Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Triennale in Ingegneria Informatica

Progettazione e realizzazione di un sistema di gestione di gare ciclistiche amatoriali basato su architettura Hibernate

Laureando:
Paolo Zinato

Relatore:
Prof. Giorgio Maria
Di Nunzio

Anno accademico 2012/2013

Indice

Capitolo 1

Introduzione

Il ciclismo amatoriale su strada è una realtà che raccoglie ogni anno un gran numero di sportivi, soprattutto in alcune regioni d'Italia, come ad esempio il Veneto. Il numero dei partecipanti a questa realtà è in continua crescita: l'ente ciclistico UISP nel 1995 registrava circa 34 mila persone tesserate, mentre ora ne registra più 46 mila[1], con un incremento annuo medio del 2% circa. I gruppi ciclistici che aderiscono a tale ente sono più di 2400, dei quali solo in Veneto sono più di 200. Le principali associazioni sportive ciclistiche hanno molteplici comitati regionali/provinciali sparsi in tutto il territorio nazionale, che promuovono le attività sportive a livello locale. Questi comitati molto spesso sono associazioni no-profit che soffrono di scarsità di risorse umane, di finanziamenti e di sistemi informativi efficienti. I sistemi informatici adottati per la gestione delle attività ciclistiche (tesseramenti, gare e manifestazioni, classifiche ecc..), sono spesso rudimentali e obsoleti, se non assenti: di solito molto lavoro viene fatto su carta o con l'utilizzo di fogli di calcolo. Il calo del numero dei volontari e del tempo che questi dedicano a tali attività è un grosso problema della realtà sportiva amatoriale d'oggi. In questo contesto, la scarsa informatizzazione fa da aggravante, perché rende ancora più laborioso e impegnativo il lavoro di queste persone. Si corre il rischio che i comitati meno innovativi e più piccoli non riescano più a gestire gli eventi sportivi, rischiando dopo molti anni di attività di collassare e sciogliersi, privando il territorio di una tradizione ben radicata.

Questo progetto non si propone certo come una soluzione a tale problema ma vuole essere d'aiuto ai piccoli comitati e al loro ente d'appartenenza, realizzando uno strumento che li aiuti nella gestione delle attività. Il lavoro è stato realizzato per un comitato ciclistico amatoriale dell'ente UISP della provincia di Venezia, ma è estendibile anche a molti altri. Si è voluto progettare un sistema efficiente, flessibile ed economico per la gestione efficace dei tesseramenti annuali, dei trofei e delle gare, delle iscrizioni e delle relative classifiche. Tale sistema è essenzialmente una applicazione realizzata in linguaggio Java che utilizza un Database Management Systems (DBMS) per la memorizzazione dei dati. Data la complessità e la quantità dei dati in gioco, l'utilizzo di un database è certamente la soluzione

[1] Dati pubblicati sul sito web dell'ente UISP: www.uisp.it/ciclismo

migliore in termini di efficienza e di utilità.

Il punto centrale di questo progetto è la gestione della persistenza in Java. Questa è stata affidata ad **Hibernate**, una piattaforma *middleware* e *open source* che permette di gestire e automatizzare il salvataggio e il reperimento degli oggetti dall'applicazione Java a un database relazionale, esonerando lo sviluppatore dall'intero lavoro relativo alla persistenza dei dati. Hibernate gestisce in maniera automatica le cosiddette operazioni CRUD (Create, Read, Update, Delete) dei database, sollevando in gran parte lo sviluppatore da recupero manuale dei dati e dalla loro conversione. In generale, nell'ambito dei sistemi informativi, queste operazioni sono quelle che più di altre portano lavoro di codifica, di test e di debug. Ridurle significativamente permette non solo di sviluppare una applicazione in breve tempo, ma anche di abbassarne il relativo costo. Un altro grande vantaggio che questo strumento porta è quello di mantenere l'applicazione portabile in tutti i database SQL, rendendola estremamente flessibile.

Prima di concentrarsi sui dettagli progettuali e implementativi del lavoro svolto, si vuole brevemente presentare il progetto definendo la sua struttura e quali strumenti sono stati utilizzati per la sua realizzazione.

1.1 Presentazione del progetto

Lo sviluppo di questo progetto è stato suddiviso in due parti: nella prima è stata affrontata la progettazione dei dati, nella seconda la progettazione dell'applicazione.

La progettazione della base di dati è stata quindi il punto di partenza dell'intero progetto ed é stata articolata secondo una metodologia ben definita che prevede quattro fasi di sviluppo da effettuare in cascata:

1. Analisi dei requisiti
2. Progettazione concettuale
3. Progettazione logica
4. Progettazione fisica

Ognuna di queste fasi si basa su un modello che permette di generare una rappresentazione formale ad un dato livello di astrazione (concettuale, logico e fisico). Ad ogni fase è stato dedicato un capitolo di questa tesi.

Lo sviluppo dell'applicazione è iniziato con la realizzazione degli oggetti Java in stile Javabean che rispecchiano le entità e le associazioni già definite nello sviluppo della base di dati. Queste particolari classi presentano una serie di campi rappresentanti le proprietà dell'oggetto ed i relativi metodi d'accesso get/set. La

struttura così realizzata permette di incapsulare negli oggetti Javabean i dati contenuti nelle tabelle della base di dati, rendendoli disponibili all'interno dell'applicazione.

Successivamente, è stato necessario introdurre uno strato intermedio di progettazione, quello relativo alla mappatura ORM (Object/Relational Mapping), così da permettere il trasferimento dei dati tra l'applicazione e il database. La scelta caratterizzante è stata quella di adottare come soluzione ORM il framework **Hibernate**. Una volta configurato e inizializzato correttamente, questo strumento ha automatizzato in gran parte l'operazione di mapping, evitando così di scrivere manualmente molte righe di codice SQL e JDBC per provvedere al recupero e alla conversione dei dati.

Terminata la fase di modellazione dei dati, si è provveduto a creare uno strato d'accesso ai dati con l'utilizzo del pattern DAO. L'idea si basa sulla possibilità di concentrare il codice per l'accesso al sistema di persistenza in una classe che si occupa di gestire la logica di accesso ai dati separandola da quella di business (applicativa).

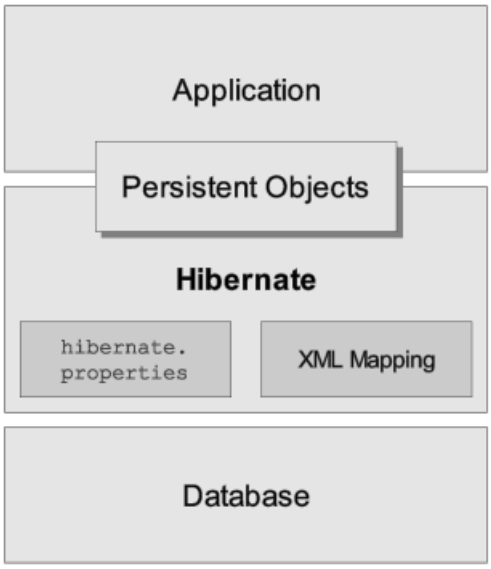


Figura 1.1: Struttura dell'applicazione con l'utilizzo Hibernate.

1.2 Strumenti utilizzati

Di seguito verranno brevemente introdotti gli strumenti che sono stati utilizzati per la realizzazione di questo progetto.

- **Microsoft Visio 2007**. Questo software è stato utilizzato per la realizzazione degli schemi della progettazione concettuale e logica della base di dati. E' un software sviluppato da Microsoft per i sistemi operativi Windows.

- **PostgreSQL**. Questo software nella versione 9.2 è stato utilizzato per la creazione della base di dati. E' un DBMS relazionale open-source e si propone come una reale alternativa rispetto ad altri prodotti come MySQL. Ha più di 15 anni di sviluppo attivo e offre caratteristiche uniche nel suo genere che lo pongono per alcuni aspetti all'avanguardia nel settore dei database.

- **PgAdmin 3**. E' una interfaccia grafica che consente di amministrare in modo semplificato un database PostgreSQL. L'applicazione è indirizzata sia agli amministratori del database sia agli utenti.

- **Geany 1.20**. Geany è un leggero editor di testo multi-piattaforma con funzionalità elementari di IDE. È disponibile per un gran numero di sistemi operativi e supporta molteplici linguaggi di programmazione. È stato utilizzato nella fase di progettazione fisica come editor SQL.

- **Eclipse**. E' un IDE per la programmazione, offre un ambiente di sviluppo integrato multi-linguaggio e multi-piattaforma. Per la realizzazione del progetto Java è stata utilizzata la versione "Juno".

- **TeX Live 2012/Debian, Gostscript, TeXmaker 3.4**. Per la stesura della tesi è stato utilizzato LaTeX. In particolare la distribuzione usata è quella TeX Live 2012/Debian, mentre come editor è stato scelto Texmaker nella sua versione 3.4.

Capitolo 2

Analisi dei requisiti

La prima fase di progettazione di una base di dati è la raccolta e l'analisi dei requisiti. Allo scopo di individuare le proprietà e le funzionalità che dovrà avere la base di dati, vengono interrogati i futuri utenti riguardo i loro requisiti sui dati. In generale questi si suddividono in tre tipologie:

- Requisiti informativi: caratteristiche dei dati
- Requisiti sui processi: operazioni sui dati
- Requisiti sui vincoli di integrità: proprietà dei dati e delle operazioni

Il risultato della fase di raccolta è, quindi, un insieme di informazioni e descrizioni specifiche, complete ma generalmente informali, dei dati coinvolti e delle operazioni su di essi. Queste permettono di capire come modellare i dati nelle successive fasi di progettazione, dove le informazioni raccolte verranno analizzate in maniera approfondita. Questa fase è solo apparentemente semplice: nella realtà è spesso la più complessa perché il processo che porta a capire cosa gli utenti effettivamente richiedono è difficilmente standardizzabile.

2.1 Caratteristiche e requisiti

Le informazioni raccolte a seguito dell'intervista al committente si possono riassumere nel seguente modo:

- Si vogliono catalogare le generalità e i recapiti di tutti i tesserati con l'ente ciclistico, ma anche di chi è tesserato con altri enti ma partecipa alle gare ad iscrizione aperta. Tutti coloro che vogliono tesserarsi devono essere iscritti ad una società sportiva ciclistica. Nella tessera, quindi, vengono specificati la società di appartenenza, il tipo di iscrizione effettuata e i dettagli della tessera (il tipo di polizza e la categoria di appartenenza). Ogni tesserato può svolgere più di una attività sportiva. Tutti i tesserati che hanno un tipo di tessera valido per l'attività amatoriale possono partecipare alle gare. Vengono tesserati anche i giudici di gara che differiscono dai ciclisti solo nel tipo di tessera.

- Ai ciclisti che partecipano alle gare, sia interni che esterni all'ente, viene assegnato un numero di gara valido per tutto l'anno ciclistico; inoltre essi vengono inseriti in un gruppo secondo la categoria di appartenenza (che dipende dall'età anagrafica del tesserato). E' possibile che alcuni trofei prevedano ulteriori suddivisioni dei ciclisti in fasce (suddivisione eseguita d'ufficio) e quindi l'appartenenza a sotto-gruppi.

- Le gare ciclistiche sono di norma associate a un trofeo o a un campionato e vi partecipano i ciclisti amatori suddivisi nei corrispettivi gruppi. Le gare sono formate da diverse corse che differiscono per la lunghezza del percorso; ad ognuna partecipano determinati gruppi. Alcuni trofei sono ad iscrizione chiusa, ovvero ammettono solo ciclisti tesserati con l'ente, altri invece sono ad iscrizione libera e ammettono ciclisti tesserati con enti diversi (ma sempre muniti di una tessera amatoriale valida).
  Di ogni gara, relativamente ad ogni trofeo, è richiesto di memorizzare la società sportiva organizzatrice, la denominazione, il luogo e la data di svolgimento.

- Di ogni gara vengono registrati gli ordini di arrivo delle varie corse e in più eventuali abbuoni/traguardi volanti vinti dai ciclisti. Gli abbuoni vengono assegnati in vari momenti della gara e di norma ai primi tre in testa. Per ogni corsa, inoltre, vengono registrati i dettagli: la lunghezza del percorso, il tempo impiegato, l'ora di partenza e la media oraria del gruppo.

- Alla conclusione di ogni gara viene redatto il verbale. Quest'ultimo registra tutti i dettagli della gara e delle singole corse, i giudici presenti, il numero dei veicoli a seguito della gara, il medico responsabile, gli eventuali infortuni e reclami da parte degli atleti in gara ed eventuali provvedimenti disciplinari.

- Le classifiche relative al rispettivo trofeo, al termine di ogni gara, vengono calcolate e aggiornate. Sono previsti tre tipi di classifiche: due individuali (una relativa ai piazzamenti e una relativa agli abbuoni) e una a squadre (calcolata sommando i punteggi di tutti i propri iscritti).

2.2 Operazioni di inserimento

Le operazioni di inserimento prendono in considerazione tutte le entità e tutti i loro attributi (a parte certe chiavi auto-incrementanti). Riguardo la frequenza annua di tali operazioni, queste saranno molto frequenti nei primi mesi quando vengono aperti i tesseramenti e programmate le gare e i trofei. Nei mesi successivi, gli inserimenti sono di norma limitati alle gare e alle relative iscrizioni.

2.3 Operazioni di interrogazione

La frequenza delle operazioni sotto riportate è indicativa, ed è stata stimata facendo riferimento all'utilizzo di una *applicazione grafica* come interfaccia tra DBMS

e utente. Questa, oltre alla possibilità di effettuare interrogazioni al DB, offre già all'utente molteplici informazioni e statistiche relative allo stato degli inserimenti.

Legenda: MF = Molto frequente, F = Frequente, R = Raro, MR = Molto raro.

Operazioni relative al tesseramento

Operazione	**Freq.**
Tutti gli attributi di un ciclista e i tutti i dettagli di tesseramento	MF
Conteggio del numero totale di ciclisti tesserati	MF
Conteggio del numero totale di ciclisti tesserati divisi per attività	R
Conteggio del numero totale di ciclisti tesserati divisi per gruppo	R
Conteggio del numero totale di ciclisti tesserati divisi per società	R
Conteggio del numero totale di ciclisti tesserati divisi per categoria	R
Conteggio di tutte le società tesserate	R

Operazioni relative ai trofei

Operazione	**Freq.**
Visualizzazione di tutti i trofei presenti in anno ciclistico con tutti gli attributi	R
Visualizzazione di tutte le gare presenti in una edizione del trofeo con tutti gli attributi	R
Conteggio di tutte le gare presenti in una edizione del trofeo	R
Visualizzazione di tutte le gare non ancora svolte in trofeo	F
Conteggio di tutte le gare non ancora svolte in trofeo	MF
Visualizzazione di tutti i punteggi degli iscritti alle gare di un Trofeo per il calcolo della relativa classifica	MF

Operazioni relative alle gare

Operazione	**Freq.**
Visualizzazione di tutti i dettagli di una gara	R
Visualizzazione di tutti gli iscritti a una gara divisi per gruppi	MF
Conteggio di tutti gli iscritti a una gara divisi per gruppi	MF
Visualizzazione dell'ordine d'arrivo di una corsa	MF
Visualizzazione dei dettagli di una corsa	MF
Visualizzazione dei vincitori degli abbuoni di una corsa	MF
Visualizzazione dei dettagli del verbale di una gara	F
Visualizzazione dei reclami di una gara	F
Conteggio dei reclami di una gara	F
Visualizzazione degli infortuni di una gara	F
Conteggio degli infortuni di una gara	F
Visualizzazione dei giudici presenti in una gara e il rispettivo ruolo	F

Capitolo 3

Progettazione Concettuale

Definiti i requisiti per la realizzazione della base di dati, si passa alla *progettazione concettuale*. Lo scopo di questa fase è quello di dare una descrizione formale e completa, anche se ad alto livello di astrazione, di ciò che si vuole rappresentare. Il prodotto di questa fase viene chiamato *modello concettuale* e permette di fornire descrizioni ad alto livello indipendenti dall'implementazione e dal DBMS scelto. In questo caso, il modello concettuale adottato è lo schema Entità-Associazione (ER) che mette a disposizione alcuni *costrutti*, quali entità e associazioni, con i quali si è descritto la realtà d'interesse.

