (numero_civico integerr,
        via     varchar(50),
        città   char(20),
        stato   char(2),
        cap     integer)

CREATE TABLE Impiegati (
        Imp# id_impiegato,
        nome char(20),
        curriculum      text,
                indirizzo ROW (numero_civico integer,
                                via     varchar(50),
                                città   char(20),
                                stato   char(2),
                                cap     integer)
        );
```

Anche le tuple restituite da una query sono viste come valori del tipo tupla, le componenti di una tupla possono essere accedute utilizzando la dot notation.

Nell'assegnamento devono essere rispettati lo stesso numero di campi e i tipi devono essere compatibili.

Il confronto può essere eseguito su uguale, diverso, maggiore e minore, >=, <=. L'ordinamento applicato è quello lessicografico, basato sui tipi dei componenti. I valori devono avere lo stesso numero di elementi e la presenza di NULL può generare un UNKNOWN.

Anche i tipi tupla non possono essere associati a vincoli di primary key, inique, foreign key.

Capitolo 4

Un Caso di Studio

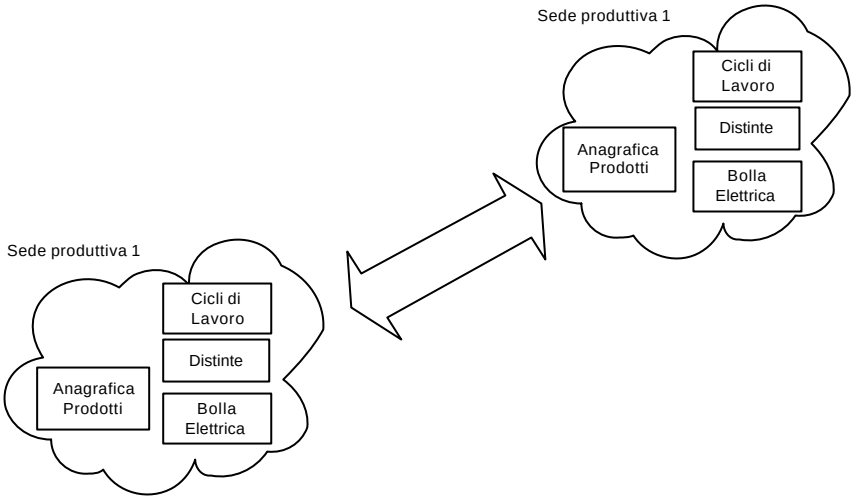
4.1 Premessa

Nella realizzazione di una base dati per la progettazione e la gestione di resistenze e materiali scaldanti per una azienda del nord-est, sono state utilizzate le potenzialità del modello relazionale ad oggetti. Tale modello ha permesso di risolvere alcuni problemi legati alla dinamicità del prodotto, alla richiesta di gestione da siti diversi, allo sviluppo di ulteriori funzionalità che diversamente avrebbero creato difficoltà di manutenzione, aggiornamento e implementazioni future. La gestione del prodotto e la sua configurazione all'interno della base dati, hanno permesso di rendere il sistema aperto ad accogliere sviluppi con tecnologie rivolte a internet, funzionalità che potranno sicuramente essere utili nelle attività svolte fuori sede.

4.2 Analisi Stato dell'arte (As-Is)

