

Tutorial JTCF-BASE

A cura di R. Gimelli, G. Necordi

V 1.0 - 2/2003

Jtech Srl tutti i diritti riservati
www.jtech.it

Sommario

1. Condizioni d'uso del documento.....	3
2. Marchi.....	3
3. Introduzione.....	4
4. Descrizione del database.....	5
5. Descrizione XML del database.....	7
6. Connessione al database.....	9
7. Persistenza dei dati.....	12
8. Descrizione XML del mapping delle business classes.....	14
9. Il motore di persistenza.....	17
10. Infrastrutture per lo sviluppo di applicazioni ed esempi di utilizzo.....	20
10.1 Le proprietà dell'ambiente applicativo.....	20
10.2 Proprietà.....	20
10.3 Servizi.....	21
10.4 Le Business classes.....	22
11. Un bocciata di... realtà.....	23

1. Condizioni d'uso del documento

Il documento è di proprietà degli autori che concedono il diritto d' usodi copia, diffusione e pubblicazione a condizione che venga riconosciuta la proprietà intellettuale dello stesso.

Il documento viene fornito senza alcuna garanzia implicita ed esplicita e nessuna responsabilità può essere ricondotta agli autori per eventuali danni - di qualunque genere e natura - che dovessero da esso derivare.

2. Marchi

JAVA™ è un marchio registrati di SUN MICROSYSTEMS.

IBM™, Microsoft™ e Oracle™ e Borland™ sono marchi registrati dai legittimi proprietari.

3. Introduzione

JTFC e' una raccolta di classi e di metodi volti a semplificare lo gestione della persistenza dei dati in Java.

Le caratteristiche:

- Supporto alla descrizione del database all' interno di un' applicazione.
- Supporto alla descrizione del mapping tra le proprietà (prpperties) delle business classes dell' applicazione ed i campi (fields) del database
- Supporto alla gestione delle infrastrutture di una applicazione.
- Supporto alla connessione del database tramite connessioni statiche o pool di connessioni.
- Supporto alla gestione dei dati del database tramite meccanismi CRUD (create, retrieve, update, delete) .
- Supporto alla personalizzazione dei metodi di retrieve tramite la descrizione dei criteri di estrazione.

4. Descrizione del database

La definizione del database avviene su tre livelli: database, tables, fields e keys.

La classe preposta alla definizione del database è **jtech.dictionary.JtDb**, essa contiene i seguenti metodi:

Metodo	Nota
<i>setName(String name)</i>	imposta il nome del database
<i>setAlias(String alias)</i>	imposta il nome alias del database (proprietà non implementata)
<i>setFlavor(String flavor)</i>	imposta il tipo di database da mappare (proprietà non implementata)
<i>addTable(JtTable table)</i>	aggiunge il mapping di una table al database
<i>Vector getTables()</i>	restituisce un vettore di oggetti di tipo JtTable contenente tutte le tables mappate al database

La classe preposta alla definizione delle tables è **jtech.dictionary.JtTable**, essa contiene i seguenti metodi:

Metodo	Nota
<i>setName(String name)</i>	imposta il nome della table
<i>setAlias(String alias)</i>	imposta il nome alias del database
<i>setDb(JtDb db)</i>	imposta il database di appartenenza della table
<i>setSchema(String schema)</i>	imposta lo schema del database della table
<i>addField(JtField table)</i>	aggiunge il mapping di una field alla table
<i>addKey(JtKey table)</i>	aggiunge il mapping di una key alla table

La classe JtTable possiede anche alcuni metodi di utilità:

Metodo	Nota
<i>getCreateSql()</i>	restituisce la stringa SQL necessaria alla creazione della table
<i>getDropSql()</i>	restituisce la stringa SQL di drop della tabella

La classe preposta alla definizione dei fields è **jtech.dictionary.JtField**, essa contiene i seguenti metodi:

Metodo	Nota
<i>setName(String name)</i>	imposta il nome dei fields
<i>setAlias(String alias)</i>	imposta il nome alias del fields (proprietà non implementata)
<i>setType(String type)</i>	imposta il tipo del field, sono validi tutti i tipi dati di SQL standard
<i>setLength(int length)</i>	imposta la lunghezza del field
<i>setDecimal(int decimal)</i>	imposta il numero di decimali del field, se previsti dal tipo impostato
<i>SetNotNull(boolean notNull)</i>	definisce se il field accetta valori nulli
<i>SetTable(JtTable table)</i>	imposta la table di appartenenza del field

La classe preposta alla definizione delle chiavi è **jtech.dictionary.JtKeys**, tramite questa definizione è possibile definire le chiavi primarie (primary keys) della table.

La classe contiene i seguenti metodi:

Metodo	Nota
<i>setIsPrimary(boolean isPrimary)</i>	definisce se la key è una chiave primaria, in realtà questa impostazione non è implementata in quanto una chiave è sempre chiave primaria
<i>setField(JtField field)</i>	definisce il field come chiave primaria

All' internodi una table non vi è limite nel numero delle chiavi primarie definibili, queste possono essere di qualsiasi tipo SQL valido, ne deve esistere almeno una. Non esiste gerarchia di chiavi, questo significa che definendone più d' una sono tutte chiavi primarie. I campi chiave devono essere mappati nella table per primi.

La descrizione del database avviene tramite la scrittura di un apposito file XML

5. Descrizione XML del database

La struttura del file XML per la descrizione delle tabelle del database, definita dal DTD database.dtd, ha come elemento root il tag `<database>`, che contiene, nell'ordine, gli elementi:

<code><datasource></code>	flavour del database, ovvero il tipo di DBMS cui viene effettuata la connessione
<code><database-name></code>	nome del database
<code><database-alias></code>	l' alias del database

Questi sono seguiti da uno o piu' elementi `<table>`, ognuno dei quali descrive una tabella nel database. Il tag `<table>` contiene nell'ordine i seguenti elementi:

<code><table-name></code>	nome della tabella
<code><table-alias></code>	l' alias della tabella
<code><table-schema></code>	schema/database di appartenenza della tabella

Seguono quindi uno o piu' elementi `<table-field>`, ognuno dei quali definisce un campo nella tabella. Il tag `<table-field>` contiene nell'ordine i seguenti elementi:

<code><field-name></code>	nome del campo
<code><field-alias></code>	l' alias del campo
<code><field-type></code>	tipo del campo, deve essere un tipo SQL (es. decimal, varchar, date, ...)
<code><field-length></code>	l' ampiezza del campo, per i tipi che ne prevedono l' impostazione (elemento comunque obbligatorio)
<code><field-decimal></code>	numero di decimali a del campo, utilizzato per i tipi che ne prevedono l' impostazione (elemento comunque obbligatorio)
<code><not-null></code>	opzionale, se presente indica che il campo non puo' assumere il valore NULL
<code><isprimary></code>	opzionale, se presente indica che il campo fa parte della chiave primaria della tabella cui appartiene

Ecco un esempio di definizione del database:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE database SYSTEM "database.dtd" >

<database>
  <datasource>MYSQL</datasource>
  <database-name>jtech</database-name>
  <database-alias>jtech</database-alias>

  <!-- definizione delle tabelle -->
  <table>
    <table-name>clienti</table-name>
    <table-alias>clienti</table-alias>
    <table-schema>jtech</table-schema>

    <!-- definizione dei campi -->
    <table-field>
      <field-name>idcliente</field-name>
      <field-alias>idcliente</field-alias>
      <field-type>decimal</field-type>
      <field-length>12</field-length>
      <field-decimal>0</field-decimal>
      <not-null></not-null>
      <isprimary></isprimary>
    </table-field>

    <table-field>
      <field-name>descrizione</field-name>
      <field-alias>descrizione</field-alias>
      <field-type>varchar</field-type>
      <field-length>20</field-length>
      <field-decimal>0</field-decimal>
    </table-field>
  </table>

  <table>
    <table-name>fattura</table-name>
    <table-alias>fattura</table-alias>
    <table-schema>jtech</table-schema>