3.1 Modello ER Generalizzato (Entità - Associazione)

L'analisi è stata affrontata definendo le entità e le associazioni che lo realizzano. Nella lettura di tale modello si ricorda che è stata richiesta una base di dati a valenza annuale, quindi senza la necessità di tenere uno storico delle informazioni. Per miglior comprensione e chiarezza, lo schema ER complessivo della base di dati è stato diviso in due parti: una parte relativa alla gestione dei tesseramenti, un'altra relativa alla gestione delle gare.

3.1.1 Gestione dei tesseramenti

Di seguito verranno descritte le entità e le associazioni che vengono coinvolte durante il tesseramento annuale. Lo schema ER è riportato in figura 3.1.

Entità

Persona Rappresenta la generica persona che viene schedata nel database.
Attributi: Codice fiscale CF, Nome, Cognome, Sesso, Luogo di nascita, Data di nascita, Luogo residenza, via, numero, CAP, Telefono, Email
Identificatore: CF

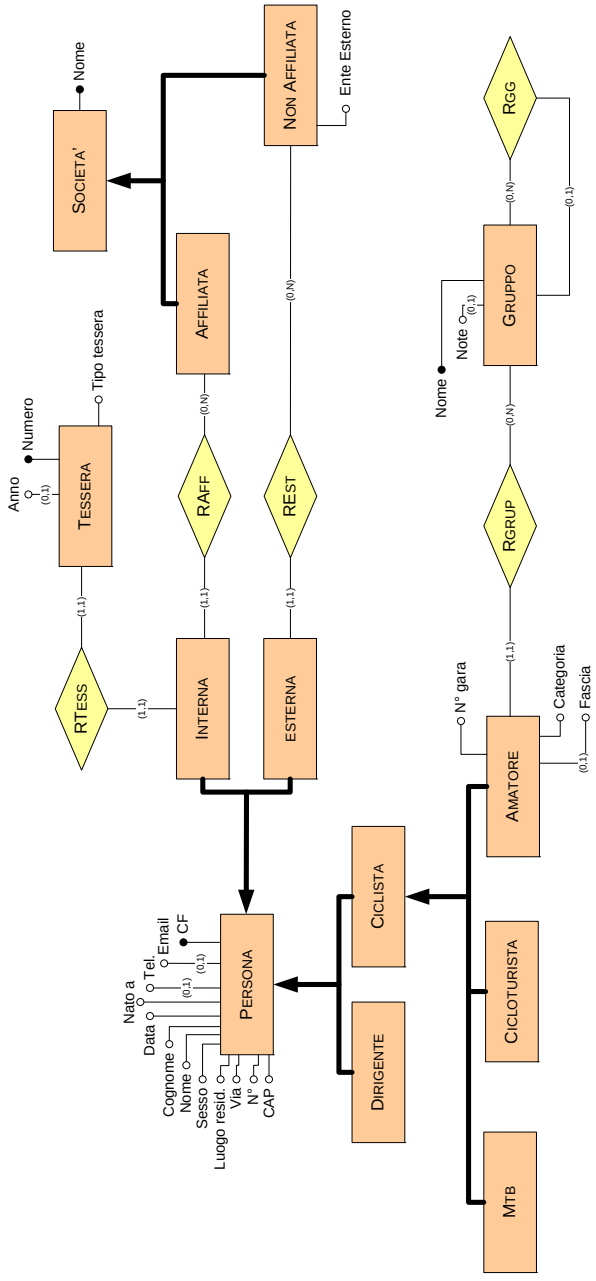


Figura 3.1: Schema ER generalizzato relativo al tesseramento

Ciclista E' il sottoinsieme delle persone che praticano attività sportiva ciclistica in modo occasionale o amatoriale.
Attributi: *Nessuno*
Identificatore: *Nessuno*

Dirigente E' il sottoinsieme delle persone che praticano attività di servizio e coordinamento, come ad esempio i giudici di gara.
Attributi: *Nessuno*

Identificatore: *Nessuno*

MTB (Mountain Bike) E' il sottoinsieme dei ciclisti che praticano l'attività ciclistica fuori dalle strade asfaltate e comunque in maniera occasionale. Non sono previste gare in cui possano partecipare.
Attributi: *Nessuno*
Identificatore: *Nessuno*

Cicloturista E' il sottoinsieme dei ciclisti che praticano l'attività ciclistica su strada in modo occasionale. Non possono partecipare alle gare.
Attributi: *Nessuno*
Identificatore: *Nessuno*

Amatore E' il sottoinsieme dei ciclisti che praticano l'attività ciclistica amatoriale su strada e possono partecipare alle gare.
Attributi: Numero_gara, Categoria, Fascia
Identificatore: *Esterno* (Persona)

Interna E' il sottoinsieme delle persone che sono tesserate con l'ente sportivo.
Attributi: *Nessuno*
Identificatore: *Nessuno*

Esterna E' il sottoinsieme delle persone che non sono tesserate con l'ente sportivo in questione, ma hanno una tessera valida con un altro ente.
Attributi: *Nessuno*
Identificatore: *Nessuno*

Tessera Rappresenta la tessera annuale di iscrizione all'ente.
Attributi: Numero, Anno, Tipo_tessera
Identificatore: Numero

Società Rappresenta una società sportiva ciclistica. Può essere o non essere affiliata all'ente ciclistico.
Attributi: Nome
Identificatore: Nome

Affiliata E' il sottoinsieme delle società sportive che sono affiliate all'ente ciclistico.
Attributi: *Nessuno*
Identificatore: *Nessuno*

Non affiliata E' il sottoinsieme delle società sportive che non sono affiliate all'ente ciclistico.
Attributi: *Nessuno*
Identificatore: *Nessuno*

Gruppo Rappresenta il gruppo di appartenenza di un amatore.
Attributi: Nome, Note
Identificatore: Numero

Associazioni

RTess Le entità coinvolte sono INTERNA (1,1), TESSERA (1,1) e AFFILIATA (0,N)
Attributi: *nessuno*
Questa associazione descrive il fatto che una persona tesserata fa parte di una società sportiva affiliata e possiede una tessera identificativa. Le cardinalità stanno ad indicare che una persona interna all'ente deve essere associata a una tessera, che una tessera viene associata ad una sola persona e che una società affiliata può non avere tesserati.

REst Le entità coinvolte sono ESTERNA (1,1) e NON AFFILIATA (0,N)
Attributi: Ente_Esterno
Questa associazione descrive il fatto che una persona esterna all'ente deve allora essere tesserata con un altro ente sportivo e far parte quindi di una società sportiva non affiliata. Le cardinalità stanno ad indicare che una persona può essere tesserata con al massimo un ente esterno e far parte di una società e che una società non affiliata può non avere iscritti.

RGrup Le entità coinvolte sono AMATORE (1,1) e GRUPPO (0,N)
Attributi: *nessuno*
Questa associazione esprime il fatto che ogni ciclista amatore fa parte di un gruppo. Le cardinalità stanno ad indicare che ogni amatore fa parte di uno ed un solo gruppo e che possono esserci gruppi formati da molti amatori o da nessuno.

RGG Le entità coinvolte sono GRUPPO (0,1) e GRUPPO (0,N)
Attributi: *nessuno*
Questa associazione esprime la relazione che esiste all'interno dei gruppi. Un gruppo infatti può avere dei sottogruppi. Le cardinalità indicano che ogni gruppo può far riferimento al massimo ad un solo altro gruppo, ma che a esso possono riferirsi più gruppi.

Scelte progettuali

L'entità cardine di questa prima parte di schema concettuale è PERSONA. Tale entità racchiude al suo interno diversi sotto-gruppi di persone, che in generale

si differenziano per alcune caratteristiche e attività svolte. Dopo una analisi che raccoglieva più fattori, è stato scelto di specializzare PERSONA in due modi diversi:

- La prima prende in considerazione l'appartenenza o meno di una persona all'ente ciclistico, specializzandola quindi in INTERNA ed ESTERNA. Tale generalizzazione è totale e disgiunta.
- La seconda considera il ruolo di tale persona, ovvero DIRIGENTE e/o CICLISTA. Tale generalizzazione quindi è totale e sovrapposta.
  A sua volta CICLISTA, è stata ulteriormente specializzata in MTB, CICLOTURISTA e *Amatore*, per distinguere l'attività ciclistica svolta. Anche questa generalizzazione è totale e sovrapposta.

3.1.2 Parte relativa alla gestione delle gare

Di seguito verranno descritte le entità e le associazioni che vengono coinvolte durante la gestione delle gare. Lo schema ER è riportato in figura 3.2.

Entità

Gara Rappresenta la singola competizione di un trofeo.
Attributi: Id, Data, Luogo, Denominazione
Identificatore: Id

Traguardo Volante Rappresenta un traguardo intermedio della gara dove vengono assegnati degli abbuoni ai ciclisti nelle prime posizioni.
Attributi: Numero
Identificatore: Numero

Trofeo Rappresenta un trofeo o un campionato presente nell'anno ciclistico in corso.
Attributi: Nome, Anno, Edizione, Accesso
Identificatore: Nome, Anno

Verbale Rappresenta il verbale ufficiale redatto a fine gara. Contiene tutte le specifiche della gara.
Attributi: Testo, Medico responsabile, Numero veicoli a seguito della gara, Provvedimenti disciplinari
Identificatore: *Esterno* (GARA)

Associazioni

Iscrizione Le entità coinvolte sono GARA (0,N), AMATORE (O,N), GRUPPO (0,N)
Attributi: Posizione_arrivo, Punti

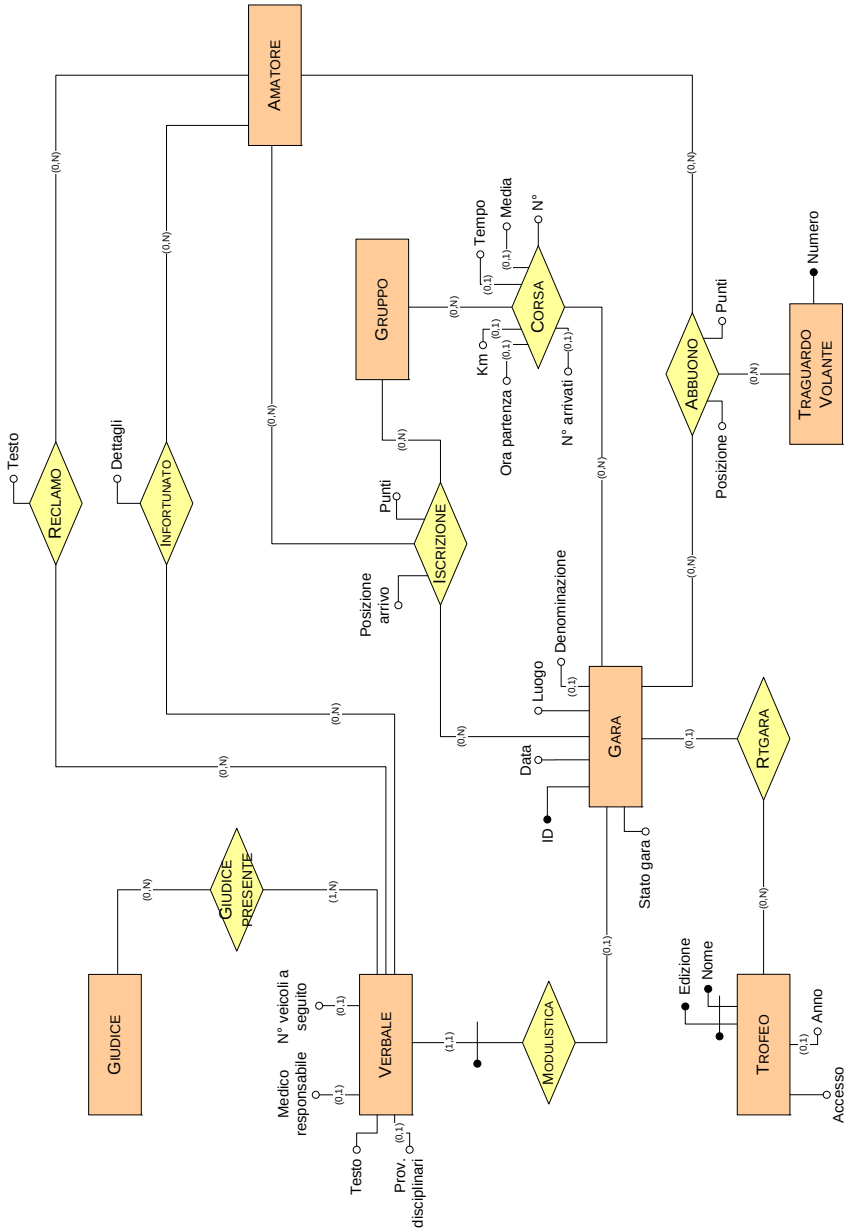


Figura 3.2: Schema ER relativo alla gestione della gara

Questa associazione esprime il fatto che un amatore può iscriversi a una o più gare. Le cardinalità indicano che a una gara possono partecipare più amatori (o nessuno nel caso la gara debba ancora essere svolta) e che un amatore può partecipare a nessuna o a più di una gara con il suo gruppo. Gli attributi stanno ad indicare rispettivamente la posizione di arrivo e punti assegnati ad un amatore partecipante a fine gara.

Corsa Le entità coinvolte sono GARA (0,N) e GRUPPO (0,N)
Attributi: Km, Ora_partenza, Tempo, Media, Numero, Numero partecipanti arrivati

Questa associazione esprime il fatto che ogni gruppo partecipa, in una gara, ad una corsa. Le cardinalità indicano che in una gara possono esserci corse di più gruppi e che un gruppo può partecipare a corse di più gare. Gli attributi sopra indicati indica le specifiche della corsa.

Abbuono Le entità coinvolte sono GARA (0,N), AMATORE (0,N) e TRAGUARDO VOLANTE (0,N)
Attributi: Posizione, Punti
Questa associazione esprime che in una gara possono venire assegnati più abbuoni, relativi a uno o più traguardi volanti, a uno o più amatori in gara. Le cardinalità indicano che possono esserci più traguardi volanti per ogni gara e che ciascuno di questi può dare un abbuono a più di un amatore.