L'azienda è composta da diversi siti produttivi presenti in varie parti del mondo. Ciascuno di essi gestisce il prodotto con sistemi locali. Tali sistemi si differenziano sia per la tecnologia utilizzata che per le modalità di gestione.

Gli articoli di produzione possono essere gestiti e progettati in un sito e prodotti in un altro. I dati risultano presenti in tutti gli ambienti.



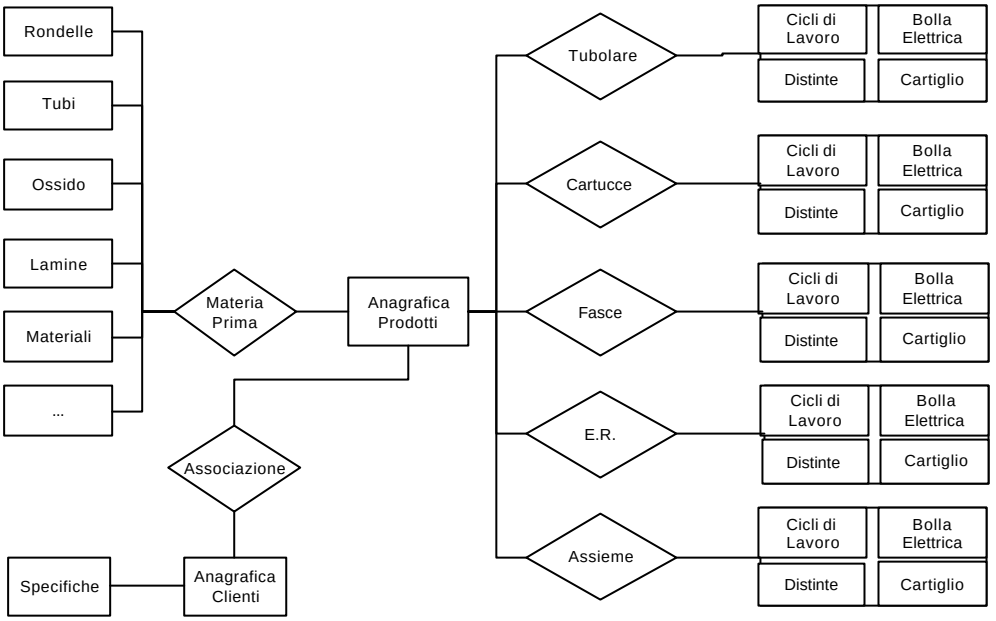
La struttura dei sistemi si basa su tecnologie obsolete, la base dati è costituita da un insieme di tabelle indipendenti ad accesso casuale, dove consistenza e integrità sono garantite da applicativi ad hoc e da processi di backup esterni al sistema.

Attualmente i "programmi tecnici", insieme di applicativi che gestiscono il prodotto dal punto di vista tecnico nella gestione e progettazione dei vari elementi, implementano le seguenti operazioni:

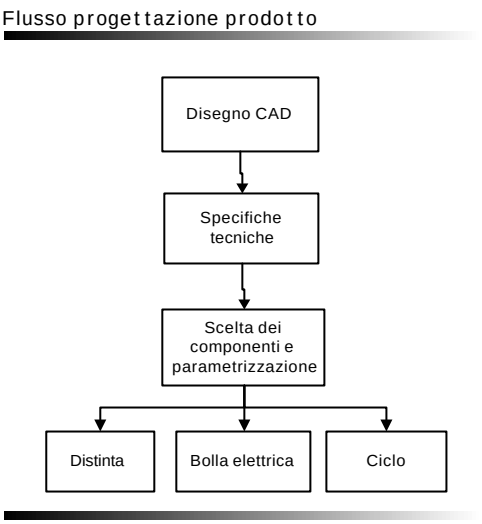
1. Codifica del prodotto, distinta per tipologia di prodotto.
2. Definizione del prodotto tramite specifiche fornite dal tecnico, un progetto eseguito su CAD (disegno) nel quale si identificano le caratteristiche strutturali, la distinta come insieme degli elementi utilizzati per formare il prodotto, il ciclo che descrive i dettagli delle fasi di lavorazione e assemblaggio dei componenti per ottenere il prodotto finito, e infine la gestione della bolla elettrica, documento che riassume i dettagli tecnici del prodotto.
3. Gestione della storia delle modifiche e del tecnico che le esegue, modifica tecnica.
4. Controllo della produzione tramite lo stato di bolla elettrica, blocco della produzione, produzione in modalità preserie, produzione a regime.
5. Implosione: trovare tutti i codici che contengono un determinato componente.
6. Possibilità di sostituire a tutti i codici che contengono un determinato elemento un altro equivalente (multisostituzione).
7. Calcolo del tempo di lavorazione e dei costo del componente.

8. Ricerca dei dati per evitare la duplicazione dei codici.
