Appendice B

UML e la modellazione di basi di dati non OO

Introduzione

Una delle lacune più serie imputate allo UML consiste nella mancanza di uno strumento formale per il disegno dello schema della base dati. Ora, esistono diverse tipologie di sistemi di gestione delle basi di dati (*DBMS*, Data Base Management System): relazionali, a oggetti, reticolari, gerarchici e funzionali. Gli unici a non risentire della lacuna sono quelli basati sul paradigma OO. Per questi la rappresentazione dello schema della base dati è ottenibile attraverso il formalismo dei diagrammi delle classi. Lo stesso schema dovrebbe essere derivabile direttamente dal modello a oggetti del dominio e dovrebbe consistere in un opportuno sottoinsieme di package del modello di disegno.

Il problema invece si pone per il disegno dello schema di base dati fondati sulle altre tipologie di DBMS.

Ma è proprio vero che lo UML non fornisce alcuno strumento per il disegno dello schema di basi di dati non fondate sul paradigma OO? La risposta non è così immediata. Se da un lato è vero che non esiste uno specifico strumento, è altresì vero che lo UML fornisce una serie di strumenti di estensione (stereotipi, valori etichettati e vincoli) che permettono di specializzarne l'utilizzo.

Dallo UML 1.4, l'insieme di nuovi elementi introdotti per specializzarne l'utilizzo dovrebbe essere organizzato opportunamente in un apposito profilo di estensione.

Nei seguenti paragrafi, per questioni di praticità e di maggiore diffusione, viene considerata esclusivamente la tipologia di base dati relazionale. Chiaramente non si intende affrontare l'argomento in maniera esaustiva, ma si vuole fornire una serie di idee al fine di rendere i lettori

interessati all'argomento in grado di definire un profilo atto a risolvere le proprie necessità di disegno di schemi di basi di dati.

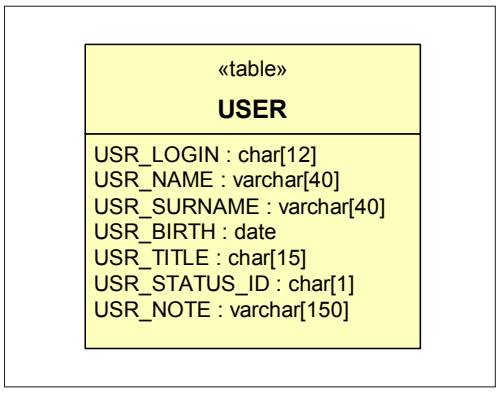
L'esempio presentato è tratto dal modello a oggetti del Capitolo 8. Questo dovrebbe favorire il conseguimento dell'obiettivo previsto: fornire elementi per la modellazione di un profilo per il disegno delle basi dati riducendo al minimo le nozioni da presentare relative all'area oggetto di analisi.

Tabelle

In primo luogo è necessario selezionare lo strumento dello UML più idoneo per il disegno e la rappresentazione di schemi di basi di dati. In questo caso non vi sono molte possibilità: si tratta di un diagramma statico di struttura focalizzato su astrazioni di tipo logico. Pertanto il formalismo da considerare è quello dei diagrammi delle classi. Logica conseguenza è l'utilizzo di un apposito stereotipo dell'elemento classe (denominato «`table`») per identificare le singole tabelle. Il dipartimento degli attributi si presta a essere utilizzato come luogo naturale in cui specificare le colonne della tabella (queste, anche se non evidenziato esplicitamente, sono stereotipi dell'elemento UML `attribute` denominati «`column`»). Per quanto attiene al dipartimento dedicato ai metodi, per il momento, può essere trascurato (considerare fig. B.1).

È buona norma per il disegno della base dati, attribuire un identificatore univoco alle tabelle, da utilizzarsi come prefisso per i nomi delle colonne. Nel caso in figura si è utilizzato per la tabella `USER` l'identificatore `USR`. Questo agevola il compito di riferirsi univocamente ai campi, senza dover necessariamente specificare sempre la tabella di appartenenza.

Figura B.1 — *Utilizzo dello stereotipo* **«`table`»** *dell'elemento classe.*



Relativamente alle colonne, non ha interesse mostrare esclusivamente nome, tipo e dimensione, ma anche tutta una serie di informazioni supplementari. Per esempio, tipicamente, è importante sapere se si tratta o meno di una chiave primaria, se è ammesso il valore `null`, e così via. Queste informazioni si prestano a essere specificate per mezzo dei meccanismi di estensione forniti dallo UML: stereotipi e valori etichettati (`tagged value`, *cfr* Capitolo 2).