RTgara Le entità coinvolte sono TROFEO (1,N) e GARA (1,1)
Attributi: Numero_gara
Questa associazione esprime il fatto che una gara fa parte di un trofeo. Le cardinalità indicano che a un trofeo devono essere associate una o più gare e che una gara deve essere associata ad un trofeo.

Modulistica Le entità coinvolte sono GARA (0,1) e VERBALE (1,1)
Attributi: *Nessuno*
Questa associazione esprime il fatto che a ogni gara è associato un verbale (a meno che non sia ancora stata disputata). Le cardinalità indicano che una gara ha al massimo un verbale e un verbale deve essere associato a una e una sola gara (non può essere altrimenti dato che *Verbale* è debole rispetto a *Gara*).

Reclamo Le entità coinvolte sono VERBALE (0,N) e AMATORE (0,N)
Attributi: Testo
Questa associazione esprime il fatto che in verbale possono essere presenti dei reclami da parte di amatori in gara. Le cardinalità indicano che un verbale può avere più reclami associati rispettivamente a più amatori e che ogni amatore può comparire nei reclami di più verbali.

Infortunato Le entità coinvolte sono VERBALE (0,N) e AMATORE (0,N)
Attributi: Testo
Questa associazione esprime il fatto che in un verbale possono essere registrati più amatori infortunati. Le cardinalità indicano che un verbale può registrare più infortuni associati rispettivamente a più amatori e ogni amatore può comparire fra gli infortunati di più verbali.

Giudice Presente Le entità coinvolte sono VERBALE (0,N) e PERSONA (0,N)
Attributi: *Nessuno*
Questa associazione esprime il fatto che nel verbale vengono registrati i giudici presenti alla gara. Le cardinalità indicano che in un verbale viene registrata la

presenza di almeno una persona in qualità di giudice di gara e che una persona può comparire come giudice in più verbali.

Scelte progettuali e regole di vincolo

In questa seconda parte di schema concettuale, relativo alle gare e ai trofei, si può notare che i concetti di classifica sono assenti. Per non inserire ridondanza nei dati, si è deciso di non memorizzare le informazioni relative alle classifiche in costrutti specifici. Le informazioni presenti nell'entità GARA e nelle sue associazioni sono già sufficienti al loro calcolo. Questo può essere fatto utilizzando specifiche operazioni di interrogazione a livello di base di dati o con metodi dedicati a livello applicativo. E' necessario introdurre alcuni vincoli inter-relazionali per garantire l'integrità dei dati. Questi sono esplicitati a parole qui di seguito, dato che non è stato possibile esprimerli tramite i formalismi grafici adottati:

- Una gara svolta deve necessariamente essere associata ad un verbale.
- Relativamente ad una GARA di un TROFEO ad *accesso* chiuso (ovvero riservato ai soli tesserati), tutte le occorrenze di AMATORE, che partecipano all'associazione ISCRIZIONE, devono essere associate ad una PERSONA tesserata.
- Relativamente al VERBALE di una GARA, tutte le occorrenze di AMATORE che partecipano alle associazioni ABBUONO, INFORTUNATO e RECLAMO devono partecipare anche alla associazione ISCRIZIONE (ovvero gli amatori devono essere iscritti alla gara).

Capitolo 4

Progettazione Logica

La progettazione logica consiste nella traduzione dello schema concettuale nel modello dei dati del DBMS. Ha come obbiettivo quello di costruire uno schema logico in grado di descrivere in maniera corretta ed efficiente le informazioni contenute nel modello concettuale. Come si vedrà, in questa fase si considerano anche aspetti legati all'integrità, alla consistenza dei dati (ovvero ai vincoli presenti su di essi) e all'efficienza.
Nella progettazione logica si possono distinguere due fasi:

- **Ristrutturazione dello schema ER**
  Lo schema concettuale viene riorganizzato e semplificato. I costrutti principali, come le entità e le associazioni, sono direttamente traducibili mentre altri, come ad esempio le generalizzazioni e gli attributi multi-valore, non hanno traduzione. Occorre, quindi, eliminare tali costrutti rappresentandoli mediante altre entità e associazioni.

- **Traduzione verso il modello logico**
  In questa fase si procede alla creazione di uno schema logico *equivalente*, che abbia cioè lo stesso carico informativo del modello concettuale ristrutturato. Le possibili traduzioni devono tenere in considerazione molteplici fattori, come ad esempio la mole di dati prevista e le operazioni di interrogazione richieste.

4.1 Schema ER ristrutturato

Nella prima fase, la ristrutturazione del modello concettuale presenta più alternative, ognuna delle quali prende in considerazione diversi fattori come ad esempio la struttura e le caratteristiche delle entità e delle associazioni coinvolte. E' importante porre attenzione anche alle prestazioni che la base di dati avrà una volta realizzata a livello fisico. Anche se non valutabili con precisione a questo livello di progettazione, due caratteristiche importanti da analizzare sono:

- L'occupazione dei dati, ovvero quanto spazio di memoria è necessario a memorizzare i dati descritti dallo schema.

- Il costo delle operazioni, valutato in termini del numero (approssimativo) di occorrenze di entità e associazioni che devono essere visitate per rispondere a una determinata operazione eseguita sulla base di dati.

In questo caso di studio, sono di particolare interesse due alternative di ristrutturazione:

1. **Sostituzione della generalizzazione con associazioni.**
   La generalizzazione viene trasformata in associazioni uno a uno che legano l'entità genitore con le entità figlie. Non ci sono trasferimenti di attributi o di associazioni e le entità figlie, divenute entità deboli, vengono identificate esternamente dall'entità genitore. Questo metodo è particolarmente conveniente quando la generalizzazione non è totale o quando ci sono forti distinzioni tra le entità figlie e l'entità genitore, come ad esempio la presenza di attributi o associazioni specifiche.
   Questa scelta comporta un risparmio di memoria per l'assenza di valori nulli che inevitabilmente sarebbero stati aggiunti in altri casi. A scapito però, si ha l'introduzione di nuovi vincoli e un incremento degli accessi per mantenere la coerenza dei dati.

2. **Accorpamento delle entità figlie della generalizzazione nell'entità genitore.**
   Le entità figlie vengono eliminate e i loro eventuali attributi e associazioni vengono ereditati dall'entità genitore. In aggiunta, tale entità guadagna un ulteriore attributo (o più di uno) che serve a distinguere il "tipo" di occorrenza, ovvero a quale entità figlia apparteneva tale occorrenza.
   Questo metodo di ristrutturazione è particolarmente conveniente quando non c'è particolare distinzione fra gli attributi delle entità figlie e quelli dell'entità genitore o quando addirittura questi non sono presenti. In queste condizioni, infatti, lo spreco di memoria dovuto alla presenta di valori nulli è limitato, come del resto anche il numero degli accessi, dato che gli attributi non sono distribuiti su varie entità.

Lo schema ER relativo alla gestione dei tesseramenti presenta molteplici generalizzazioni, sviluppate anche su più livelli. Una accurata analisi ha portato ad adottare, per ognuna di queste la strategia di ristrutturazione migliore.
I seguenti punti illustrano con maggior dettaglio i passaggi di ristrutturazione svolti:

1. La ristrutturazione della generalizzazione Affiliata – Non Affiliata di Società. Tale generalizzazione è totale e disgiunta, l'entità Non Affiliata presenta un solo attributo mentre Affiliata nessuno. L'alternativa di ristrutturazione scelta è stata la numero (2). Dopo l'operazione, Società ha ereditato l'attributo *Ente Esterno* (che diventa opzionale), le associazioni RAff e REst e ha guadagnato l'attributo *Affiliata*.

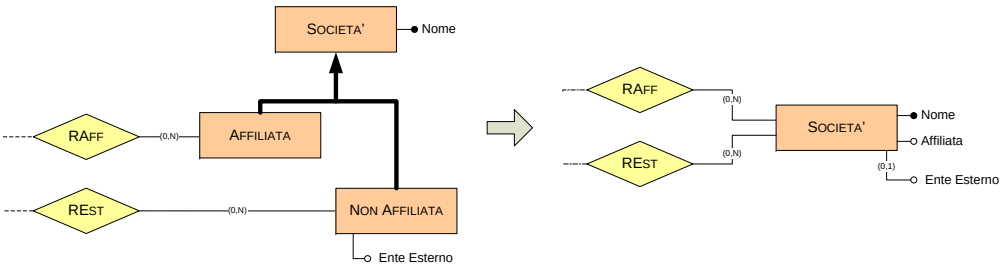


Figura 4.1: Ristrutturazione della generalizzazione AFFILIATA – NON AFFILIATA.

2. La ristrutturazione della generalizzazione INTERNA – ESTERNA di PERSONA. Questa generalizzazione è di natura simile a quella del punto precedente; è totale e disgiunta e non presenta attributi nelle entità figlie. E' stata preferita anche in questo caso l'alternativa (2): PERSONA ha ereditato le tre associazioni RTESS, RAFF e REST e guadagnato l'attributo *Tesserata*. Si è scelto inoltre di accorpare le due associazioni RAFF e REST in una nuova associazione RSOC.

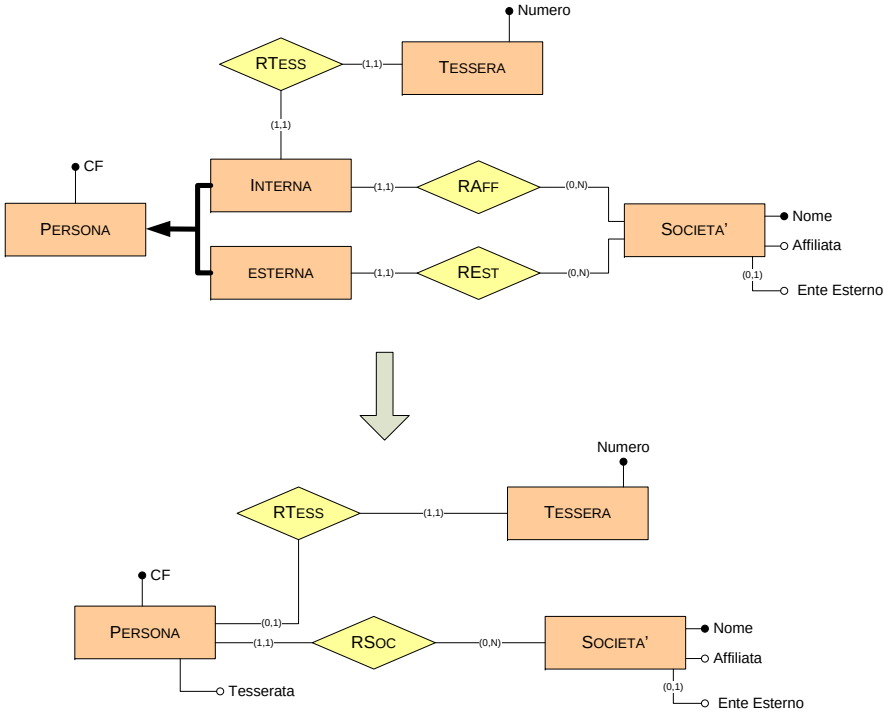


Figura 4.2: Ristrutturazione della generalizzazione INTERNA – ESTERNA.

3. La ristrutturazione della generalizzazione DIRIGENTE – CICLISTA di PERSONA. Come nei casi precedenti, questa generalizzazione è stata ristrutturata adottando l'alternativa (2). Il risultato è che PERSONA eredita l'asso-

ciazione e l'attributo di CICLISTA e, dato che la generalizzazione è totale e sovrapposta, guadagna l'attributo multi-valore *Ruolo*.

4. La ristrutturazione della generalizzazione MTB – CICLOTURISTA – AMATORE di CICLISTA. Questa generalizzazione è stata ristrutturata adottando entrambi i metodi visti in precedenza. L'entità AMATORE si distingue dalle altre per la presenza di molteplici attributi; per questo è stata ristrutturata adottando il metodo (1) e trasformata quindi in entità debole attraverso l'associazione RPA con PERSONA. La parte di generalizzazione contenente le entità MTB e CICLOTURISTA è stata ristrutturata con il metodo (2), accorpando quindi tali entità a PERSONA. A quest'ultima è stato, di conseguenza, aggiunto anche l'attributo multi-valore *Attività*.

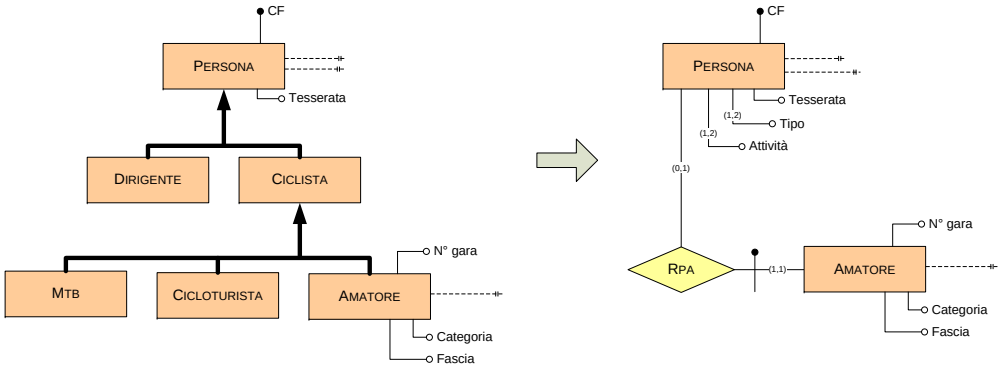


Figura 4.3: Le ristrutturazioni delle generalizzazioni DIRIGENTE – CICLISTA e MTB – CICLOTURISTA – AMATORE.

Nelle figure 4.4 e 4.5 sono riportati gli schemi ER complessivi dove sono state apportate le modifiche dovute alla fase di ristrutturazione appena descritta.
Gli attributi multi-valori aggiunti a PERSONA in questa fase di progettazione devono essere anch'essi ristrutturati prima di passare al modello relazionale. L'attributo *Ruolo* è stato sostituito dagli attributi DIRIGENTE e CICLISTA, mentre l'attributo ATTIVITÀ dagli attributi CICLOTURISTA e MTB. In questo modo sono state conservate tutte le informazioni espresse dai valori che tali attributi potevano assumere.

4.1.1 Regole di vincolo aggiuntive

La fase di ristrutturazione, come si è visto, porta a introdurre ulteriori vincoli inter-relazionali in aggiunta a quelli già descritti nella fase di progettazione concettuale, al fine di preservare l'integrità dei dati:

- Una PERSONA *tesserata* deve necessariamente essere associata ad una TESSERA e ad una SOCIETÀ *affiliata*. Se invece una persona non è tesserata, deve essere associata ad una SOCIETÀ *non affiliata* (alla quale si può specificare l'ente ciclistico di appartenenza) e non deve avere assegnata alcuna TESSERA.