9. Ricerca per cliente per sapere quali codici utilizza.

Schema As-Is
Gestione del prodotto



L'applicativo è utilizzato oltre che dall'Ufficio Tecnico che ne gestisce i codici contenuti, anche dall'Ufficio Vendite con funzionalità di ricerca e verifica dei prodotti esistenti in modo da poterli proporre ai clienti. Un accesso è dato anche alla Qualità affinché possa approvare le modifiche sia nelle situazioni normali, sia nei casi in cui esiste una normativa cogente. Alla Produzione è consentito verificare i codici in lavorazione tramite la stampa della bolla elettrica e del disegno.



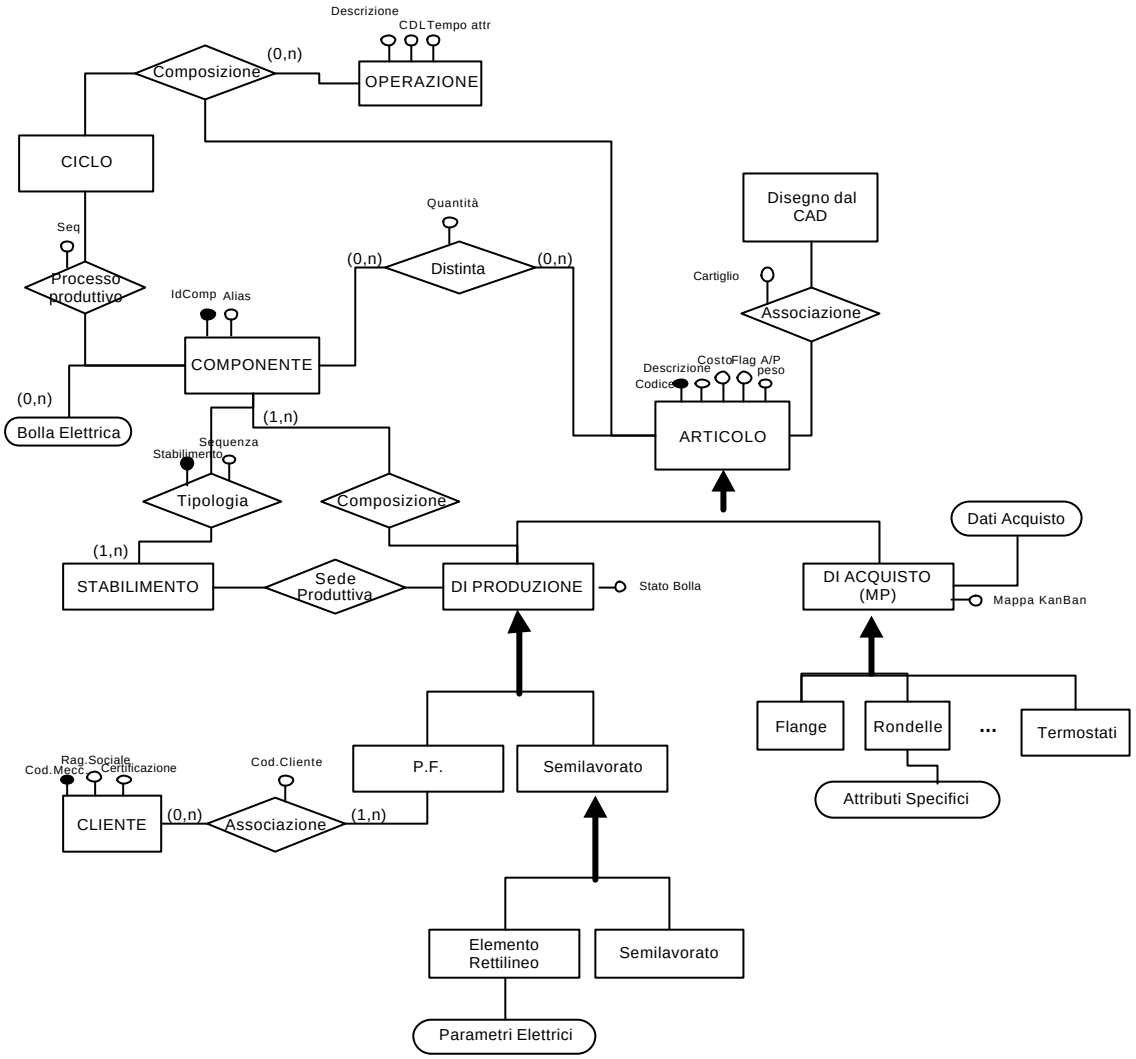
4.3 Modello Dati (To-Be)

Il nuovo progetto avviato persegue principalmente i seguenti obiettivi strategici:

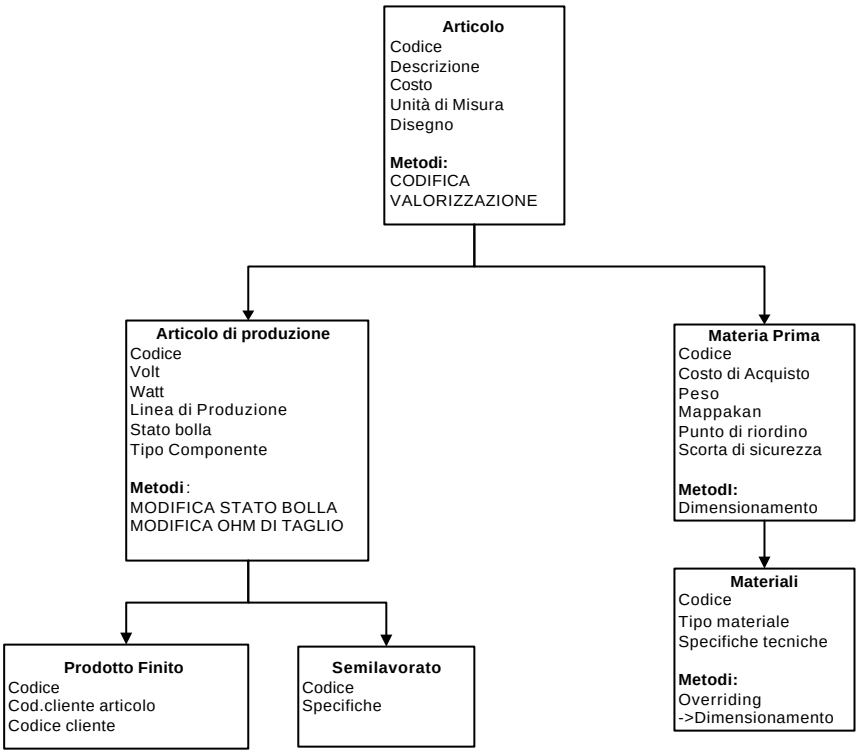
- centralizzazione dell'informazione, per condividere in modo completo le attività svolte in ogni sede e per evitare lo sviluppo di codici diversi ma equivalenti.
- Ridurre i costi di materia prima recuperando materiali già presenti nei magazzini.
- Possibilità di consultazione dei dati da qualsiasi sito preposto.
- Simulazione di nuovi codici per rispondere alla esigenza del cliente di ottenere una offerta nel tempo più breve possibile.
- Velocità di progettazione, per far fronte in modo rapido alle richieste del mercato, considerando che l'impresa si rivolge sia ad un mercato customizzato (un'azienda produce ciò che un cliente richiede), sia ad un mercato standard dove la produzione è di serie e per grossi lotti.

Lo schema E-R proposto è rappresentato in seguito: l'articolo di produzione, prodotto finito o semilavorato, è caratterizzato da bolla elettrica, distinta e ciclo. L'impiego di un semilavorato in un prodotto finito è stabilito dalla soddisfazione dei parametri elettrici.

Ad ogni articolo può essere associato anche un disegno proveniente dal CAD, al quale va abbinato il cartiglio, insieme di informazioni riportanti le specifiche tecniche ed elettriche del disegno, eventuali modifiche, e altro. Tale cartiglio è costruito in funzione della tipologia del prodotto.