Si inizi con il considerare le informazioni più semplici. In primo luogo si può considerare che per default nessun campo ammetta valori `null` e che l'eventualità che ciò sia possibile venga evidenziata attraverso il valore etichettato `{null=true}`, o più semplicemente `{null}`, posto a fianco del campo.

Per quanto riguarda l'indicazione di chiavi primarie (`primaryKey`), chiavi importate o straniere (`foreignKey`) e secondarie (`secondaryKey`), un buon sistema consiste nel considerare nuovi stereotipi specializzanti l'elemento «`column`». Il nome di tali stereotipi potrebbe essere, rispettivamente: «`primaryKey`», «`foreignKey`» e «`secondaryKey`».

Si ricordi brevemente che la relazione tra due tabelle, nelle basi dati relazionali, si ottiene esportando la chiave primaria di una tabella nell'altra, in cui prende il nome di chiave straniera (`foreignKey`).

Ora un primo problema è relativo al fatto che, spesso, le chiavi di una tabella possono essere composte da più colonne. Questo problema può essere risolto utilizzando in associazione degli stereotipi appositi valori etichettati. Per esempio `{xKeyPart=1}`, `{xKeyPart=2}`, … dove il carattere `x` andrebbe sostituito con le lettere `p`, `f` o `s` a seconda che si tratti, rispettivamente, di chiave primaria, straniera o secondaria.

Chiaramente qualora non vi sia specificato alcun valore etichettato, va assunto che la chiave non sia composta, ovvero che sia costituita da un'unica colonna.

Un secondo problema potrebbe nascere qualora ci fossero più chiavi.

> Nel caso di chiavi primarie, a dire il vero, non dovrebbe mai accadere, altrimenti potrebbe essere il caso di aver prodotto un errato disegno della base dati. Situazioni del genere, tipicamente, sono conseguenza dell'inglobamento in un'unica tabella di diverse entità. Questa anomalia è riscontrabile dalla presenza di dipendenze funzionali (alcuni attributi dipendono da una chiave, un altro gruppo da un'altra, ecc.). La 3BCNF (terza forma normale di Boyce e Codd) si occupa proprio di verificare e quindi risolvere dipendenze funzionali.
> In ogni modo situazioni del genere potrebbero essere risolte strutturando ulteriormente i valori etichettati associate ai vari stereotipi:
>
> `{pKey, id=A, part=1}`, `{pKey, id=A, part=2}`, `{pKey, id= B, part=1}`, ecc.
>
> La situazione si complica decisamente. La notizia buona è che la necessità di ricorrere a tali livelli di dettaglio è poco frequente.

Quanto riportato per la chiave primaria rimane perfettamente valido per chiavi straniere e per chiavi secondarie di ricerca qualora si decida di evidenziarle (probabilmente è più corretto mostrare queste informazioni attraverso opportuni indici, come riportato successivamente).

Per le chiavi straniere potrebbe essere opportuno riportare la tabella di provenienza. Questo risolverebbe il problema di dover specificare l'identificatore per distinguere le diverse chiavi straniere eventualmente presenti in una stessa tabella. Qualora poi una chiave straniera sia presente in diverse tabelle, dovrebbe essere sempre considerata come tabella di provenienza quella in cui recita il ruolo di chiave primaria.

Dall'analisi del diagramma di fig. B.2 è possibile effettuare alcune considerazioni. Nella tabella `USER` è stata evidenziata una chiave secondaria composta. Poiché è l'unica, è stato sufficiente riportare l'indicazione delle parti che la costituiscono (`{part=1}` e `{part=2}`). Nella modellazione, è sempre opportuno utilizzare un approccio flessibile: qualora non ci sia alcuna possibilità di confusione è appropriato non complicare troppo le cose.

Più importante invece è l'anomalia presente nella tabella `PROFILE`. La prima parte della chiave primaria (`PRF_USR_LOGIN`) è contemporaneamente chiave straniera, informazione che però non emerge dal diagramma. Ciò è risolvibile adottando uno dei due approcci seguenti:

1. utilizzare valori etichettati in associazione con le chiavi primarie per specificare la loro essenza di chiavi straniere;

2. dar luogo a un nuovo stereotipo («primaryFKey») che eredita sia da «primaryKey», sia da «foreignKey» (ereditarietà multipla).