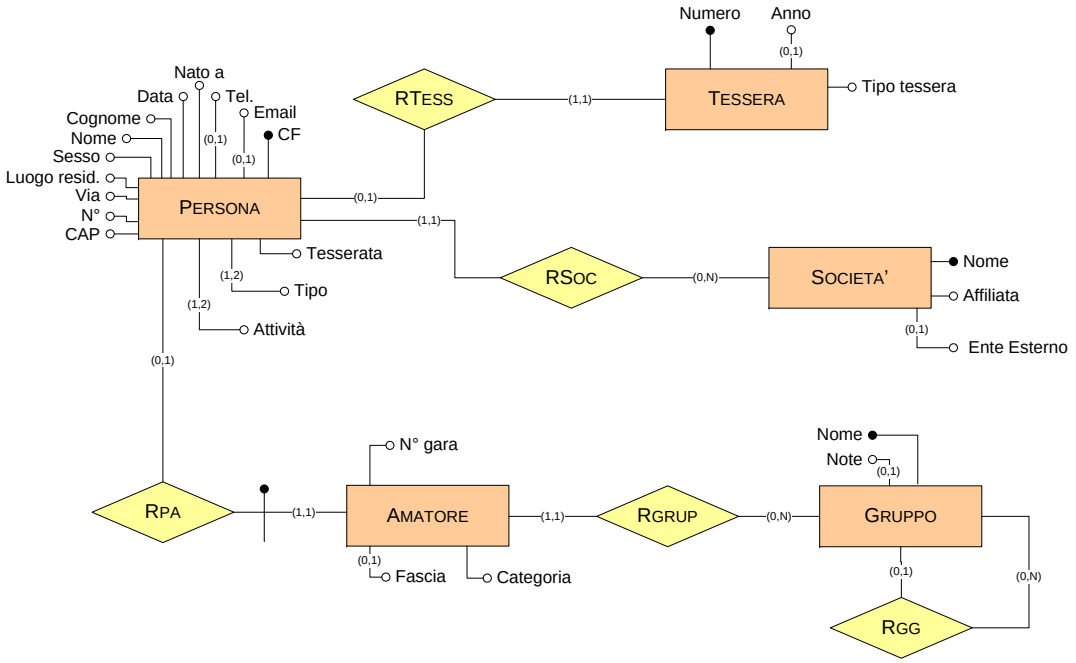


Figura 4.4: Schema ER relativo alla gestione dei tesseramenti dopo le operazioni di ristrutturazione.

- In un VERBALE, ogni occorrenza di PERSONA che partecipa all'associazione GIUDICE PRESENTE deve essere "Dirigente".

Vincoli inter-relazionali complessi, come quelli appena descritti, sono di difficile implementazione durante la fase di progettazione fisica, dove gli strumenti messi a disposizione dal DBMS scelto posso rivelarsi insufficienti e inadeguati. In questo caso, il compito di implementare tali vincoli sarà affidato a un livello applicativo più alto.

4.2 Modello Relazionale

Terminata la prima fase di progettazione, si passa alla traduzione[1] del modello concettuale prodotto.
Il modello logico adottato è il modello relazionale, che rappresenta la base di dati come una collezione di *relazioni*. Ogni relazione viene vista come una tabella ed in generale rappresenta una entità o associazione del modello concettuale. In ognuna di queste, ogni colonna corrisponde ad un *attributo* e ogni riga, chiamata *tupla*, a una collezione di dati collegati.

[1] In riferimento alle regole di traduzione adottate si veda *Basi di dati, Modelli e linguaggi di programmazione (P. Atzeni, S. Cerri, S. Paraboschi, R. Torlone), cap. 9*

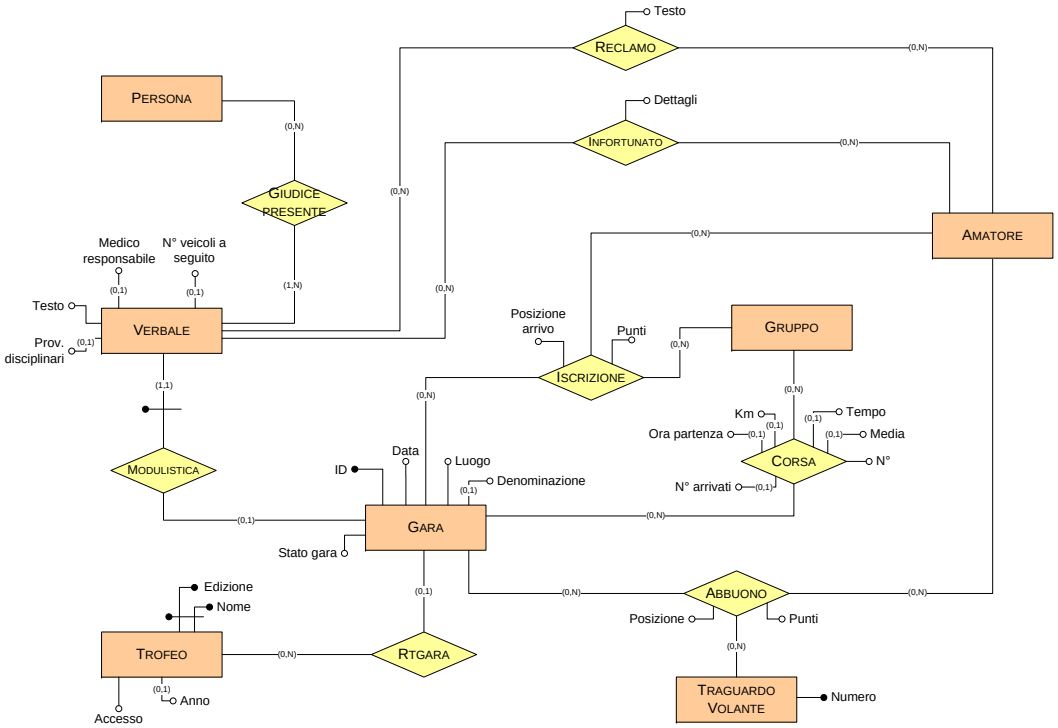


Figura 4.5: Schema ER relativo alla gestione della gara dopo le operazioni di ristrutturazione

Schema Ralazionale complessivo

Lo schema relazionale riportato nelle figure 4.6 e 4.7 è il prodotto dalla traduzione dello schema ER. Con il formalismo grafico adottato è possibile rappresentare, oltre alle relazioni e ai relativi attributi, anche i vincoli di integrità referenziale esistenti tra le stesse. Tale sistema permette di tenere traccia delle associazioni originali dello schema ER, individuando e percorrendo i *cammini di join*. Per una migliore comprensione sono state adottate le seguenti convenzioni:

NOME (**Attributo1**, **Attributo2**, Attributo3, Attributo4 ...) dove:

- **NOME** indica il nome della tabella;
- **Attributo1**: indica la chiave primaria (se più attributi sono sottolineati c'è una chiave composta);
- **Attributo2**: indica un attributo obbligatorio;
- Attributo3: indica un attributo non obbligatorio;
- Attributo4 : indica un vincolo inter-referenziale.

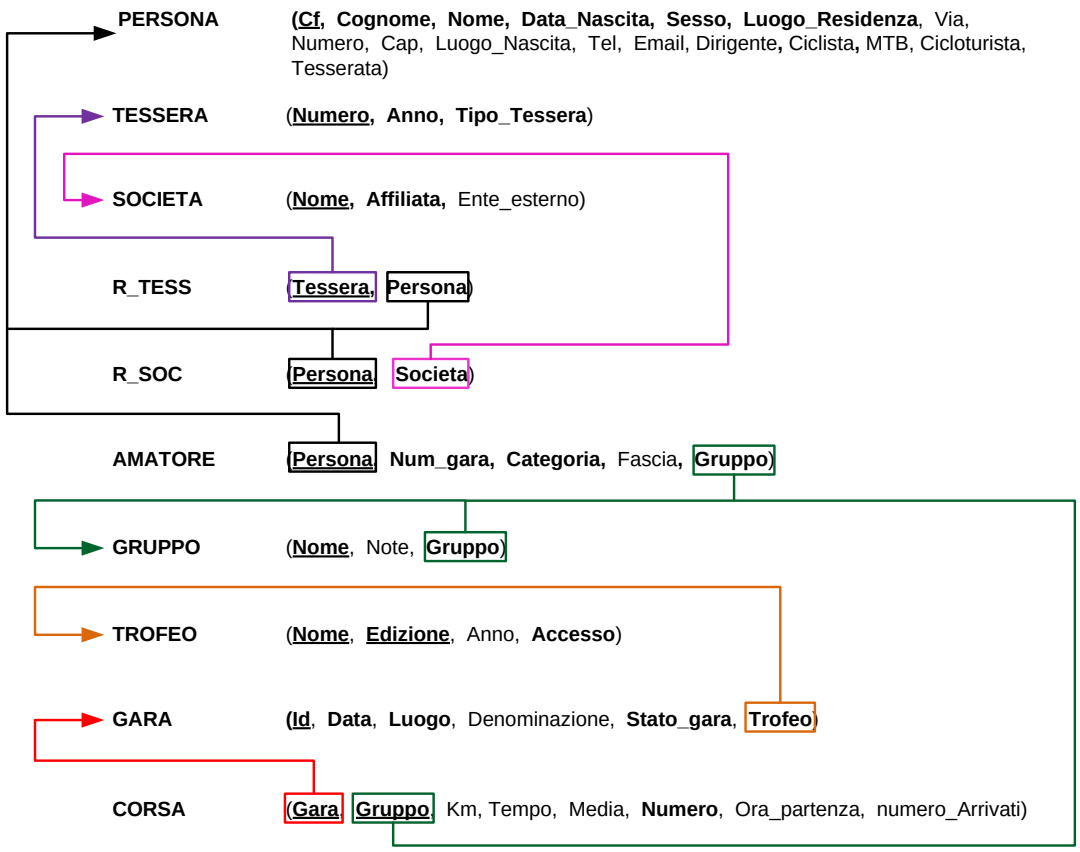


Figura 4.6: Prima parte dello Schema Relazionale

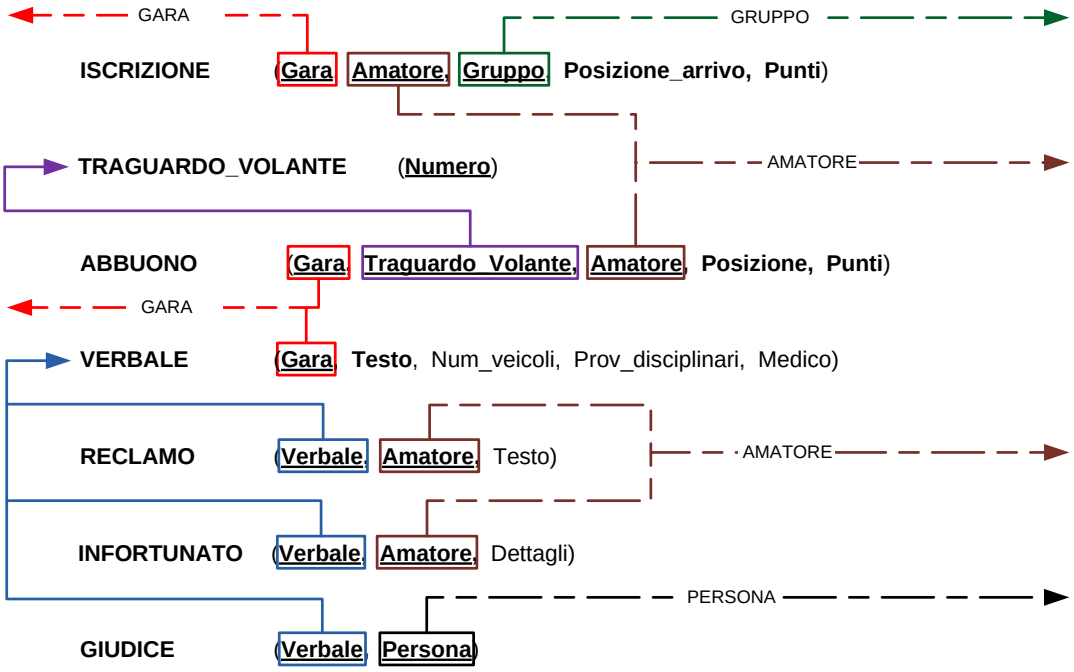


Figura 4.7: Seconda parte dello Schema Relazionale

Capitolo 5

Progettazione Fisica

L'ultima fase di progettazione della base di dati è quella fisica. A partire dallo schema logico e dalle caratteristiche richieste si produce lo schema fisico in SQL costituito dalle effettive definizioni delle relazioni e dai relativi parametri.
L'implementazione di ogni relazione dello schema logico è riassumibile con i seguenti passi:

1. Definizione di una tabella avente il nome della relazione che la rappresenta
2. Elencazione dei rispettivi attributi assegnando a ciascuno il rispettivo nome e un dominio idoneo
3. Specificazione della chiave primaria (eventualmente più di una)
4. Definizione dei vincoli intra-relazionali (ad esempio vincoli `NOT NULL` clausole `CHECK` o valori `DEFAULT`)
5. Definizione dei vincoli inter-relazionali, ovvero i vincoli di integrità referenziale `FOREIGN KEY`

Il codice SQL prodotto da questa fase non verrà direttamente implementato nel DBMS, ma servirà come linea guida nella definizione delle mappature delle classi persistenti durante la fase di programmazione Java.