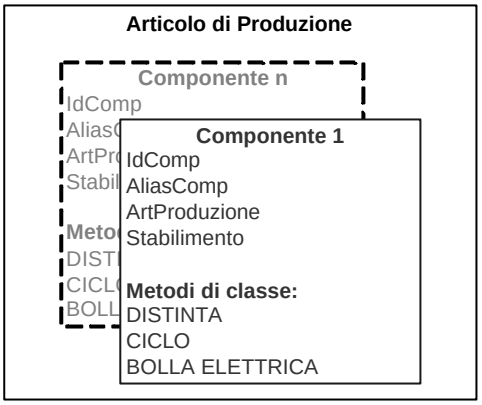


Per l'articolo si definisce una struttura UDT, con un supertipo che raccoglie gli attributi comuni tra elemento di produzione e materiale di acquisto. Il metodo legato all'articolo è la codifica, operazione eseguita su ciascun elemento introdotto nella base dati legata a regole aziendali stabilite per ciascuna tipologia di articolo.



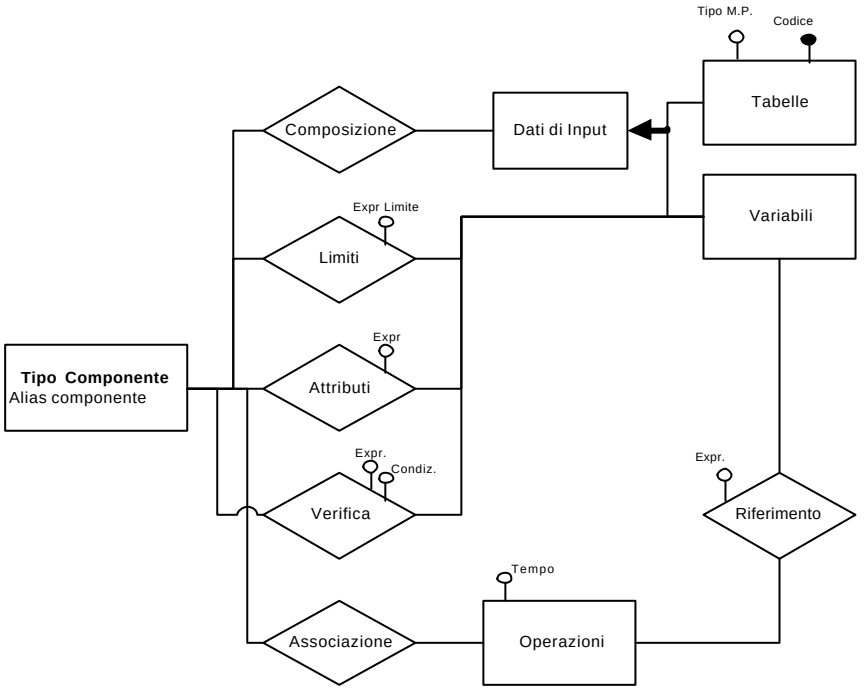
Per soddisfare alla richiesta di velocità di progettazione e di simulazione di nuovi codici, è stato implementato un configuratore: strumento software a disposizione dell'Ufficio Tecnico e, con funzionalità limitate, dall'Ufficio Vendite, che permette attraverso wizard guidati di progettare un nuovo prodotto partendo da caratteristiche tecniche inserite dall'utente e sfruttando modelli strutturati implementati da un tecnico senior.

Il configuratore alla fine della progettazione ritorna il prodotto strutturato come un array di componenti.



Il tipo componente è il modello da dove partire per configurare il nuovo prodotto; la sua gestione sfrutta le proprietà della classe degli oggetti.

Struttura Componente



Il tipo componente rappresenta la struttura parametrica del prodotto.

Il prodotto è composto da uno o più componenti, dove ciascun componente è strutturato con parametri in input, possiede limiti di impiego, attributi che ne definiscono le specifiche tecniche, condizioni da verificare se viene inserito all'interno di un altro

componente, nel qual caso modifica il valore ad ulteriori parametri, e contiene le operazioni che servono per essere prodotto.

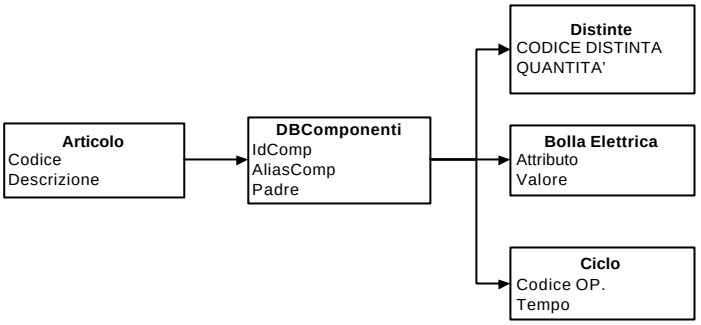