La soluzione ritenuta più opportuna è la seconda.

Figura B.2 — *Utilizzo degli stereotipi* ***primaryKey***, ***foreignKey*** *e* ***secondaryKey***.

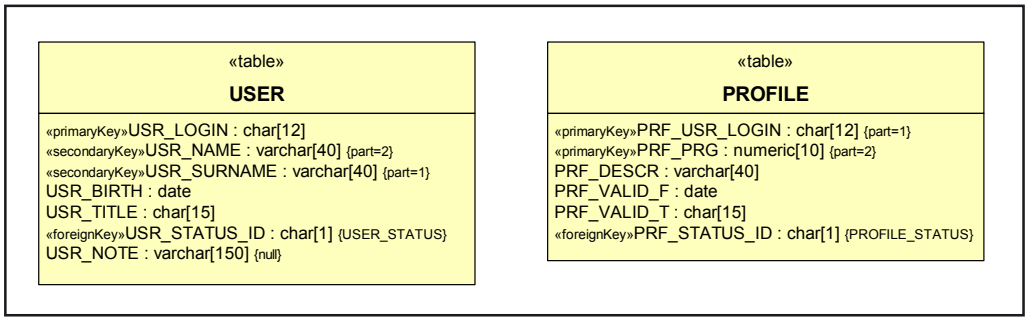
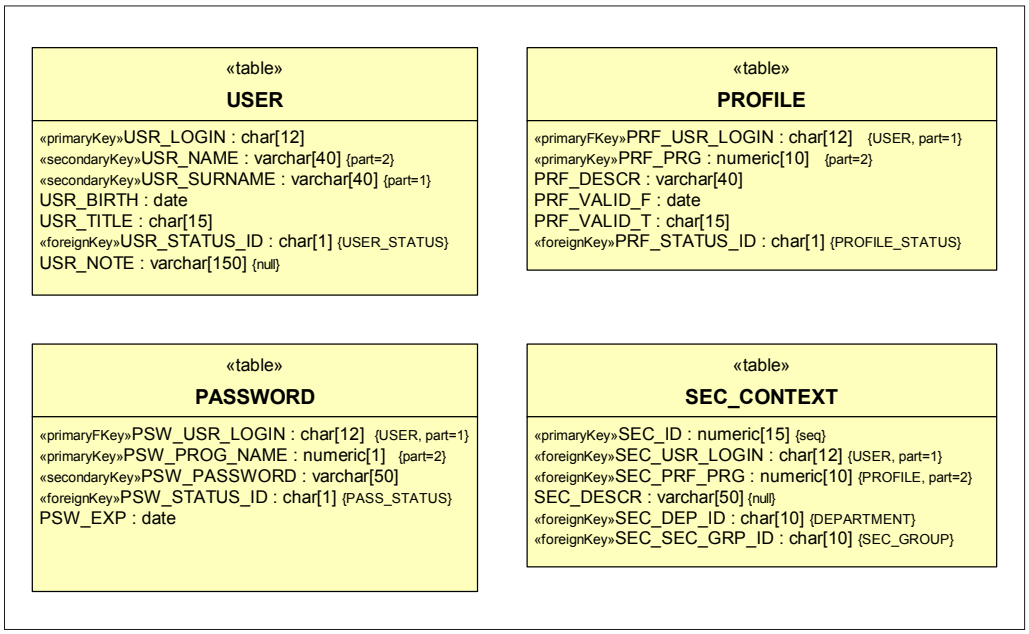


Figura B.3 — *Introduzione dello stereotipo* ***primaryFKey****.*



Brevemente, il diagramma di fig. B.3 sancisce che un utente può disporre di diverse password (evidentemente solo una nello stato operativo) mentre ciascuna di esse può essere riferita solo a un utente. Questo inoltre può disporre di diversi *profiles*, di cui solo uno nello stato attivo, e ciascuno di essi è riferito a un solo utente. Infine un profilo è costituito da diversi contesti di sicurezza, in cui ciascuno di essi associa un gruppo di sicurezza a uno specifico dipartimento.