5.1 Codice SQL

Viene riportato di seguito il codice SQL prodotto in questa fase di progettazione.

```
1 CREATE TYPE TIPO_CATEGORIA AS ENUM ('A', 'B', 'C', 'D', 'E', '
    SE', 'MSE');
  CREATE TYPE TIPO_FASCIA AS ENUM ('A', 'B');
3 CREATE TYPE TIPO_TESSERA AS ENUM ('A1-Ciclisti', 'B1-Ciclisti')
    ;
  CREATE TYPE TIPO_ACCESSO AS ENUM ('Libero', 'Chiuso');
5
  CREATE SCHEMA GestioneGara
```

```
7
CREATE TABLE Persona (
9     cf VARCHAR(16) PRIMARY KEY,
      cognome VARCHAR(250) NOT NULL,
11    nome VARCHAR(250) NOT NULL,
      sesso CHAR(1) NOT NULL,
13    data_Nascita DATE NOT NULL,
      luogo_nascita VARCHAR(250),
15    luogo_residenza VARCHAR(250),
      via VARCHAR(250),
17    numero VARCHAR(6),
      cap CHAR(5),
19    telefono CHAR(11),
      email VARCHAR(255),
21    dirigente BOOLEAN NOT NULL,
      ciclista BOOLEAN NOT NULL,
23    cicloturista BOOLEAN NOT NULL,
      mtb BOOLEAN NOT NULL,
25    tesserata BOOLEAN NOT NULL
      )
27
CREATE TABLE Tessera (
29    numero VARCHAR(20) PRIMARY KEY,
      anno SMALLINT,
31    tipo_tessera TIPO_TESSERA NOT NULL
      )
33
CREATE TABLE Societa (
35    nome VARCHAR(50) PRIMARY KEY,
      affiliata BOOLEAN NOT NULL,
37    ente_esterno VARCHAR(250)
      )
39
CREATE TABLE R_Tess (
41    fk_tessera VARCHAR(20) PRIMARY KEY
        REFERENCES Tessera(numero) ON DELETE CASCADE ON UPDATE
          CASCADE,
43    fk_persona VARCHAR(16) NOT NULL UNIQUE
        REFERENCES Persona(cf) ON DELETE CASCADE ON UPDATE
          CASCADE
45    )

47  CREATE TABLE R_Soc (
      fk_persona VARCHAR(16) PRIMARY KEY
49      REFERENCES Persona(cf) ON DELETE CASCADE ON UPDATE
          CASCADE,
      fk_societa VARCHAR(50) NOT NULL
```

```
51          REFERENCES Societa(nome) ON DELETE CASCADE ON UPDATE
                CASCADE
        )
53
    CREATE TABLE Gruppo (
55      nome VARCHAR(2) PRIMARY KEY,
        note VARCHAR(250),
57      fk_gruppo VARCHAR(2)
            REFERENCES Gruppo(nome) ON DELETE CASCADE ON UPDATE
                CASCADE
59      )

61  CREATE TABLE Amatore (
        fk_persona VARCHAR(16)
63          REFERENCES Persona(cf) ON DELETE CASCADE ON UPDATE
                CASCADE,
        numero_gara VARCHAR(4) UNIQUE NOT NULL,
65      categoria TIPO_CATEGORIA NOT NULL,
        fascia TIPO_FASCIA,
67      fk_gruppo VARCHAR(2)
            REFERENCES Gruppo(nome) ON DELETE CASCADE ON UPDATE
                CASCADE,
69      PRIMARY KEY (fk_persona)
        )
71
    CREATE TABLE Trofeo (
73      edizione SMALLINT,
        nome VARCHAR(250),
75      accesso TIPO_ACCESSO,
        anno SMALLINT,
77      PRIMARY KEY (edizione, nome)
        )
79
    CREATE TABLE Gara (
81      id SERIAL4 PRIMARY KEY,
        data DATE NOT NULL,
83      luogo VARCHAR(250) NOT NULL,
        denominazione VARCHAR(250),
85      stato_gara BOOLEAN NOT NULL,
        fk_edizione_trofeo SMALLINT NOT NULL,
87      fk_nome_trofeo VARCHAR(250) NOT NULL,
        FOREIGN KEY (fk_edizione_trofeo, fk_nome_trofeo)
89          REFERENCES Trofeo(edizione,nome) ON DELETE CASCADE ON
                UPDATE CASCADE
        )
91
    CREATE TABLE Corsa (
93      fk_gara INTEGER
```

```
        REFERENCES Gara(id) ON DELETE CASCADE ON UPDATE CASCADE,
95      fk_gruppo VARCHAR(2)
        REFERENCES Gruppo(nome) ON DELETE CASCADE ON UPDATE
            CASCADE,
97      km SMALLINT,
        tempo TIME,
99      media REAL,
        numero SMALLINT NOT NULL,
101     ora_partenza TIME,
        numero_arrivati SMALLINT,
103     PRIMARY KEY (fk_gara, fk_gruppo)
        )
105
    CREATE TABLE Iscrizione (
107     fk_gara INTEGER
        REFERENCES Gara(id) ON DELETE CASCADE ON UPDATE CASCADE,
109     fk_persona_amatore VARCHAR(16)
        REFERENCES Amatore(fk_persona) ON DELETE CASCADE ON
            UPDATE CASCADE,
111     fk_gruppo VARCHAR(2)
        REFERENCES Gruppo(nome) ON DELETE CASCADE ON UPDATE
            CASCADE,
113     posizione_arrivo SMALLINT DEFAULT 0,
        punti SMALLINT DEFAULT 0,
115     PRIMARY KEY (fk_gara, fk_persona_amatore, fk_gruppo)
        )
117
    CREATE TABLE Traguardo_volante (
119     numero SMALLINT PRIMARY KEY
        )
121
    CREATE TABLE Abbuono (
123     fk_gara INTEGER
        REFERENCES Gara(id) ON DELETE CASCADE ON UPDATE CASCADE,
125     fk_traguardo_volante SMALLINT
        REFERENCES Traguardo_volante(numero) ON DELETE CASCADE ON
            UPDATE CASCADE,
127     fk_persona_amatore VARCHAR(16)
        REFERENCES Amatore(fk_persona) ON DELETE CASCADE ON
            UPDATE CASCADE,
129     posizione SMALLINT,
        punti SMALLINT,
131     PRIMARY KEY (fk_gara, fk_traguardo_volante,
            fk_persona_amatore)
        )
133
    CREATE TABLE Verbale (
135     fk_gara INTEGER PRIMARY KEY
```

```
        REFERENCES Gara(id) ON DELETE CASCADE ON UPDATE CASCADE,
137     testo VARCHAR(500) NOT NULL,
        numero_veicoli SMALLINT,
139     prov_disciplinari VARCHAR(500),
        medico VARCHAR(250)
141     )

143 CREATE TABLE Reclamo (
        fk_verbale INTEGER
145         REFERENCES Verbale(fk_gara) ON DELETE CASCADE ON UPDATE
                CASCADE,
        fk_persona_amatore VARCHAR(16),
147     testo VARCHAR(500) NOT NULL,
        FOREIGN KEY (fk_persona_amatore)
149         REFERENCES Amatore(fk_persona) ON DELETE CASCADE ON
                UPDATE CASCADE,
        PRIMARY KEY (fk_verbale, fk_persona_amatore)
151     )

153 CREATE TABLE Infortunio (
        fk_verbale INTEGER
155         REFERENCES Verbale(fk_gara) ON DELETE CASCADE ON UPDATE
                CASCADE,
        fk_persona_amatore VARCHAR(16)
157         REFERENCES Amatore(fk_persona) ON DELETE CASCADE ON
                UPDATE CASCADE,
        dettagli VARCHAR(500) NOT NULL,
159     PRIMARY KEY (fk_verbale, fk_persona_amatore)
        )
161
    CREATE TABLE Giudice (
163     fk_verbale INTEGER
            REFERENCES Verbale(fk_gara) ON DELETE CASCADE ON UPDATE
                CASCADE,
165     fk_persona VARCHAR(16)
            REFERENCES Persona(cf) ON DELETE CASCADE ON UPDATE
                CASCADE,
167     PRIMARY KEY (fk_verbale, fk_persona)
        )
```

Capitolo 6

Architettura Hibernate

In Java, l'accesso ad un database relazionale viene normalmente effettuato tramite le API JDBC (Java Database Connectivity). L'SQL può essere scritto a mano direttamente nel codice Java e le operazioni possibili sono di basso livello, come l'esecuzione di una query, la valorizzazione dei suoi parametri e la possibilità di scorrere tra i risultati dell'interrogazione. Nelle applicazioni aziendali, spesso l'interesse maggiore si concentra sul problema della modellazione dei dati e sulla parte di *business logic* rispetto a quella implementativa. In questi ambiti, quindi, il codice di accesso ai dati risulta spesso un'operazione meccanica, prona ad errori e di difficile manutenibilità.

6.1 L'Object Relational Mapping

L'*Object Relational Mapping* si pone come una tecnica alternativa per la realizzazione del layer di persistenza degli oggetti. Rispetto alle tecniche *hand-coded* come JDBC, realizza la persistenza automatica (e trasparente) di oggetti di applicazioni Java in tabelle di un DB relazionale, basandosi su meta-dati in stile XML. In pratica, lavora trasformando (in modo reversibile) i dati da una rappresentazione all'altra.

Un framework ORM consiste principalmente di quattro componenti:

- Una API per eseguire le operazioni CRUD sugli oggetti delle classi del modello;
- Un linguaggio o una API per specificare query che fanno riferimento alle classi e alle loro proprietà;
- Uno strumento per la specifica del mapping mediante metadati;
- Una tecnica per interagire con oggetti transazionali al fine di eseguire funzioni di ottimizzazione come *lazy load fetching* e *dirty checking*.

Un software ORM si posiziona a livello *middleware*, cioè a un livello intermedio fra l'applicazione e la base di dati. Automatizzando la maggior parte del lavoro legato alla persistenza, permette di concentrare l'attenzione sui problemi di *businnes logic* dell'applicazione, enfatizzando tale aspetto rispetto a quello implementativo. Inevitabilmente, introduce alcune penalizzazioni in termini di performance e di complessità, che però devono essere valutate considerando sia le politiche di ottimizzazione fornite, sia gli aspetti realizzativi quali tempo e budget.
In generale, un software ORM offre i seguenti vantaggi:

- Isola la logica di persistenza dei dati all'interno di in un solo livello favorendo la modularità complessiva dell'applicazione;
- Aumenta il livello di astrazione a favore della semplicità di utilizzo e della riduzione della quantità di codice, mascherando le classiche operazioni CRUD che occupano molto tempo di stesura del codice e di testing;
- Riduce il tempo di sviluppo, limitato alla definizione e alla gestione dei meta-dati che descrivono la mappatura;
- Essendo implementato come *middleware* offre diverse opportunità di ottimizzazione;
- Offre un'elevata portabilità rispetto alla tecnologia DBMS utilizzata: basta apportare alcune modifiche al file di configurazione del sistema ORM utilizzato per passare da un DBMS all'altro.

6.2 Hibernate

Esistono molti strumenti ORM che hanno raggiunto un grande livello di qualità e affidabilità. Uno di questi è certamente **Hibernate**, che rappresenta lo standard risolutivo per il mapping object/relational automatico. Hibernate non solo implementa la tecnica ORM, ma mette anche a disposizione altre funzionalità, creando un insieme di soluzioni per la persistenza in Java. È un software open source distribuito su licenza GNU, sviluppato dal 2001 e arrivato ora alla versione 4.1 con lo scopo principale di offrire migliori capacità nella gestione della persistenza semplificando tutte le complessità. Per questo è uno strumento *full ORM* (tutto il codice SQL viene automaticamente generato dalla descrizione basata sui metadati) e supporta *object modeling* sofisticati. Interagisce, quindi, con le API ORM e le classi del modello di dominio "appoggiandosi" e sfruttando le tecnologie SQL e JDBC per astrarre il livello SQL/JDBC sottostante.

Conosciuto per la sua stabilità e qualità, Hibernate è stato progettato per lavorare anche in applicazioni basate su cluster di Server e per fornire un'architettura altamente ottimizzabile e estendibile. Consente, infatti, di sviluppare classi persistenti senza stravolgere il paradigma Object-Oriented, permettendo l'uso di ereditarietà, polimorfismo e strutture dati; inoltre, non richiede l'implementazione

di nessuna interfaccia o classe. Offre molteplici tecniche di ottimizzazione, come ad esempio *lazy initialization* (l'inizializzazione pigra, cioè la tattica di istanziare un oggetto solo nel momento in cui tale operazione è effettivamente necessaria) e *many fetching* (recuperare più istanze possibili). In generale, si può affermare che Hibernate offre potenzialmente performance migliori o comunque equiparabili a una programmazione basata su JDBC.

Oltre a riconoscere il JPAQL (Java Persistence Query Language) e l'SQL nativo, Hibernate è dotato di un linguaggio di interrogazione molto potente: l'HQL. Ha una sintassi volutamente simile all'SQL ma è pienamente orientato agli oggetti e comprende nozioni come l'ereditarietà, il polimorfismo e l'associazione. Permette quindi di creare query utilizzando esclusivamente le classi oggetto e le loro variabili.

6.2.1 Approcci di sviluppo

Grazie alla sua modularità e flessibilità, Hibernate riesce ad adattarsi il più possibili ai bisogni reali dello sviluppatore. Permette tre approcci tradizionali di programmazione:

- **Top down**
  Si parte da un diagramma delle classi di dominio e dalla sua implementazione in Java e si ha completa libertà rispetto allo schema della base di dati. Si specifica la mappatura e infine si utilizza Hibernate (in particolare lo strumento *hbm2ddl*) per generare lo schema della base di dati. Questo approccio spesso è sconsigliato: utilizzare appieno gli strumenti di un DBMS richiede uno sforzo notevole durante la fase di mappatura, paragonabile all'utilizzo diretto di JDBC.

- **Bottom up**
  Si parte da un base di dati esistente *legacy* e si ha completa libertà rispetto al dominio dell'applicazione. Si usa una serie di strumenti per generare lo schema di base del codice Java del delle classi persistenti e la loro mappatura (come gli strumenti `jdbcconfiguration`, `hbm2hbmxml` e `hbm2cfgxml`). Questo utilizzo, anche se possibile, è atipico: è inevitabile l'impossibilità di sfruttare le potenzialità della programmazione ad oggetti e ottenere un codice elegante ed efficiente.

- **Meet in the middle**
  Si parte da una base di dati legacy pre-esistente (oppure già definita) e da un diagramma delle classi di dominio dell'applicazione e ne si realizza la mappatura. E' il caso più comune e più interessante anche se più complesso, dove si ricorre ad Hibernate per implementare l'approccio generico all'ORM.

L'approccio adottato in questo progetto rientra nell'ultima tipologia presentata, anche se le classi di dominio non erano già definite, ma sono state realizzate

durante la fase di mappatura cercando di seguire una traduzione "naturale" delle tabelle della base di dati nel dominio a oggetti.

6.2.2 Utilizzo

In questo progetto, Hibernate è stato utilizzato in maniera nativa, ovvero facendo ricorso esclusivamente alle interfacce che esso mette a disposizione per realizzare la persistenza.
In alternativa, è possibile utilizzare Hibernate come *Provider* ed implementare l'interfaccia JPA di Java per realizzare la persistenza. In questo caso anziché utilizzare i file XML, le istruzioni di mappatura vengono inserite nel codice sorgente utilizzando annotazioni in stile *Java Annotation*. Esistono inoltre altri modi di utilizzo, soprattutto all'interno di Java EE, i quali però esulano dall'obbiettivo di questa tesi e non sono stati considerati.

Interfacce

Le principali interfacce che Hibernate mette a disposizione per l'accesso alla base di dati sono tre:

- **Session**: ogni sua istanza rappresenta una sessione di comunicazione tra l'applicazione e la base di dati. Comprende i metodi per salvare e caricare gli oggetti;
- **Transaction**: ogni sua istanza rappresenta una unità di lavoro atomica al cui interno vengono effettuate una o più operazioni di tipo CRUD;
- **Query**: ogni sua istanza permette di creare, specificare ed eseguire *query* (sia nel linguaggio HQL che SQL).

6.2.3 Configurazione e startup

Poiché Hibernate è progettato per funzionare in ambienti differenti, esistono un gran numero di librerie e di parametri di configurazione. Per la maggior parte delle applicazioni, le librerie richieste oltre a quelle necessarie al funzionamento sono molto limitate e solitamente è sufficiente inserirle nella cartella di lavoro. Anche i parametri di configurazione sono limitati e vanno specificati all'interno di un file XML.

Librerie necessarie

Le librerie necessarie sono contenute all'interno del pacchetto di installazione, scaricabile direttamente dal sito web del progetto, e vanno inserite all'interno della directory di lavoro, in genere nella cartella `lib`. Per un uso tipico di Hibernate è sufficiente copiare tutti i file .jar contenuti nella cartella `required`.
Infine, è necessario disporre di alcune librerie aggiuntive:

- La libreria **c3p0.jar**
  Anche se Hibernate ha il suo pool di connessioni, è preferibile utilizzarne uno più performante, come ad esempio C3P0. Quando viene istanziato, questo gestore si occupa di aprire una serie di connessioni verso il database. Ogni qualvolta altri oggetti necessitano di accedere al database, richiedono la connessione al gestore che ne fornisce una libera. Quando l'operazione termina, l'oggetto restituisce la connessione al gestore che potrà quindi rimetterla a disposizione di altri oggetti che la richiederanno. In questo modo si evita il degrado delle prestazioni, in quanto le connessioni non vengono ricreate ogni volta, ma solamente in fase di attivazione dell'applicazione.