    <table-field>
      <field-name>idfattura</field-name>
      <field-alias>idfattura</field-alias>
      <field-type>decimal</field-type>
      <field-length>12</field-length>
      <field-decimal>0</field-decimal>
      <isprimary></isprimary>
    </table-field>

    <table-field>
      <field-name>idcliente</field-name>
      <field-alias>idcliente</field-alias>
      <field-type>decimal</field-type>
      <field-length>12</field-length>
      <field-decimal>0</field-decimal>
      <not-null></not-null>
    </table-field>

    <table-field>
      <field-name>data</field-name>
      <field-alias>data</field-alias>
      <field-type>date</field-type>
      <field-length>8</field-length>
      <field-decimal>0</field-decimal>
    </table-field>
  </table>
</database>
```

6. Connessione al database

La connessione al database viene gestita dalla classe **jtech.persistence.JtTransactionManager**. Questa classe utilizza due metodologie differenti di connessione al database: tramite connessioni "statiche" o tramite pool di connessioni.

Le connessioni statiche sono identificate da un nome che viene assegnato al momento della loro creazione, gli statement vengono associati al nome della connessione, ne può esistere uno per connessione.

La prima connessione creata diventa la ' defaultconnection' ,ovvero la connessione utilizzata dalle operazioni sul database nel caso non venga specificato su quale connessione operare.

JtTransactionManager fornisce una serie di metodi statici a supporto delle connessioni statiche:

Metodo	Nota
<i>RegisterTM(String name, String driver, String url, String user, String pwd)</i>	registra una connessione assegnandogli il nome specificato nel parametro <i>name</i> e associandola al database localizzato dal parametro <i>url</i> e gestibile tramite il driver (jdbc o jdbc:odbc) specificato dal parametro <i>driver</i> . Il nome utente e la password assegnati dagli ultimi due parametri sono utilizzati solo dai driver che gestiscono questi valori.
<i>Connection getConnectionTM(String name)</i>	restituisce la connessione associata al nome specificato nel parametro <i>name</i>
<i>String getDefaultConnectionNameTM()</i>	restituisce il nome della connessione di default
<i>boolean setAutoCommit(String name, boolean state)</i>	valorizza l' opzione <i>autoCommit</i> della connessione identificata dal primo parametro al valore specificato dal secondo
<i>setTransactionIsolation(String name, int level)</i>	assegna il livello di isolamento nel caso le operazioni effettuate sulla connessione identificata dal primo parametro avvengano sotto ciclo di sincronia
<i>boolean commit(String name) rollback(String name)</i>	confermano o annullano le modifiche effettuate sotto controllo di sincronia sulla connessione identificata dal nome passato come parametro

Il pool di connessioni permette una gestione più trasparente delle connessioni, è inoltre fondamentale nello sviluppo di applicazioni client server Web-oriented, dove il numero di client è molto variabile e non si vuole correre il rischio di sovraccaricare il database server con connessioni statiche.

In sintesi il ciclo di vita del pool di connessioni è il seguente:

- L' applicazione (solitamente risiedente sul server) inizializza il pool di connessioni all' atto della partenza, vengono aperte un numero prestabilito di connessioni
- Quando un client sollecita una operazione sql sul database, il gestore del pool di connessioni fornisce la prima connessione libera tra quelle disponibili e la segna come "in uso".
- Il gestore può aprire nuove connessioni, fino ad un massimo prestabilito, se al momento di una richiesta da un nuovo client non vi sono più connessioni libere ovvero tutte le connessioni sono occupate da altri client
- Se tutte le connessioni sono occupate ed è stato raggiunto il massimo consentito, il client che richiede una connessione viene messo in attesa con un ciclo di retry parametrizzabile.
- terminate le operazioni sql il client rilascia la connessione.
- Il gestore del pool si occupa di mantenere periodicamente lo stato di connessioni, in particolare se un client ha superato il tempo massimo stabilito di utilizzo di una connessione, questa viene rilasciata.

E' possibile gestire pools di connessioni attivi per diversi databases. Il primo pool aperto assume anche il valore di pool default. In caso di più pool di connessioni è necessario specificare il nome logico del database così come definito nel mapping.

E' importante notare che è possibile utilizzare il pool di connessioni anche tramite applicazioni Stand-alone, questo consente di avere una gestione più agevole delle connessioni. Gli esempi legati a questo tutorial utilizzano un pool di connessioni.