Attraverso l'utilizzo dei valori etichettati è possibile specificare tutta una serie di informazioni aggiuntive. Da tener presente che sebbene dati significativi forniscano importanti informazioni, quando essi sono in eccesso sortiscono l'effetto contrario. Si corre il rischio di rendere il diagramma confuso, poco leggibile, e quindi si risolvono esclusivamente in elementi di disturbo: si passa da informazioni a meri dati.

Nel diagramma di fig. B.3 si è ritenuto utile evidenziare l'eventualità i cui i valori delle chiavi primarie siano generati dal DBMS oppure fornite (in qualche modo) dagli utenti. Nel primo caso, per esempio con il DBMS Oracle, questo fornisce il meccanismo dei *sequence* per generare valori sequenziali. Pertanto, qualora una chiave principale sia un numero sequenziale generato dal sistema, potrebbe essere utile specificare un apposito valore etichettato `{seq}`.

Relazioni

Secondo le prescrizioni delle basi di dati relazionali, le relazioni tra tabelle si ottengono attraverso l'esportazione delle chiavi primarie. In particolare, se una tabella è relazionata a un'altra, è necessario inserire la chiave primaria di quest'ultima nella prima, nella quale prende il nome di chiave straniera. Per esempio nella tabella `PROFILE` è presente l'identificatore univoco dell'utente cui appartiene.

Da notare che il legame si ottiene con un criterio opposto rispetto alla modellazione a oggetti: in questo caso gli indirizzi dei profili appartenenti a uno specifico utente sono presenti nell'oggetto utente.

Ora l'impostazione di appositi valori etichettati a fianco dei vari attributi potrebbe essere già di per sé sufficiente per specificare le varie relazioni tra le tabelle. Ciò nonostante si preferisce mostrare più esplicitamente tali legami al fine di aumentare il grado di chiarezza e di immediatezza di fruizione del modello.

A tal proposito ci sono alcune alternative circa la tipologia di relazione da utilizzare. In particolare è possibile ricorrere alle relazioni di dipendenza o di associazione. Quantunque esistano valide argomentazioni per entrambe le soluzioni, l'autore preferisce la seconda.

Figura B.4 — *Relazioni tra oggetti e tra tabelle in database relazionali. Da notare che se nella relazione che unisce la classe* ***`User`*** *a quella* ***`Profile`*** *non fosse stato presente il vincolo di navigabilità, anche gli oggetti istanza della classe* ***`Profile`*** *avrebbero dovuto memorizzare l'indirizzo dell'unico oggetto* ***`User`*** *di appartenenza.*