- La libreria **postgresql.jdbc4.jar**
  Per connettersi alla base di dati, Hibernate si appoggia al driver JDBC del DBMS scelto. Per questo, tale libreria deve essere presente: in questo caso è stata inserita quella di PostgreSQL;

- La libreria **log4j.jar**
  Hibernate utilizza per la gestione dei log degli eventi *log4j*, che è uno dei tool più usati per la gestione dei log su Java. All'interno della cartella sorgente `src` deve essere presente, inoltre, il file di configurazione `log4j.properties`.

Anche queste librerie sono tutte disponibili in modalità open-source nei rispettivi siti web dei progetti.

Configurazione

Una volta reperite le librerie, è necessario fornire ad Hibernate alcuni parametri di configurazione, secondo l'utilizzo prescelto. Occorre in primo luogo creare il file di configurazione `hibernate.cfg.xml` all'interno della cartella `src`. Al suo interno, contenuti nell'elemento `session-factory`, vengono inserite tutte le impostazioni per l'accesso ad una particolare base di dati, come ad esempio:

- I parametri contenenti le informazioni necessarie per impostare la connessione JDBC. Hanno la forma `(hibernate).connection.*` e vanno modificati a seconda dei parametri del Server e del DBMS prescelto.

- I parametri di configurazione del pool di connessioni C3P0, come ad esempio:

  - min size: il numero minimo di connessioni che devono essere pronte in ogni momento;
  - max size: il numero massimo di connessioni aperte gestite dal pool;
  - timeout: il tempo al termine del quale una connessione aperta non più usata viene rimossa;

- Le opzioni di logging del codice sql auto-generato;

- I parametri di alcuni strumenti di Hibernate, tra cui `hibernate.hmb2ddl.auto` che consente l'esportazione automatica dello schema sql a partire dai descrittori XML mediante il tool *hbm2ddl* ;
- L'elenco di tutti i file XML di mappatura delle classi persistenti.

Alternativamente tutti i parametri di Hibernate visti possono essere inseriti nel file `hibernate.properties` posto sempre all'interno della cartella di lavoro. In appendice, è riportato il file di configurazione `hibernate.cfg.xml` utilizzato per questo progetto, dove si possono riconoscere gli elementi sopra citati.

Come si vedrà nel paragrafo successivo, queste specifiche vengono caricate durante la fase di startup e inserite all'interno dell'oggetto `SessionFactory`. Tale oggetto, infatti, contiene la configurazione di Hibernate prescelta e gioca il ruolo di gestore delle sessioni: ogni istanza dell'oggetto *Session* viene creata a partire da tale oggetto secondo le impostazioni definite. Di norma, all'interno di una applicazione, ne esiste una sola istanza che viene creata una sola volta durante la fase di inizializzazione di Hibernate.
Come consigliato dei realizzatori, a meno che non si lavori in ambiente JavaEE e quindi con strumenti più avanzati, è buona prassi realizzare una classe `HibernateUtil` (all'interno del package `persistence`) che provveda all'inizializzazione statica di Hibernate e alla gestione della singola istanza di *SessionFactory*. In questa classe l'oggetto *SessionFactory* viene istanziato all'interno di un blocco di inizializzazione statico. La classe `HibernateUtil` viene fornita dagli sviluppatori di Hibernate all'interno del package di installazione.
In definitiva, ogni qualvolta si deve accedere ad un oggetto *Session* dall'interno di una applicazione basta invocare il metodo:

```
HibernateUtil.getSessionFactory().openSession();
```

Startup

Un tipico esempio di procedura di startup è racchiuso nella seguente linea di codice:

```
SessionFactory sessionFactory = new Configuration().
   configure().buildSessionFactory();
```

In questa fase, Hibernate costruisce e inizializza l'oggetto *SessionFactory* a partire da un oggetto *Configuration*. Quando *new Configuration()* è chiamato, Hibernate cerca il file `hibernate.properties` nella cartella di lavoro e, nel caso questo sia presente, tutte le impostazioni lì definite vengono caricate e aggiunte all'oggetto *Configuration*. Quando invece è chiamato *configure()*, Hibernate cerca nella stessa cartella il file `hibernate.cfg.xml` e, nel caso questo non venga trovato, lancia un'eccezione. Le impostazioni definite al suo interno vengono, in modo analogo al caso precedente, caricate e aggiunge all'oggetto Configuration(). Nel caso siano contenute delle impostazioni presenti anche nel file `hibernate.properties`,

queste vengono sovrascritte.

Per ogni dettaglio o specifica si rimanda alla documentazione di Hibernate disponibile sul sito web del progetto.

Capitolo 7

Mappature

Lo sviluppo dell'applicazione è iniziato con la definizione delle classi di dominio come in qualsiasi applicazione Java. Le classi realizzate sono oggetti in stile Javabean che fungono da contenitori di informazioni rispecchiando le entità e le associazioni già definite nella progettazione della base di dati. Nell'ottica di Hibernate, questi oggetti fungono da "interpreti" del *mapping object-relazionale*.

7.1 I Javabean

Un Javabean è una classe Java che presenta una serie di campi, rappresentanti le proprietà dell'oggetto ed i relativi metodi get/set. Lo scopo del loro utilizzo come classi oggetto è quello di incapsulare al loro interno i dati presenti (o da inserire) nelle tabelle della base di dati.
Al fine di funzionare come una classe Javabean, la classe di un oggetto deve obbedire a determinate convenzioni in merito ai nomi, alla costruzione e al comportamento dei metodi. Queste convenzioni rendono possibile l'utilizzo di *tool* che possono usare, riusare, sostituire e connettere i JavaBean. Non solo: l'aspetto più importante è che, grazie alle proprietà della reflection di java, Hiberate riesce a interrogare la classe dell'oggetto ed operare sui metodi accessori (get e set) per accedere ai dati contenuti.

Le convenzioni richieste sono:

- La classe deve avere un costruttore senza argomenti;
- Le sue proprietà devono essere accessibili usando get, set, is (usato per i booleani al posto di get) e altri metodi (così detti metodi accessori) seguendo una convenzione standard per i nomi;
- La classe deve essere serializzabile (quindi implementare l'interfaccia `serializzable`);
- Non deve contenere alcun metodo richiesto per la gestione degli eventi.

Di norma quindi si rappresentano le classi definendo le rispettive variabili private ma rendendo pubblici i metodi di get/set/is e i metodi di business.

E' importante che tutte le classi mappate, che rispecchiano una tabella della base di dati, abbiano una proprietà rappresentante la chiave primaria che le identifichi univocamente. Nel caso in cui l'identificatore sia una chiave composta, si dovrà creare una classe oggetto interna avente tutte le proprietà che costituiscono tale chiave. Sarà questo nuovo oggetto a rappresentare la chiave composta nella classe persistente (si veda l'esempio 7.3.1). Inoltre è fondamentale che ogni classe ridefinisca i metodi `equals()` e `hashCode()` basandosi sul proprio oggetto o proprietà chiave, così da consentire ad Hibernate di comparare e distinguere le classi persistenti in modo corretto.

In appendice è riportato un esempio di classe persistente in stile Javabean rappresentante l'entità "Persona". Si noti come questo esempio segua le specifiche appena descritte e in particolare come al suo interno vengono implementati i metodi `equals` e `hashcode`, basati esclusivamente sulla variabile **"cf"** che rispecchia l'attributo chiave di *Persona "codice fiscale"*.

7.2 I Mapping

Per poter correttamente costruire un ponte tra il dominio di dati relazionale e quello a oggetti, Hibernate ha bisogno di conoscere le informazioni relative all'applicazione, come le proprietà delle classi e le relazioni esistenti fra loro. Tali informazioni vengono prelevate dai file XML di mappatura che accompagnano le varie classi oggetto. In tali file sono descritte le strutture delle tabelle della base di dati e le corrispondenze con le proprietà delle classi oggetto.
In generale nella struttura di ogni file di mappatura, si possono distinguere i seguenti elementi:

1. L'intestazione del file di mappatura:

   ```
   <?xml version="1.0"?>
      <!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD//EN"
      "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
   ```

   Al suo interno viene specificato l'indirizzo nel quale trovare il DTD per verificare la correttezza sintattica del file XML (se questo non è presente nella libreria .jar di Hibernate).

2. Il nome della classe persistente e della tabella a cui essa corrisponde, attraverso l'elemento `<class>`. Indica ad Hibernate come rendere persistente e caricare gli oggetti di una classe sfruttando la rispettiva tabella. Ogni sua istanza è, quindi, rappresentata da una tupla della tabella.

3. L'identificatore della classe, mappato attraverso il l'elemento `<id>`. Esistono due tipi di identificatori: quelli semplici nei quali si usa un unica proprietà

come chiave, e quelli composti nei quali la chiave è una combinazione di più proprietà.

4. Le proprietà della classe, mappate attraverso l'elemento `<property>`. Il mapping associa a ciascuna proprietà della classe un attributo della rispettiva tabella.

5. Eventuali associazioni con atri oggetti, mappate attraverso gli elementi `<one-to-many>`, `<many-to-one>` e `<many-to-many>`. Esistono, quindi, configurazioni diverse a seconda del tipo di associazione che intercorre fra questi oggetti.

7.3 Esempi

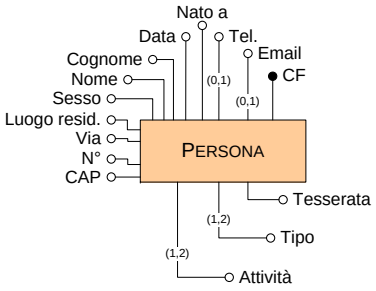
Nei prossimi paragrafi verrà analizzata più in dettaglio la composizione di questi file presentando alcuni esempi significativi.

7.3.1 Mappatura delle Entità

In questa sezione varranno presentati tre casi significativi di mappature di classi persistenti in tabelle del database. Tali classi rappresentano nel modello a oggetti le omonime entità dello schema ER. I casi presentati riguardano le mappature di entità con chiave semplice, composta ed esterna.

Mappatura di Persona

La classe persistente `Persona` rispecchia l'omonima entità dello schema ER riportata qui sotto assieme alla relativa tabella del modello relazionale. Il codice Java di questa classe è riportato completo in appendice.



PERSONA (Cf, Cognome, Nome, Data_Nascita, Sesso, Luogo_Residenza, Via, Numero, Cap, Luogo_Nascita, Tel, Email, Dirigente, Ciclista, MTB, Cicloturista, Tesserata)

Figura 7.1: L'entità e la relazione Persona

Questo caso di studio presenta un'entità dotata di chiave primaria semplice e alcuni attributi e rappresenta il caso di mappatura più semplice. Nel file di mappatura si può vedere l'uso di alcuni elementi XML descritti in precedenza, quali i tag <class>, <id> e <property>:

```
<class name="Persona" table="PERSONA">

<!-- Proprieta chiave di Persona -->
      <id name="cf"
       length="16">
      </id>

      <property name="nome"
       not-null="true"/>

    ...

</class>
```

La proprietà *"Sesso"* della classe `Persona` è di tipo *enum*, definita nella classe `Sesso.java`. La mappatura di questa proprietà richiede qualche specifica in più:

```
<!-- Proprieta di tipo enum. Il tipo 12 corrisponde al tipo base
    "String" -->
    <property name="sesso" column="sesso" length="1" not-null="
        true">
       <type name="org.hibernate.type.EnumType">
             <param name="enumClass">db.table.Sesso</param>
             <param name="type">12</param>
       </type>
    </property>
```

Mappatura di Trofeo

L'entità TROFEO presenta una chiave composta, è infatti costituita dalla coppia di attributi *edizione* e *nome*:

Figura 7.2: L'entità e la relazione Trofeo

Nella realizzazione della classe persistente `Trofeo` che la rappresenta, la chiave composta è stata rappresentata da una classe interna statica che contiene le proprietà `edizione` e `nome`:

```
public static class Id implements Serializable {

    private Byte edizione;
    private String nome;

    public Id() {}
```

```
    public Id(Byte edizione, String nome) {
        this.edizione = edizione;
        this.nome = nome;
    }

    \\Metodi hashCode() e equals()
}
```

Anche se non necessario, questo approccio di traduzione è da preferire in quanto la chiave composta diventa una classe-tipo ed è più funzionale sia all'interno dei metodi della classe TROFEO sia nelle operazioni di ricerca.

```
public class Trofeo implements Serializable {

    \\Classe interna Id
    ...

    private Id id;
    private TipoAccesso accesso;
    private short anno;

    public Trofeo() {};

    public Trofeo(byte edizione, String nome) {
        id = new Id(edizione,nome); }

    public Id getId() {return this.id;}
    ...

    \\Metodi hashCode() e equals()
```

La mappatura della chiave composta avviene con l'utilizzo del tag `<composite-id>`, specificando al suo interno la classe-chiave e le proprietà che la compongono:

```
<class name="Trofeo" table="TROFEO">

<!-- Chiave composta realizzata dalla classe interna Id -->
    <composite-id name="id" class="Trofeo$Id">
        <key-property name="edizione"/>
     <key-property name="nome"/>
  </composite-id>
...
```

Utilizzando lo strumento di generazione automatica del codice SQL **<hbm2ddl>** si può testare l'effettiva correttezza della mappatura. In questo caso il codice SQL ottenuto, è perfettamente congruente con quello definito durante la *progettazione fisica* della base di dati (5).

```
create table TROFEO (
    edizione int2 not null,
    nome varchar(255) not null,
    accesso varchar(6),
    anno int2,
    primary key (edizione, nome)
);
```

Mappatura di Amatore

La classe amatore rappresenta l'omonima entità debole. Tale entità viene identificata esternamente attraverso l'associazione RPA con PERSONA.