JtTransactionManager fornisce una serie di metodi per la gestione del pool di connessioni:

Metodo	Nota
<i>createPool(Properties properties, Properties dbProperties)</i>	<p>Inizializza ed apre il pool di connessioni, l' <i>eencadbProperties</i> contiene le impostazioni specifiche del database se necessarie mentre l' elenco <i>properties</i> contiene tutti i parametri di configurazione:</p> <ul style="list-style-type: none"> • JtConnectionPool.DATABASE_NAME - il nome logico del database da connettere, se non specificato viene creato un pool con un nome di default • JtConnectionPool.DRIVER - driver JDBC del database • JtConnectionPool.URL - url del database da connettere • JtConnectionPool.USER - utente del database da connettere • JtConnectionPool.PASSWORD - password del database da connettere • JtConnectionPool.MIN_CONNECTIONS - numero di connessione che il gestore delle pooled connections deve aprire in partenza • JtConnectionPool.MAX_CONNECTIONS - numero massimo di connessioni aperte consentite • JtConnectionPool.CLIENT_RETRIES - numero di tentativi da effettuare in caso di connessioni occupate durante la richiesta di un client. • JtConnectionPool.RETRY_TIMEOUT - tempo di attesa tra un tentativo e l' altro (milliSecondi) • JtConnectionPool.CONNECTION_TIMEOUT - timeout oltre il quale una connessione in uso viene dichiarata scaduta, 0 imposta un timeout infinito (milliSecondi) • JtConnectionPool.CONTROLLER_DELAY - Tempo di attesa del controller tra una verifica e l' altra delle connessioni
<i>closePool()</i>	chiude tutti i pool di connessioni aperti
<i>closePool(String dbName)</i>	chiude il pool di connessioni al database specifico
<i>Connection getPooledConnection()</i>	restituisce una pooled connection da l pool di default, al termine la connessione deve essere rilasciata con il metodo close() di Connection
<i>Connection getPooledConnection(String dbName)</i>	restituisce una pooled connection dal pool specifico, al termine la connessione deve essere rilasciata con il metodo close() di Connection
<i>boolean isPoolControlled()</i>	restituisce true se il TransactionManager è in regime di pooled connections
<i>jtech.util.JtTableModel getPoolStatus()</i>	restituisce un table model contenente lo stato delle connessioni

7. Persistenza dei dati

La logica di persistenza dei dati gestiti di una business class viene demandata alla superclasse specializzata **jtech.persistence.JtPersistenceEntity** che implementa l'interfaccia **jtech.persistence.JtPersistentObject**

La classe JtPersistenceEntity provvede a generare ed eseguire le transazioniSQL per il salvataggio, aggiornamento, lettura e cancellazioni dei dati gestiti dalle business classes. Per fare ciò si appoggia al mapping tra le proprietà delle business classes ed i campi del database, il mapping viene *registrato* con l'ausilio di alcune classi specializzate.

La classe preposta alla descrizione di una property di una business class è **jtech.mapper.JtPropertyMapper**, essa contiene i seguenti metodi:

Metodo	Nota
<i>setPropertyName(String name)</i>	registra il nome della property da mappare al field, il nome della property deve essere specificate senza la particella "get" o "set" iniziale. Se la property della business class è un tipo dati composto in questo campo è necessario esprimere il nome della property del tipo dati che mappa il field
<i>setPropertyType(String type)</i>	registra il tipo della property, nel caso di tipi composti deve essere specificato il tipo risultante dal metodo utilizzato per fornire il valore da scrivere nella table (metodo get), esso deve inoltre coincidere con il tipo del metodo set

La classe preposta alla descrizione del mapping tra un field ed una property è **jtech.mapper.JtFieldMapper**, essa contiene i seguenti metodi:

Metodo	Nota
<i>setField(JtField newValue)</i>	registra il JtField contenente la descrizione del campo del database da mappare alla property
<i>AddPropertyMapper(JtPropertyMapper newValue)</i>	registra il propertyMapper da mappare al field, se la proprietà fa parte di un data type, o una gerarchia di data type, devono essere registrate una di seguito all'altra tutte le proprietà a partire dalla prima di tipo semplice, che è mappata al field, ed a salire quelle di tipo composto