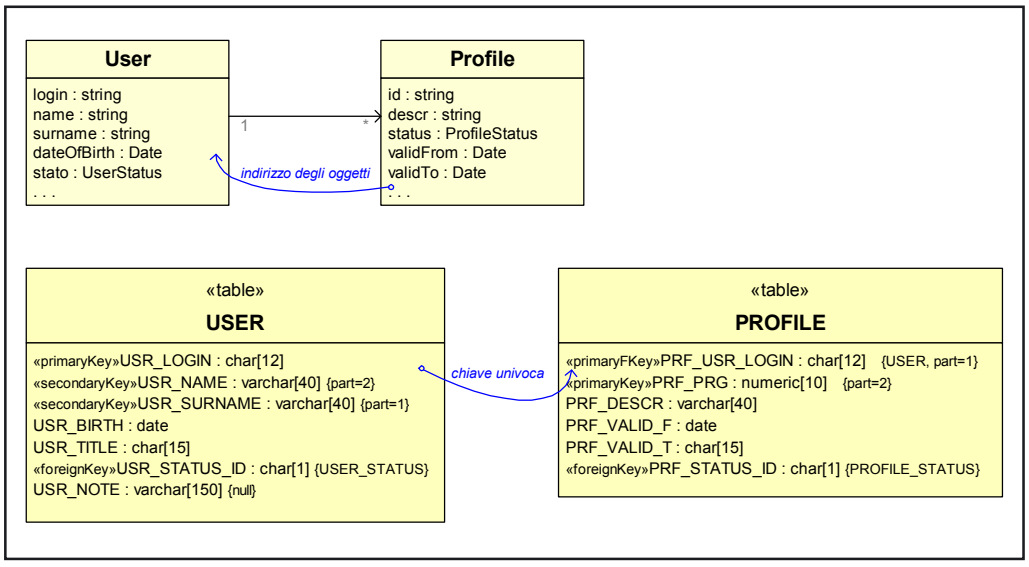
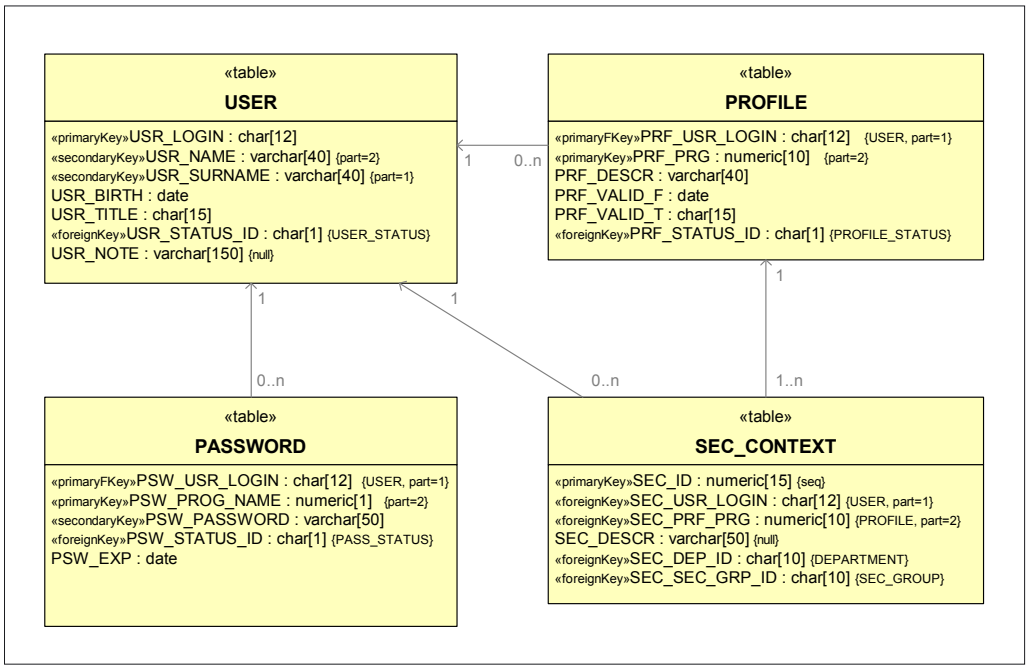


Figura B.5 — *Utilizzo della relazione di associazione per visualizzare legami tra tabelle.*



A sostegno di questa scelta esistono diversi motivi. In primo luogo si tratta di un legame strutturale, poi è possibile evidenziare eventuali navigabilità in entrambe le direzioni, si può mettere in luce la molteplicità, si può associare un nome. Per quanto concerne la navigabilità, se una tabella esporta la propria chiave primaria in un'altra, evidentemente dai record di quest'ultima è possibile navigare in quelli della prima, ma non viceversa.

Eventualmente è possibile specificare sulla relazione i campi che rendono possibile la navigazione. Si tratterebbe però di un'informazione ridondante, già abbondantemente specificata nelle classi.

Per concludere il presente paragrafo, vale la pena di ricordare brevemente che nei database relazionali non è possibile rappresentare relazioni `(n,n)`. Ciò è ottenibile per mezzo dell'introduzione di opportune tabelle di legame che decompongano relazioni `(n,n)` in due relazioni `(n,1)` e `(1,n)`.

Indici

Brevemente, gli indici sono delle strutture create sulle tabelle al fine di migliorare le prestazioni della ricerca. Questo miglioramento delle prestazioni si ripercuote su opera-

zioni di selezione, aggiornamento e cancellazione di record. I DBMS normalmente creano automaticamente indici per la chiave primaria. È opportuno che il disegnatore della base dati dichiari eventuali indici per altri campi utilizzati per la selezione dei record. Chiaramente la creazione e manutenzione degli indici ha un costo e quindi, prima di creare un indice, andrebbero considerati fattori come la dimensione della tabella a cui si riferirebbe il nuovo indice, quanto le performance siano stringenti, quanto frequentemente dovrebbe venir eseguita l'operazione migliorata dall'introduzione dell'indice, ecc.