Nel modello così strutturato, l'"espressione" occupa un ruolo significativo nella struttura: il tipo *expr* con il metodo *risolvi_espressione*. La maggior parte degli attributi non possiedono un valore di tipo semplice, ma di tipo espressione. L'introduzione di questo elemento è dovuta alla struttura parametrica dei modelli implementati, e questo tipo permette di rendere il modello applicabile in diverse situazioni. I valori degli attributi sono, nella maggior parte dei casi, legati a degli algoritmi di calcolo dipendenti dalla struttura intrinseca del componente, dai materiali con cui può essere sviluppato e dalla forma che deve avere. Anche le tabelle di componentistica contengono attributi con valori semplici e attributi parametrici; ad esempio un materiale potrebbe essere utilizzato in più ambienti, aria, acqua e olio, e la temperatura che può raggiungere dipende dall'ambiente di utilizzo.

La caratteristica dell'oggetto espressione è data dal fatto che non contiene al suo interno solo espressioni di tipo matematico, ma anche variabili da calcolare in funzione del tipo progetto e dai componenti usati per produrlo. Gli algoritmi di calcolo dei tempi ciclo ad esempio presentano nel loro interno un riferimento anche al soggetto che devono trattare: l'operazione di punzonatura lamiera per il calcolo del tempo ciclo richiede il numero di pezzi che può ottenere per lamiera, il quale poi a sua volta dipende dalla dimensione dei pezzi.

I metodi di classe DISTINTA, CICLO e BOLLA ELETTRICA sono procedure che, dalla parametrizzazione del componente, restituiscono i valori di distinta, bolla e ciclo che descrivono il prodotto.

Nella struttura finale del prodotto codificato, gli attributi sono fissi; sono valori effettivi e non parametrici. Il legame tra prodotto elaborato e tipo di componente viene mantenuto per gestire le modifiche future. La modifica al prodotto richiede il ricalcolo dei parametri e le verifiche di consistenza con le nuove informazioni; questo tipo di struttura permette di ripercorrere tutto il processo di progettazione iniziale per riverificare la congruenza delle modifiche sempre attraverso il configuratore.

Legame Articolo - Componente



4.4 Alcune istruzioni dell'implementazione