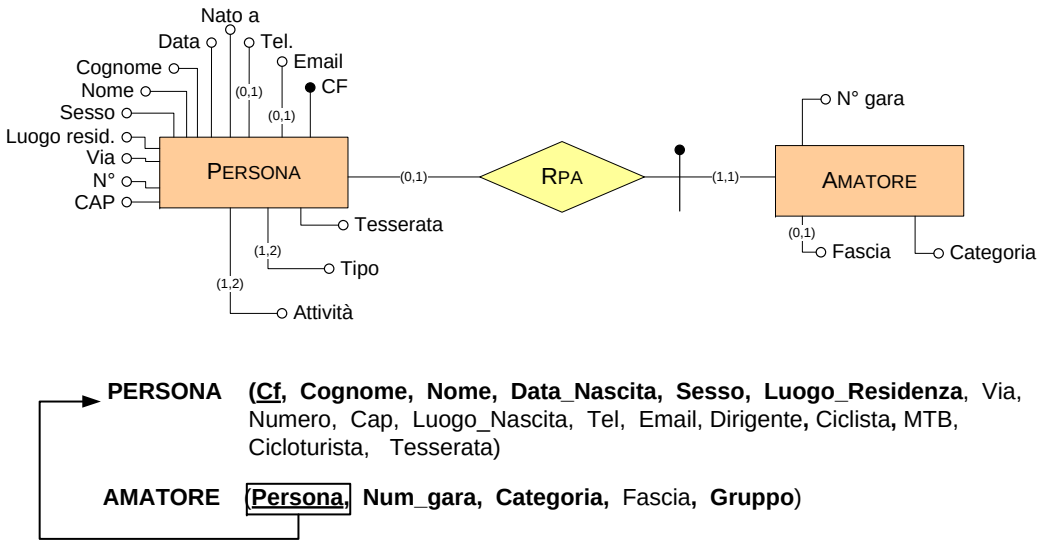


Figura 7.3: L'entità e la relazione Amatore

Nella classe `Amatore`, questa relazione è rappresentata inserendo un puntatore ad un oggetto `Persona` ed eliminando il metodo `setId()`.

```
public class Amatore implements Serializable {

    private String id;
    private Persona persona;
    ...

    // Costruttori
    ...

    public String getId() {return id;}

    public Persona getPersona() {return persona;}
    public void setPersona(Persona persona) {
        //Bidirezionale
        this.persona= persona;
        persona.setAmatore(this); }

    ...

    // Metodi hashCode() e equals()
}
```

Nel file di mappatura si crea una associazione uno a uno tramite il puntatore *persona* inserendo in aggiunta il comando `constrained=true`. Questo permette nella

dichiarazione della chiave, di indicare ad Hibernate attraverso il tag `<generator class=foreign»` di utilizzare quella della classe `persona`.

```
<class name="Amatore" table="AMATORE">

<!-- Chiave condivisa, dipendente dalla chiave di Persona-->
     <id name="id" column="fk_persona" length="16">
      <generator class="foreign">
         <param name="property">persona</param>
      </generator>
     </id>

   ...

 <!-- Associazione uno a uno con Persona che definisce la chiave
     condivisa.-->
   <one-to-one
      name="persona"
      class="Persona"
      constrained="true"/>
</class>
```

Per permettere la navigazione bi-direzionale tra gli oggetti `Amatore` e `Persona`, all'interno di quest'ultima è stato aggiunto un puntatore ad *amatore*. All'interno del relativo file di mappatura è stata aggiunta una associazione inversa basata su tale puntatore:

```
   ...
      <one-to-one
      name="amatore"
      class="Amatore"
      cascade="save-update"/>
   ...
```

Anche in questo caso il codice SQL ottenuto utilizzando lo strumento <hbm2ddl> è perfettamente congruente con quello definito durante la *progettazione fisica* della base di dati (5).

```
create table AMATORE (
    fk_persona varchar(16) not null,
    numero_gara varchar(4) not null unique,
    categoria varchar(3) not null,
    fascia varchar(2),
    fk_gruppo varchar(2),
    primary key (fk_persona)
);

alter table AMATORE
    add constraint FKF5958D033B45862E
    foreign key (fk_persona)
    references PERSONA;
```

7.3.2 Mappatura delle Associazioni

Mappatura di Rtgara

Questa associazione è di tipo *uno a molti* senza attributi e associa le entità Trofeo e Gara.

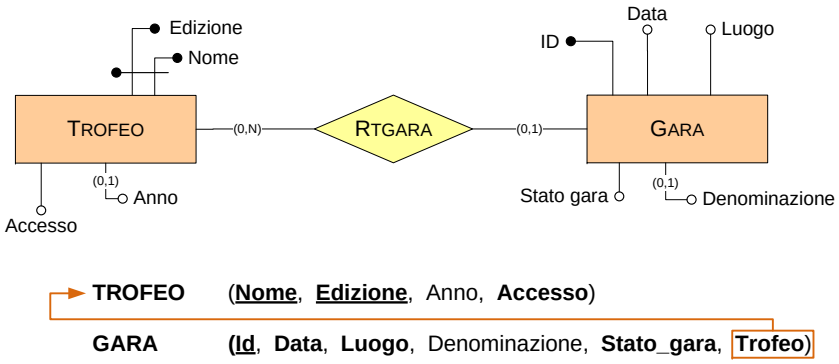


Figura 7.4: L'associazione Rtgara fra le entità Trofeo e Gara

Per realizzare questa mappatura è stato aggiunto all'interno della classe Gara un riferimento ad un oggetto Trofeo:

```
...
    private Trofeo trofeo;
...
```

Nel file di mappatura questa proprietà è stata mappata utilizzando l'elemento <many-to-one>. In questo modo, Hibernate, crea un vincolo interrelazionale fra la tabella gara e quella trofeo.

```
<many-to-one name="trofeo"
    cascade="save-update">
        <column name="fk_edizione_trofeo"/>
        <column name="fk_nome_trofeo"/>
</many-to-one>
```

Per rendere bi-direzionale l'associazione è stata inserita nella classe Trofeo una struttura dati che contenga tutti i riferimenti agli oggetti Gara. Tipicamente per mappare associazioni inverse si utilizza il contenitore *List*<>:

```
...
    private List<Gara> gare = new ArrayList<Gara>();
...
```

La sua mappatura è stata realizzata mediante l'utilizzo dell'elemento <bag>. In questo modo si ha un vantaggio in termini di prestazioni rispetto all'utilizzo di una struttura dati Set<> mappata tramite <set>: dato che non viene eseguito nessun controllo di duplicati, la sua inizializzazione richiede minor tempo. Dato che in una associazione inversa non possono mai essere presenti duplicati (poiché si andrebbe a violare il vincolo di chiave primaria nella relativa tabella), questa soluzione è da preferire.

```
<bag name="gare" inverse="true"
 cascade="save-update">
 <key>
       <column name="fk_edizione_trofeo"/>
       <column name="fk_nome_trofeo"/>
 </key>
 <one-to-many class="Gara" />
</bag>
```

Anche in questo caso, utilizzando lo strumento <hbm2ddl> si ottiene il risultato voluto.

```
create table AMATORE (
    ...
    fk_edizione_trofeo int2,
    fk_nome_trofeo varchar(255),
   ...
);

alter table GARA
     add constraint FK21448979EF2B71
     foreign key (fk_edizione_trofeo, fk_nome_trofeo)
     references TROFEO;
```

Mappatura di Reclamo

Questa mappatura rappresenta il caso più complesso affrontato in questo progetto. Si tratta di una associazione di tipo *molti a molti* con attributi e associa le entità VERBALE e AMATORE.

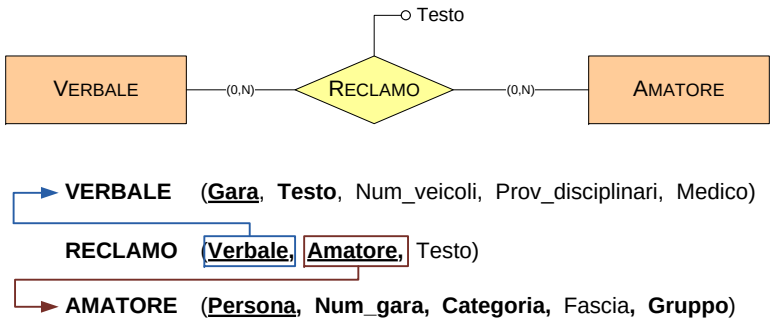


Figura 7.5: L'associazione RECLAMO fra le entità VERBALE e AMATORE

A differenza del caso visto in precedenza, per realizzare questa associazione si è dovuto creare la classe RECLAMO. Questa classe, oltre alle proprietà rappresentanti gli attributi dell'associazione, contiene una classe interna statica nella quale sono presenti i riferimenti alle proprietà chiave delle classi VERBALE e AMATORE.

```
public class Reclamo implements Serializable {

// Classe interna
public static class Id implements Serializable {
```

```
    private Long id_verbale;
    private String id_amatore;
...
}

// Variabili e Costruttori
private Id id;
private Verbale verbale;
private Amatore amatore;

private String testo = null;
...

public Reclamo(Verbale verbale, Amatore amatore, String testo) {
    this.verbale = verbale;
    this.amatore = amatore;

    this.testo = testo;

    //Set primary key
    id  = new Id(verbale.getIdGara(), amatore.getId()); }

// Metodi Set/Get e Equals
... }
```

La classe interna rappresenta, quindi, l'oggetto chiave della classe. Sotto è riportato il file di mappatura che mostra come è stata definita la chiave composta utilizzando i campi contenuti nella classe interna. Inoltre sono stati definiti i vincoli di integrità referenziale sulle chiavi utilizzando l'elemento <one-to-many>.

```
<class name="Reclamo" table="RECLAMO" mutable="false">

    <!-- Chiave composta realizzata dalla classe interna Id -->
        <composite-id name="id" class="Reclamo$Id">
            <key-property name="id_verbale" column="fk_verbale" />
            <key-property name="id_amatore" column="
                fk_persona_amatore" length="16"/>
        </composite-id>

        <property name="testo"
          length="500"
            not-null="true"/>

    <!-- Associazioni uno a molti read-only che realizzano
    i vincoli inter-referenziali degli attributi chiave-->

          <many-to-one name="verbale" column="fk_verbale"
            not-null="true"
            insert="false"
            update="false" />

          <many-to-one name="amatore" column="fk_persona_amatore"
            not-null="true"
            insert="false"
```

```
            update="false" />

    </class>
```

La prova della correttezza della mappatura così realizzata, ancora una volta, è data dallo strumento <hbm2dll>.

```
create table RECLAMO (
    fk_verbale int8 not null,
    fk_persona_amatore varchar(16) not null,
    testo varchar(500) not null,
    primary key (fk_verbale, fk_persona_amatore)
);

alter table RECLAMO
    add constraint FK6B510E475E690D89
    foreign key (fk_persona_amatore)
    references AMATORE;

alter table RECLAMO
    add constraint FK6B510E47B6006F0C
    foreign key (fk_verbale)
    references VERBALE;
```

Capitolo 8

DAO

Utilizzare un livello di persistenza generico permette di astrarre dalla tecnologia ORM usata e di fornire un livello di astrazione maggiore, concentrato esclusivamente sui servizi forniti. Permette infatti di nascondere l'implementazione effettiva del livello di persistenza e permette di far coesistere approcci misti all'ORM, come ad esempio Hibernate e JDBC.

8.1 Struttura e implementazione

Il livello di persistenza progettato è composto da due gerarchie parallele (fig. 8.1): le interfacce da un lato e le loro implementazioni dall'altro. Tutte le operazioni di persistenza di base, necessarie per ogni classe persistente (operazioni CRUD e metodi di ricerca), sono definite e raggruppate nella super-interfaccia generica `GenericDAO`. Una super-classe implementa tali operazioni adottando una particolare soluzione di persistenza: in questo caso la super-classe `GenericDAOHibernate` utilizza Hibernate. L'interfaccia generica viene estesa da altre sotto-interfacce, nelle quali vengono definite le specifiche operazioni d'accesso ai dati relative a ciascuna classe persistente. Anche in questo caso, è possibile avere una o più implementazioni di tali interfacce.
Sotto è mostrato come la classe `GenericDAO` raggruppa tutte le operazioni CRUD di base condivise da tutte le classi persistenti, come i metodi di ricerca `readById()`, `readAll()` o i metodi `saveOrUpdate()` e `delete()`:

```
public interface GenericDAO<T, ID extends Serializable> {
    T readById(ID id);
    List<T> readAll();
    List<T> readByExample(T exampleInstance, String...
        excludeProperty);
    T saveOrUpdate(T entity);
    void delete(T entity);
    void flush();
    void clear();
}
```

Eventualmente, può comprendere anche i metodi flush() e clear().
Una possibile sua implementazione mediante l'uso di Hibernate è riportata di se-

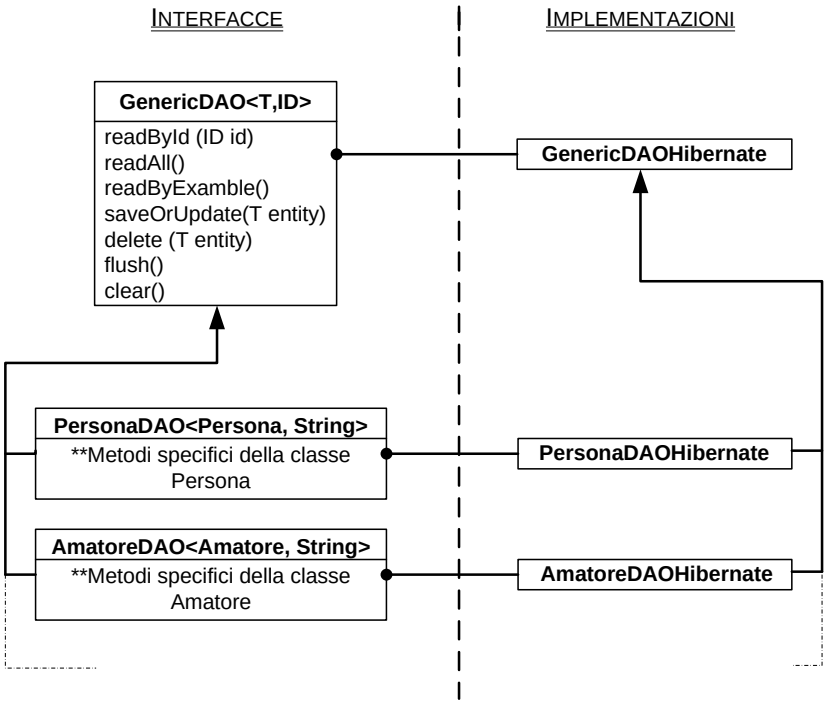


Figura 8.1: Struttura del livello generico di persistenza mediante l'uso dei DAO

guito dalla super-classe `GenericHibernateDAO` (in appendice è riportata integralmente). Questa classe include le proprietà specifiche di un livello di persistenza realizzato con Hibernate:

```
public abstract class GenericHibernateDAO<T, ID extends Serializable
    >
        implements GenericDAO<T, ID> {

    private Class<T> persistentClass;
    private Session session;

    public GenericHibernateDAO() {
        this.persistentClass = (Class<T>) ((ParameterizedType)
            getClass()
             .getGenericSuperclass()).getActualTypeArguments()[0];}

    public T readById(ID id) {
        T entity = (T) getSession().load(getPersistentClass(), id);
        return entity; }

    public T saveOrUpdate(T entity) {
      getSession().saveOrUpdate(entity);
        return entity; }

    public void delete(T entity) {
        getSession().delete(entity);}

...
```

```
}
```

8.2 Esempio: il DAO dell'oggetto Amatore

Per mostrare come sono state realizzate le sotto-interfacce comprendenti i metodi specifici delle classi, si è scelto come caso di studio quella relativa all'oggetto Amatore. Nell'esempio sottostante viene riportata la sotto-interfaccia `AmatoreDAO` e la sua implementazione mediante la sotto-classe `AmatoreDAOHibernate` che estende la super classe `GenericDAOHibernate`.

```
public interface AmatoreDAO extends GenericDAO<Amatore, String> {

    //Metodi per la navigazione tra oggetti
    Collection<Iscrizione> readIscrizioni(Amatore amatore);
    Collection<Abbuono> readAbbuoni(Amatore amatore);

    //Metodi specifici dell'oggetto
    long getConteggioAmatori(Gruppo gruppo);
    boolean isInGara(Gara gara, Amatore amatore);

public class AmatoreDAOHibernate
    extends GenericHibernateDAO<Amatore, String>
    implements AmatoreDAO {

    @Override
    public long getConteggioAmatori(Gruppo gruppo) {
        Query q = (Query) getSession().getNamedQuery("conteggioAmatori
            ");
         q.setParameter("gruppo", gruppo);
         return (Long)q.uniqueResult(); }

    ...

    //Metodi per la navigazione tra oggetti
    @Override
    public Collection<Iscrizione> readIscrizioni(Amatore amatore){
        return amatore.getIscrizioni();}

    ...
```

All'interno di questa classe si può vedere come sono stati implementati specifici metodi, come quelli che permettono la navigazione tra oggetti o l'esecuzione di una query.