La classe preposta alla descrizione delle relazioni tra le istanze delle business classes è **jtech.mapper.JtRelationMapper**; questa metodologia può essere utilizzata quando un' istanza di una business class contiene un *contenitore* di istanze di un' altrabusiness class, ad esempio una fattura contiene un' insieme di righe di dettaglio fattura, sicché effettuando il retrieve dei dati della fattura automaticamente viene effettuato il retrieve del dettaglio fattura nell' apposito contenitore. La relazione tra oggetti viene implementata specificando quali *properties* dell' oggetto *relante* (la fattura), sono legate alle *properties* dell' oggetto *relato* (il dettaglio fattura).

Ecco i principali metodi:

Metodo	Nota
<i>AddRelatedProperties(JtPropertyMapper propertyFrom, JtPropertyMapper propertyTo)</i>	aggiunge la relazione tra due property, la prima dell' oggetto relante e la seconda per l' oggetto relato. Possono venire mappate un numero infinito di relazioni
<i>setContainerMethod(String newValue)</i>	imposta il nome dei metodi di gestione del container degli oggetti relati. Dal nome devono essere escluse le particelle "get" o "set" in quanto gestite in automatico. Il container deve essere di tipo vector, perciò il metodo set deve avere un solo parametro di tipo Vector, mentre, analogamente, il metodo get deve restituire un tipo <i>Vector</i>
<i>setRelatedObject(JtPropertyMapper per newRelatedObject)</i>	imposta il JtPropertyMapper che rappresenta l' oggetto, ad esempio il dettaglio fattura sarà registrato con <i>propertyName</i> = "DettaglioFattura" e <i>propertyType</i> = "jtech.samples.DettaglioFattura"
<i>setDeletePolicy(boolean newDeletePolicy)</i>	true se la cancellazione persistente dell' oggetto relante provoca la cancellazione degli oggetti relati

La classe che fornisce i metodi e mantiene la registrazione del mapping dell' applicazione è **jtech.mapper.JtRelationMapper**, essa contiene i seguenti metodi:

Metodo	Nota
<i>registerMapper(String className, JtFieldMapper fieldMapper)</i>	registra, per una business class, il mapping tra le property (registrate nell' oggetto JtFieldMapper) i campi del database. Il metodo aggiunge il mapping al contenitore dei mappings delle properties.
<ul style="list-style-type: none"> <i>registerRelation(String className, String relationName, JtRelationsMapper relationsMapper)</i> 	registra, per una business class, le relazioni con altre business class. Il metodo aggiunge la relazione al contenitore delle relazioni

La descrizione del mapping delle business class al database avviene tramite la definizione di un apposito file XML.

8. Descrizione XML del mapping delle business classes

La struttura del file XML per la descrizione del mapping, definita dal DTD `properties.dtd`, ha come elemento root il tag `<properties-mapping>`, che contiene uno o piu' elementi di `<class>`, ognuno dei quali definisce il mapping di una business class, le sue proprieta' ed eventuali relazioni con le altre business classes. L' elemento `<class>` contiene gli elementi:

<code><class-name></code>	nome della business class
<code><property></code>	proprieta' della business class ed il suo mapping alla campo del database. Ogni elemento <code><class></code> puo' contenere uno o piu' elementi <code><property></code> ognuno dei quali definisce il mapping di una delle proprieta' della business class
<code><relation></code>	opzionale, definisce una relazione tra una property della business class e una property di un' altra business class. Ogni elemento <code><class></code> puo' contenere zero o piu' elementi <code><relation></code> , ognuno dei quali definisce una relazione tra le business class e le altre

Ogni elemento `<property>` contiene i seguenti elementi che definiscono il mapping tra la property della business class e le tabelle del database:

<code><property-name></code>	nome della proprieta' della business class
<code><property-type></code>	tipo Java della proprieta' ; se si tratta di un tipo diverso da uno primitivo bisogna qualificare completamente il nome del tipo (es. <code>jtech.util.JtData</code>)
<code><table-name></code>	nome della tabella su cui mappare la property (relativa alla definizione del database)
<code><field-name></code>	nome del campo della tabella su cui mappare la property (relativa alla definizione del database). NOTA: gli elementi <code><table-name></code> e <code><field-name></code> sono opzionali, perche' qualora la property fosse un tipo complesso, non mappato direttamente in su un campo di una tabella questi elementi vanno ommessi.

se la property e' di tipo complesso occorre tramite elementi `<property>` nidificati andare ad definire il mapping delle proprieta' elementari della proprieta' complessa. Ad esempio per una property di tipo ***jtech.util.JtData***:

```

<property>
  <property-name>dataFattura</property-name>
  <property-type>jtech.util.JtData</property-type>
  <property>
    <property-name>dataSql</property-name>
    <property-type>String</property-type>
    <table-name>fattura</table-name>
    <field-name>data</field-name>
  </property>
</property>

```

dove l' elemento `<property>` nidificato definisce il mapping verso le tabelle del database della proprieta' elementare di tipo `String` della proprieta' complessa di tipo `JsonData`.

Un esempio piu' complesso e' il seguente, dove la proprieta' complessa articolo articolo viene mappata nelle sue componenti elementare nelle tabelle del database tramite gli elementi `<property>` nidificati:

```
<property>
  <property-name>articolo</property-name>
  <property-type>jtech.dynamo.esempi.Articolo</property-type>
  <property>
    <property-name>gruppoMerceologico</property-name>
    <property-type>String</property-type>
    <table-name>dettaglio</table-name>
    <field-name>gruppo</field-name>
  </property>
  <property>
    <property-name>codiceArticolo</property-name>
    <property-type>String</property-type>
    <table-name>dettaglio</table-name>
    <field-name>codiceart</field-name>
  </property>
  <property>
    <property-name>descrizioneArticolo</property-name>
    <property-type>String</property-type>
    <table-name>dettaglio</table-name>
    <field-name>descrizione</field-name>
  </property>
  <property>
    <property-name>prezzoUnitario</property-name>
    <property-type>float</property-type>
    <table-name>dettaglio</table-name>
    <field-name>importo</field-name>
  </property>
</property>
```

Gli elementi `<property>` possono anche essere nidificati a piu' livelli.

Ogni elemento `<relation>` contiene invece i seguenti elementi che definiscono una relazione tra un insieme di proprietà della business class con un insieme di proprietà di un' altra classe:

<code><relation-name></code>	nome della relazione
<code><container-name></code>	nome della proprietà della classe, di tipo <code>Vector</code> , che conterra' gli oggetti relati
<code><multiplicity></code>	elemento opzionale, se presente viene interpretato come valore <code>true</code> .
<code><delete-policies></code>	<p>elemento opzionale, ha un attributo type che puo' assumere i valori <code>error</code> o <code>delete</code>. Se l' elemento e' presente e' necessario specificare l' attributo, i cui valori assumono il significato:</p> <ul style="list-style-type: none"> • error: la cancellazione di una istanza delle classe che abbia oggetti relati su questa relazione non viene eseguita, e viene lanciata una eccezione di tipo <code>JtDeleteException</code> dal metodo <code>delete()</code> • delete: la cancellazione di una istanza della classe che abbia oggetti relati su questa relazione provoca la eliminazione dell' oggetto e di tutti gli oggetti relati <p>Se l' elemento non e' presente la cancellazione viene effettuata per l' oggetto senza effettuare verifiche su questa relazione.</p>
<code><load-on-retrieve></code>	elemento opzionale, se presente indica che la retrieve di un oggetto deve anche provocare la retrieve degli oggetti relati su questa relazione che andranno a popolare il <code>Vector</code> definito dall' elemento <code><container-name></code>
<code><related-object></code>	<p>elemento che definisce la classe verso la quale e' definita la relazione. Contiene a sua volta i seguenti elementi:</p> <ul style="list-style-type: none"> • <code><object-name></code> nome della classe relata (a livello di mapping, corrispondente all' elemento <code><class-name></code> per la classe relata. • <code><object-type></code> tipo Java completamente qualificato della calsse relata (es. <code>jtech.dynamo.esempi.DettaglioFattura</code>)