```
CREATE TYPE expr AS CHARACTER VARYING(150)
METHOD risolvi_expr (idcomp INTEGER)
  RETURNS CHARACTER VARYING(50)

CREATE INSTANCE METHOD risolvi_expr(idcomp INTEGER)
  RETURNS CHARACTER VARYING(50)
  FOR expr
  <implementazione di risolve espressione
  sostituisce alle variabili il valore in funzione del componente>
RETURNS CHARACTER VARYING(50)

CREATE TYPE articolo
AS (
Codice          CHARACTER (19),
Descrizione     CHARACTER VARYING(30),
FlagAP          CHARACTER (1),
Costo           DECIMAL (9,2),
Unita_misura    CHARACTER(3),
Disegno CHARACTER(10)
)
INSTANTIABLE
```

```
NOT FINAL
METHOD codifica()
  RETURNS CHARACTER (19)
METHOD valorizzazione()
  RETURNS MONEY;

CREATE INSTANCE METHOD codifica()
  RETURNS CHARACTER (1)
  FOR articolo
RETURN CASE
  WHEN SELF.FlagAP = 'A' THEN <spCodiceMP>
  ELSE SELF.FalgAP = 'P' THEN <spCodicePF>
  END;

CREATE TABLE articolo OF articolo
  (REF IS id_articolo SYSTEM GENERATED)

CREATE TYPE articolo_prod
UNDER articolo
AS (
volt            INTEGER,
watt            INTEGER,
linea           INTEGER,
stato_bolla     CHARACTER(1)
)
INSTANTIABLE
NOT FINAL
METHOD upd_stato_bolla()
  RETURNS CHARACTER(1);