8.3 L'interfaccia DAOFactory

All'interno di un eventuale controllore, gli oggetti DAO come quello appena visto, vengono costruiti e istanziati, così da poterli utilizzare per effettuare delle operazioni sulle relative classi persistenti. Per mantenere il livello d'astrazione voluto, senza inserire all'interno dei livelli applicativi superiori del codice legato ad una

specifica implementazione dei DAO, è stato necessario creare una struttura particolare. Si è ricorsi ad un modello basato sulla classe DAOFactory: tale classe viene istanziata con una sua implementazione specifica e, tramite essa, vengono creati tutti i DAO richiesti.

```
public abstract class DAOFactory {

    public static DAOFactory instance(Class factory) {
        try {
            return (DAOFactory)factory.newInstance();}
        catch (Exception ex) {
            throw new RuntimeException("Impossibile creare il
                DAOFactory: " + factory);
        }
    }

    // Elenco di tutti metodi astratti che restituiscono una istanza
        del rispettivo DAO
    public abstract AmatoreDAO getAmatoreDAO();
    public abstract PersonaDAO getPersonaDAO();
    ...
    }
```

La sua implementazione con Hibernate è rappresentata dalla classe sottostante:

```
public class HibernateDAOFactory extends DAOFactory {

    // Metodi pubblici che restituiscono una istanza del relativo
        DAO
    public AmatoreDAO getAmatoreDAO() {
        return (AmatoreDAO)instantiateDAO(AmatoreDAOHibernate.class
            );}

    ...

    private GenericHibernateDAO instantiateDAO(Class daoClass) {
            try {
                return (GenericHibernateDAO)daoClass.newInstance();}
            catch (Exception ex) {
                throw new RuntimeException("Impossibile istanziare il
                    DAO: " + daoClass, ex);}
    }
}
```

In questo modo, il codice all'interno dei controllori è indipendente dalle varie implementazioni dei DAO: eventuali modifiche legate al sistema di persistenza non coinvolgerebbero gli strati applicativi superiori. L'esempio riportato nel prossimo paragrafo mostra l'utilizzo della DAOFactory all'interno di un eventuale metodo controllore.

8.4 Esempio di uso dei DAO

E' riportato un tipico metodo di creazione e utilizzo dei DAO: per prima cosa viene istanziata la classe DAOFactory, facendo riferimento all'implementazione Hibernate usata, e in seguito, tramite tale classe, vengono creati gli oggetti DAO con i quali vengono effettuate le operazioni di persistenza richieste.

```
public utilizzoDAO(){
    \\
    DAOFactory daof = DAOFactory.instance(DAOFactory.HIBERNATE);

    // Preparazione dei DAO
    PersonaDAO personaDAO = daof.getPersonaDAO();
    AmatoreDAO amatoreDAO = daof.getAmatoreDAO();

    //Utilizzo dei DAO
    amatore.setPersona(persona);
    amatoreDAO.saveOrUpdate(amatore);

    ...
}
```

Capitolo 9

Conclusioni

L'obbiettivo di questo elaborato è stato quello di proporre e realizzare un sistema che fosse d'aiuto all'ambiente ciclistico amatoriale per gestire in maniera efficiente le attività sportive. Questo ambiente soffre di scarsi finanziamenti e di sistemi informativi inadeguati. Spesso non c'è la reale possibilità economica di adottare soluzioni informatiche commerciali efficienti e l'onere di compensare tale mancanza è affidato al lavoro manuale dei volontari.

Per questo si è cercato di realizzare una applicazione che fosse efficiente e flessibile ma allo stesso tempo poco costosa: da quì viene la scelta di utilizzare software liberi come: Java, Hibernate e PostgreSQL. Il vantaggio principale è quello di avere prodotti concreti e affidabili senza appesantire i costi di sviluppo. Inoltre, il software libero o open-source è spesso multi-piattaforma, ovvero non è legato ad un particolare sistema operativo e permette così di sviluppare una applicazione perfettamente portabile. La scelta che più ha caratterizzato questo progetto, è stata quella di utilizzare un database per la memorizzazione efficiente dei dati e il framework Hibernate per interfacciarlo con l'applicazione Java.

Per questo, nella prima parte di lavoro, si è concentrata l'attenzione nel progettare una base di dati capace di raccogliere e organizzare efficientemente tutte le informazioni che l'ambiente ciclistico amatoriale richiedeva, soddisfando le reali necessità di chi si occupa di tali eventi. La fase di *progettazione concettuale* è stata molto importante e impegnativa perché richiedeva di individuare, fra le molteplici possibilità di modellazione, quale fosse quella più efficiente e funzionale, capace di descrivere in maniera completa e ottimale la realtà. Particolare rilievo è stato dato anche alla fase di *progettazione logica* dove è stato necessario ristrutturare lo schema ER progettato nel punto precedente. Nella fase di *progettazione fisica* la base di dati ha trovato la sua effettiva implementazione. Il codice SQL prodotto è stato molto importante, infatti è stato utilizzato come linea guida durante la fase di mappatura delle classi.

Successivamente ci si è concentrati nello sviluppo dell'applicazione Java. Questa fase si incentrata sull'utilizzo di Hibernate come gestore della persistenza dei

dati. In partenza, si è dovuto comprendere come configurare correttamente il framework così da utilizzarlo nel modo migliore per soddisfare gli obiettivi che ci si era prefissati. In seguito, in maniera parallela, si sono realizzate le classi persistenti e le loro mappature verso il dominio relazionale. Questa fase ha necessitato di uno studio approfondito dei costrutti messi a disposizione da Hibernate, in quanto si è dovuto analizzare diverse tipologie di *mapping*, così da individuare quelle che portavano ad un risultato che fosse in accordo con la struttura progettata. Gli esempi riportati nel cap. 7 sono alcuni casi significativi e stanno a rappresentare l'intera fase di lavoro.

La scelta di utilizzare Hibernate ha portato vantaggi molto importanti. Il lavoro di interfacciamento tra l'applicazione e la base di dati è stata limitata alla sola realizzazione delle mappature, riducendo così drasticamente le righe di codice e il tempo di sviluppo necessario. Anche il tempo di debug è stato significativamente diminuito, essendo limitato alla sola verifica di correttezza delle mappature. L'utilizzo di Hibernate e del linguaggio Java ha reso questo progetto perfettamente scalabile e portabile, rendendolo indipendente dal sistema operativo e dal DBMS scelto. L'applicazione e la base di dati così strutturata è facilmente modificabile, capace di rispondere alle necessità sempre nuove che spesso emergono durante il ciclo di vita di una applicazione.

L'ultima fase di lavoro è stata quella di creare un livello di persistenza generico, capace di astrarre la tecnologia ORM utilizzata. In questo modo è possibile fornire ai livelli applicativi superiori uno strato d'accesso concentrato esclusivamente sui servizi forniti dal modello. Si predispone così il sistema in vista di eventuali piste di sviluppo o modifiche, permettendo anche la coesistenza di approcci misti all'ORM (per esempio affiancando JDBC o altri framework ad Hibernate).

In definitiva, gli strumenti utilizzati in questo progetto hanno permesso di creare un'applicazione che risponde perfettamente alle esigenze e agli obbiettivi posti in partenza. L'efficienza della base di dati e la flessibilità offerta da Hibernate sono stati i punti di forza del progetto. Una possibile pista di sviluppo ulteriore è quella di realizzare un'interfaccia grafica utilizzando le librerie *Java Swing*, così da sviluppare un'*applicazione desktop*. Alternativamente, si potrebbe combinare l'uso del framework Hibernate con un altro importante framework: Spring. In questo modo, grazie ai componenti che Spring mette a disposizione, si andrebbe a sviluppare un'applicazione orientata al web.

Appendice

hibernate.cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://www.hibernate.org/dtd/hibernate-configuration-3.0.
              dtd">

<hibernate-configuration>
<session-factory>

    <!--Parametri connesione DataBase Server (PostGreSQL) -->
        <property name="hibernate.connection.driver_class">org.
            postgresql.Driver</property>
        <property name="hibernate.connection.url">jdbc:postgresql://
            localhost/HibernateTest</property>
        <property name="hibernate.connection.username">postgres</
            property>
        <property name="connection.password">postgres</property>
        <property name="hibernate.dialect">org.hibernate.dialect.
            PostgreSQLDialect</property>

    <!-- Parametri configurazione C3P0 -->
        <property name="hibernate.c3p0.min_size">5</property>
        <property name="hibernate.c3p0.max_size">20</property>
        <property name="hibernate.c3p0.timeout">300</property>
        <property name="hibernate.c3p0.max_statements">50</property>
        <property name="hibernate.c3p0.idle_test_period">3000</
            property>

    <!-- SQL to stdout logging -->
        <property name="hibernate.show_sql">false</property>
        <property name="hibernate.use_sql_comments">false</property>
        <property name="hibernate.format_sql">true</property>

    <!-- Parametri di Hibernate -->

          <!-- Disable the second-level cache  -->
          <property name="cache.provider_class">org.hibernate.cache.
              NoCacheProvider</property>
          <property name="cache.use_query_cache">false</property>
          <property name="cache.use_minimal_puts">false</property>
```

```
35          <!-- Use thread-bound persistence context propagation,
                scoped to the transaction -->
            <property name="current_session_context_class">thread</
                property>
37          <property name="hibernate.order_updates">true</property>

39      <!-- parametri tools di Hibernate -->
            <!-- hbm2ddl.auto (validate | update | create | create-drop)
                -->
41          <property name="hibernate.hbm2ddl.auto">create-drop</property>

43      <!--Lista dei file XML di mappatura -->
            <mapping resource="db/table/Persona.hbm.xml"/>
45
            [...]
47
    </session-factory>
49  </hibernate-configuration>
```

Persona.java

```
1
    public class Persona implements Serializable {
3
        //Variabili e Costruttori
5       private String cf;
        private String nome;
7       private String cognome;
        private Sesso sesso;
9       private GregorianCalendar dataNascita;

11      ...

13      //Costruttore vuoto

15      public Persona() {}

17      //Costruttore Semplice
        public Persona(String cf) {
19          this.cf = cf;
        }
21
        //Metodi Get/Set
23      public String getCf() {return cf;}

25      public String getNome() {return nome;}
        public void setNome(String nome) {this.nome = nome;}
27
        ...
29
        //Metodi Comuni
31      @Override
```

```
    public int hashCode() {
33      return getCf().hashCode();
    }
35
    @Override
37  public boolean equals(Object obj) {
        if (this == obj) return true;
39        if (obj == null) return false;
          if (!(obj instanceof Persona)) return false;
41        final Persona that = (Persona) obj;
          return getCf().equals(that.getCf());
43  }
}
```

Persona.hbm.xml

```
<hibernate-mapping>
2   <class name="Persona" table="PERSONA">

4   <!-- Proprieta chiave di Persona -->
        <id name="cf"
6         length="16">
        </id>
8
        <property name="nome"
10        not-null="true"/>

12      <property name="cognome"
          not-null="true"/>
14
    <!-- Proprieta di tipo enum. Il tipo 12 corrisponde al tipo base
        "String" -->
16      <property name="sesso" column="sesso" length="1" not-null="
            true">
          <type name="org.hibernate.type.EnumType">
18              <param name="enumClass">db.table.Sesso</param>
                <param name="type">12</param>
20        </type>
        </property>
22
        <property name="dataNascita"
24        column="data_nascita"
          type="calendar_date"
26        not-null="true"/>

28    </class>
</hibernate-mapping>
```

GenericHibernateDAO.java

```
1  public abstract class GenericHibernateDAO<T, ID extends Serializable
       >
           implements GenericDAO<T, ID> {
3
       private Class<T> persistentClass;
5      private Session session;

7      @SuppressWarnings("unchecked")
       public GenericHibernateDAO() {
9          this.persistentClass = (Class<T>) ((ParameterizedType)
               getClass()
                .getGenericSuperclass()).getActualTypeArguments()[0];
11      }

13     public void setSession(Session s) {
           this.session = s;
15     }

17     protected Session getSession() {
         if (session == null)
19           session = HibernateUtil.getSessionFactory().
                 getCurrentSession();
           return session;
21     }

23     public Class<T> getPersistentClass() {
           return persistentClass;
25     }

27     @SuppressWarnings("unchecked")
       public T readById(ID id) {
29         T entity = (T) getSession().load(getPersistentClass(), id);
           return entity;
31     }

33     public List<T> readAll() {
           return findByCriteria();
35     }

37     @SuppressWarnings("unchecked")
       public List<T> readByExample(T exampleInstance, String...
           excludeProperty) {
39         Criteria crit = getSession().createCriteria(
               getPersistentClass());
           Example example =  Example.create(exampleInstance);
41         for (String exclude : excludeProperty) {
               example.excludeProperty(exclude);
43         }
           crit.add(example);
45         return crit.list();
       }
47
       public T saveOrUpdate(T entity) {
49       getSession().saveOrUpdate(entity);
```

```
            return entity;
51      }

53      public void delete(T entity) {
            getSession().delete(entity);
55      }

57      public void flush() {
            getSession().flush();
59      }

61      public void clear() {
            getSession().clear();
63      }

65      /**
         * Metodo accessorio utile all'interno delle sottoclassi.
67       */
        @SuppressWarnings("unchecked")
69      protected List<T> findByCriteria(Criterion... criterion) {
            Criteria crit = getSession().createCriteria(
                getPersistentClass());
71          for (Criterion c : criterion) {
                crit.add(c);
73          }
            return crit.list();
75      }

77  }
```

Bibliografia

[1] **C. Bauer, G. King**, *Java persistence with Hibernate*. Manning, 2007.

[2] **P.Atzeni, S. Ceri, S. Paraboschi, R. Torlone,** *Basi di dati, modelli e linguaggi di interrogazione*. McGraw-Hill, 3â edizione, 2009.

[3] **A. Elmastri, B. Navathe,** *Sistemi di basi di dati, fondamenti*. Pearson, 5â edizione, 2007.

[4] **M. Rampazzo,** *Progettazione e realizzazione di un sistema di gestione di esperimenti scientifici basato su architetture Spring-Hibernate*. Tesi di Laurea, Università degli studi di Padova, 2012.

[5] **M. Carraro,** *Studio di Hibernate attraverso i costrutti del modello relazionale*. Tesi di Laurea, Università degli studi di Padova, 2012.

[6] **A. Baudoin,** *Impara* "LaTeX!", Documentazione on-line di LaTeX, 1998.

[7] **Hibernate** `http://www.hibernate.org`

[8] **Postgresql** `http://www.postgresql.org`

[9] **Eclipse** `http://www.eclipse.org`

[10] **Supporto per la programmazione** `http://www.html.it`