<code><relation-property></code>	<p>elemento che definisce la relazione tra una proprietà della classe e quella della classe relata. Possono essere presenti uno o piu' di questi elementi, nel caso la relazione fosse impostata su piu' di una proprietà (chiavi multiple). Contiene a sua volta i seguenti elementi:</p> <ul style="list-style-type: none"> • <code><relating-property-name></code> nome della proprietà della business class • <code><relating-property-type></code> tipo Java della proprietà della business class • <code><related-property-name></code> nome della proprietà nell' oggetto relato su cui si definisce la relazione • <code><related-property-type></code> tipo della proprietà nell' oggetto relato

9. Il motore di persistenza

E' tempo di tornare alla superclasse **jtech.persistence.JtPersistenceEntity**, i metodi *esposti* dall' interfaccia sono i seguenti:

Metodo	Nota
<i>boolean save()</i>	effettua il salvataggio dei dati della business class tramite un statement SQL INSERT, l' accesso al database fisico viene garantito dalla connessione di default oppure tramite una pooled connection (vedi <i>jtech.persistence.JtTransactionManager</i>). L' equivalente metodo <i>save(String connectionName)</i> accede al database attraverso una specifica connessione
<i>boolean update()</i>	effettua l' aggiornamento dei dati della business class tramite un statement SQL UPDATE utilizzando la connessione di default oppure tramite una pooled connection. L' equivalente metodo <i>update(String connectionName)</i> accede al database attraverso una specifica connessione
<i>boolean delete()</i>	effettua la cancellazione dei dati della business class tramite un statement SQL DELETE utilizzando la connessione di default oppure tramite una pooled connection. L' equivalente metodo <i>delete(String connectionName)</i> accede al database attraverso una specifica connessione. La selezione del set dati da cancellare avviene tramite il valore delle properties mappate ai campi definiti come primary keys
<i>boolean retrieve()</i>	effettua la valorizzazione delle properties della business class tramite un statement SQL RETRIEVE utilizzando la connessione di default oppure tramite una pooled connection. L' equivalente metodo <i>retrieve(String connectionName)</i> accede al database attraverso una specifica connessione. La selezione del set dati da valorizzare avviene tramite il valore delle properties mappate ai campi definiti come primary keys
<i>java.util.Vector retrieveByAlternateKey(Criteria criteria, Object[] values, String[] orderBy)</i>	effettua la valorizzazione delle properties della business class tramite un statement SQL RETRIEVE utilizzando la connessione di default oppure tramite una pooled connection. La selezione del set dati da valorizzare avviene tramite l' uso di campi come chiavi alternative (o meglio delle relative properties a loro mappate) identificati dall' istanza della classe specializzata <i>jtech.persistence.Criteria</i> la quale produce una clausola WHERE personalizzata. Gli altri due parametri identificano l' array di valori da utilizzare per il retrieve dei dati specificati nello stesso ordine con cui sono stati specificate le properties nel criterio di selezione e il nome delle properties da utilizzare come ordinamento del result set. Attenzione, nell' array dei valori i tipi java primitivi devono essere specificati tramite le corrispondenti classi (int -> Integer, boolean -> Boolean, long -> Long ecc.). Il risultato della retrieve sarà un Vector contenente gli oggetti della business class che soddisfano il criterio di ricerca
<i>int countByAlternateKey(Criteria criteria, Object[] values, String[] orderBy)</i>	restituisce il numero di record selezionati tramite uno statement SQL RETRIEVE COUNT(*) utilizzando la connessione di default. Per l' uso dei parametri valgono le stesse note del metodo <i>retrieveByAlternateKey</i>