CREATE INSTANCE METHOD upd_stato_bolla(stato CHARACTER(1))
  RETURNS CHARACTER(1)
  FOR articolo_prod
RETURN CASE
```

```
  WHEN SELF.stato_bolla = 'S' AND stato = 'p' THEN 'p'
  WHEN SELF.stato_bolla = 'p' AND stato ='S'  THEN 'S'
        WHEN stato = 'p' THEN 'N'
  ELSE stato
  END;

CREATE TABLE articolo_prod OF articolo_prod
  (REF IS id_articoloprod SYSTEM GENERATED)

CREATE TYPE materia_prima
UNDER articolo
AS (
Codice          CHARACTER(19),
Costo           DECIMAL(9,2),
Peso            DECIMAL (9,4),
Pto_riordino    INTEGER,
scorta          INTEGER
)
INSTANTIABLE
NOT FINAL
METHOD dimensionamento()
  RETURNS EXPR;

CREATE INSTANCE METHOD   dimensionamento()
  RETURNS EXPR
  FOR material_prima
RETURN fcDimensiona(SELF.codice)

CREATE PROCEDURE distinta (
  IN padre CHARACTER(19),
  IN azienda CHARACTER(3),
  OUT distinta ARRAY(...))
LANGUAGE SQL
<implementazione>
```

END MODULE

Ricerca dei componenti di un prodotto finito:

```
SELECT C.ARTPRODUZIONE, C.IDCOMP, C.ALIASCOMP
FROM COMPONENTE C
WHERE C.ARTPRODUZIONE = <codice>
```

Ricerca degli elementi di distinta di un prodotto di produzione:

```
CALL distinta (<codice>, <azienda>)
```

Ricerca dei padri (prodotti finiti) che contengono un componente:

```
SELECT a.padre
FROM DISTINTE d, DBCOMPONENTI A
WHERE d.idcomp=a.idcomp
        AND d.[codice distinta] = <cod.componente>
```

Estrazione bolla elettrica di un componente

```
SELECT b.attributo, b.valore.risolvi_expr()
FROM [BOLLA ELETTRICA] b
WHERE b.idcomp=<id componente>
```

Modifica dello stato di bolla

```
UPDATE stato_bolla = b.upd_stato_bolla (<stato>)
FROM  articolo_prod b
WHERE b.codice = <codice>
```

Bibliografia

[1] E. Bertino, L.D. Martino, “Sistemi di basi di dati orientate agli oggetti”, Addison-Wesley Masson, 1992

[2] P. Atzeni, S. Ceri, S. Paraboschi, R. Torlone, “Basi di dati, Modelli e linguaggi di interrogazione”, McGraw-Hill, 2002

[3] P. Atzeni, S. Ceri, P. Fraternali, S. Paraboschi, R. Torlone, “Basi di dati, Architettura e linne di evoluzione”, 2003

[4] J. Melton, A. R. Simon, “SQL:1999 – Understanding Relational Language Components”, Morgan Kaufmann, 2001

[5] J. Melton, “Advanced SQL:1999 – Understanding Object Relational and Other Advanced Features”, Morgan Kaufmann, 2003

[6] G. Booch, “Object-Oriented Analysis and Design with Applications”, Benjamin/Cummings, 1994

[7] T. Budd, “An Introduction to Object-Oriented Programming”, Addison-Wesley, 1991

[8] K. Arnold, J. Gosling, D. Holmes, “Java, Manuale ufficiale”, Addison-Wesley, 2001

[9] Bruce Eckel, “Thinking in Java”, www.planetpdf.com

[10] M. Atkinson, F. Bancilhon, D. De Witt, K. Dittrich, D. Maier, S. Zdonik, “ The Object-Oriented Database System Manifesto“, Proc. First Int. Conf. On Deductive and Object-Oriented Databases, Kyoto, 1989

[11] M.Stonebraker, L.A. Rowe, B.G. Lindsay, J. Gray, M.J. Carey, M.L. Brodie, P.A. Bernstein, D. Beech, “Third-Generation Database System Manifesto – The Committee for Advanced DBMS Function”, ACM Sigmod, vol. 19, 1990

[12] www.service-architecture.com