Gli indici possono essere creati per un singolo campo o per un insieme. Eventualmente è possibile specificare funzioni che agiscono su tali campi, come per esempio `UPPER()` (trasforma i caratteri in maiuscolo). Le informazioni relative agli indici possono essere utili, ma sicuramente non sono di importanza fondamentale. Anche in questo caso è possibile ricorrere a diverse alternative. Le principali consistono nel ricorrere a

- ulteriori stereotipi dell'elemento classe (etichettati «index») per mostrare tutti i dettagli degli indici associati alla tabella a cui si riferiscono. Questo offre il vantaggio di mostrare un maggior numero di informazioni. Gli svantaggi invece sono che il diagramma diventa molto confuso (ogni tabella ne prevedrebbe altre per gli indici), non è possibile decidere se visualizzare o meno queste informazioni, ecc.;

- utilizzare la sezione dedicata ai metodi, assumendo che si tratti di indici. Questa soluzione presenta l'unico svantaggio di non poter mostrare tutte le informazioni relative agli indici. I vantaggi sono invece che è possibile mantenere il diagramma elegante e chiaro, è possibile decidere se visualizzare o meno le informazioni relative agli indici (escludendo o visualizzando la sezione delle operazioni), e il diagramma è più facile da produrre e manutenere.

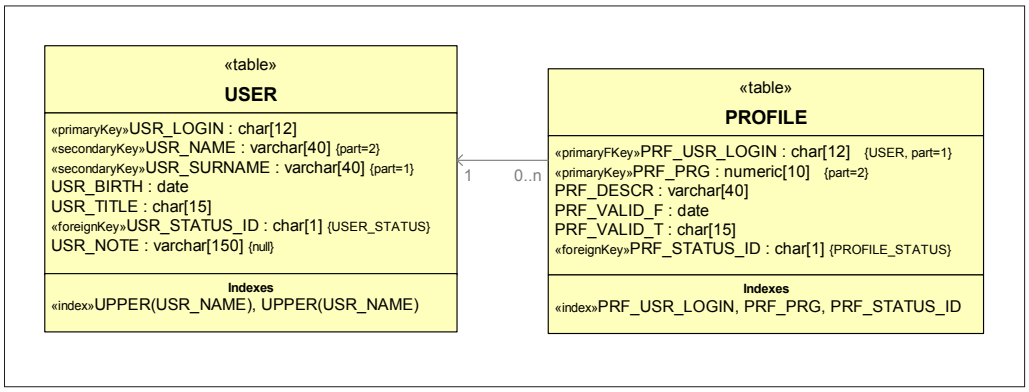
Chiaramente la soluzione ritenuta migliore è la seconda.

Dall'analisi di fig. B.6 è possibile notare che la tabella degli `USER` prevede due indici: uno implicito relativo alla chiave univoca `USR_LOGIN` e un secondo indice utilizzato per poter reperire gli utenti in base a cognome e nome. Per quanto concerne la tabella `PROFILE`, si hanno nuovamente due indici: quello relativo alla chiave primaria (`PRF_USR_LOGIN` e `PRF_PRG`) e un altro utilizzato per selezionare, per uno specifico utente, il profilo di che si trova in un determinato stato (per esempio `ACTIVE`).

I trigger

I gestori dei database permettono di specificare operazioni da eseguire qualora si verifichino particolari eventi: inserimento, aggiornamento e cancellazione di un record. Pertanto, alcune volte potrebbe essere utile mostrare la presenza di tali procedure nel modello dello schema della base dati.

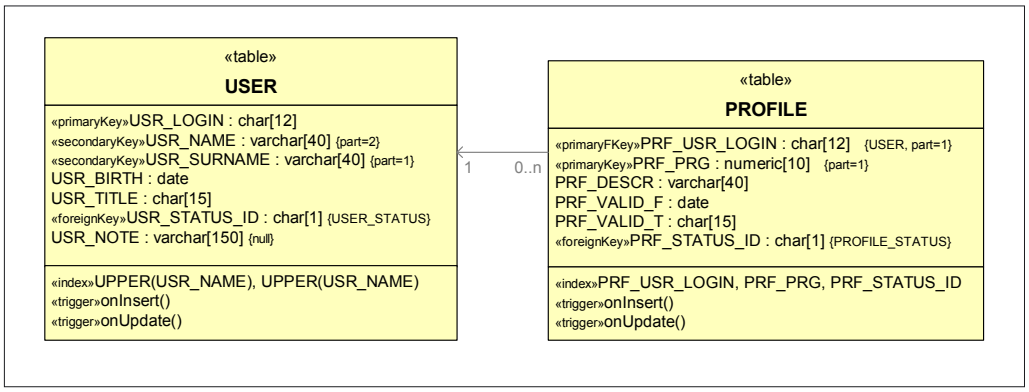
Figura B.6 — *Visualizzazione degli indici.*



Qui la situazione si complica. In primo luogo bisogna identificare un apposito dipartimento. Sebbene le direttive dello UML prevedano la possibilità di definire dipartimenti supplementari da associare all'elemento classe, raramente i tool supportano tale funzionalità. L'unica alternativa pertanto consiste nel condividere il dipartimento dedicato ai nomi con i vari metodi. Al fine di evitare confusione, è possibile utilizzare gli stereotipi «`index`» per evidenziare indici e «`trigger`» per i trigger. Questi ultimi possono essere esclusivamente `onInsert`, `onUpdate` e `onDelete`. Ora, per quanto attiene alla specificazione dei contenuti, si tratterebbe del problema analogo di dettagliare il comportamento dei metodi. Un'alternativa potrebbe essere quella di utilizzare appositi campi note. Se da un lato questa tecnica permetterebbe di ottenere il risultato voluto, dall'altro renderebbe il diagramma veramente complicato da leggere. Le uniche possibilità percorribili si riducono a specificare il contenuto dei trigger in apposite sezioni non visualizzabili, prevederne unicamente la presenza oppure non specificarli del tutto. Probabilmente l'alternativa migliore è la seconda: essere consapevoli della presenza di un trigger è comunque situazione migliore rispetto a non averne alcuna conoscenza.

Diverse architetture prevedono un funzionamento basato sulla tecnica dell'*optmistic locking* (blocco ottimistico). In altre parole quando un utente richiede di reperire un record per poterlo modificare, questo non viene bloccato (*lock*) inibendo (temporaneamente) altre scritture. In sostanza si assume che, nel lasso di tempo tra il momento in cui un utente seleziona un record e quello in cui conferma gli aggiornamenti (oppure il *lock* viene rimosso per *time-out*), il record in questione non venga modificato da altri processi. Questa assunzione, con le giuste cautele e i dovuti controlli, è legittima e particolarmente utile, specie quando si realizzano sistemi web-based. Però è necessario realizzare una serie di tecniche atte ad assicurarsi che le modifiche apportate non vadano a ricoprire eventuali altre effettuate nel lasso di tempo di cui sopra. Al fine di poter effettuare tale controllo, una tecnica è quella

Figura B.7 — *Visualizzazione trigger.*



di inserire una colonna artificiale per ogni tabella aggiornabile concorrentemente atta a memorizzare il *timestamp* dell'ultimo aggiornamento. Pertanto la selezione di un record comporta la memorizzazione del l'informazione relativa all'ultimo aggiornamento (*timestamp*). A questo punto, per assicurarsi che il record non sia stato aggiornato tra il momento in cui viene selezionato e quello in cui deve essere modificato, è sufficiente verificare che il *timestamp* letto non sia cambiato. Volendo sfruttare questa tecnica, risulta una buona idea delegare al DBMS la gestione dei *timestamp*. A tal fine è sufficiente specificare *triggers* da attivare all'atto dell'inserimento e dell'aggiornamento dei record.

Conclusioni

In questa Appendice si è mostrato un semplice esempio di profilo utilizzabile per modellare, attraverso lo UML, strutture di base dati. Lungi dal voler considerare la trattazione esaustiva, si è pensato di fornire alcuni spunti utili per realizzare un profilo formale, funzionale alle peculiari necessità dei vari ambienti.

Si è anche dimostrato che, se da un lato è vero che lo UML non prevede alcun meccanismo formale per disegnare e rappresentare lo schema delle basi dati di sistemi non OO, dall'altro è possibile aggirare tale lacuna creando appositi profili tramite un intelligente utilizzo dei meccanismi di estensione standard.

Da notare che qualora esistesse un profilo di questo tipo, approvato dallo OMG, potrebbe essere possibile, attraverso l'utilizzo di un qualsiasi tool, generare direttamente gli script per la creazione e la gestione della base dati.