<i>SetDirty(int dirty)</i>	Imposta il tipo di trattamento che le istanze di una business class presenti nel contenitore di un oggetto relante subiranno in automatico allo scatenarsi di un evento di save, update o delete. Valori validi sono <i>JtPersistenceEntity.SAVE</i> , <i>JtPersistenceEntity.UPDATE</i> e <i>JtPersistenceEntity.DELETE</i>

Come appena esposto la classe **jtech.persistence.Criteria** permette di creare clausole WHERE personalizzate. Questo è possibile tramite la registrazione di una serie di condizioni legate alle properties della business class, ovviamente la clausola WHERE viene costruita utilizzando il nome dei campi del database mappati alle properties indicate. La clausola costruita viene automaticamente utilizzata dalla logica del metodi `retrieveByAlternateKey` e `countByAlternateKey`.

Le condizioni vengono costruite tramite la classe **jtech.persistence.SimpleCondition**, la quale contiene i seguenti campi e costruttore:

Variabili e costruttori	Nota
<code>public static final int EQ</code>	condizione equal
<code>public static final int NE</code>	condizione not equal
<code>public static final int GT</code>	condizione greater then
<code>public static final int GE</code>	condizione greater equal
<code>public static final int LT</code>	condizione less then
<code>public static final int LE</code>	condizione less equal
<code>public static final int LIKE</code>	condizione like
<code>SimpleCondition(String propertyName, int condition)</code>	il costruttore crea una simple condition tramite il nome della property alla quale associare una condizione e la condizione da utilizzare

La classe **jtech.persistence.Criteria** contiene i seguenti campi, costruttori e metodi:

Variabili, costruttori, metodi	Nota
<code>public static final int NOP</code>	nessun operatore
<code>public static final int AND</code>	operatore <i>and</i>
<code>public static final int NAND</code>	operatore <i>and not</i>
<code>public static final int OR</code>	operatore <i>or</i>
<code>public static final int NOR</code>	operatore <i>or not</i>
<code>Criteria(String name, String className)</code>	nome descrittivo del criterio e nome della business class
<code>add(int operator, SimpleCondition condition)</code>	aggiunge un operatore ed una SimpleCondition al criterio, la prima SimpleCondition deve essere registrata con operatore NOP
<code>void add(int operator, Criteria criteria)</code>	aggiunge un operatore ed un criterio al criterio attuale. Questo metodo permette di comporre dei metodi complessi con l' uso di parentesi. Il criterio aggiunto verrà estratto all' interno di parentesi tonde. Il primo criterio deve essere inserito con operatore NOP

Esempio: assumendo che le properties *codiceArticolo* e *totale* della business class *DettaglioFattura* siano mappate rispettivamente ai fields *codice* e *totale* della table *dettaglio*, il seguente codice produce la clausola *WHERE dettaglio.codice = ? AND dettaglio.totale = ?*

```
Criteria criteria = new Criteria("test criteria", "DettaglioFattura");
SimpleCondition simple = new SimpleCondition("codiceArticolo", SimpleCondition.EQ );
criteria.add(Criteria.NOP, simple);
simple = new SimpleCondition("totale", SimpleCondition.EQ);
criteria.add(Criteria.AND, simple);
```

10. Infrastrutture per lo sviluppo di applicazioni ed esempi di utilizzo

Questa sezione descrive, con esempi, lo sviluppo completo di una applicazione.

Le necessità evidenziate come requirements necessari sono:

- Le proprietà dell' ambiente applicativo. Dove e come gestirle.
- Logica applicativa. Le business classes.
- La persistenza. Connettività al Database.
- L' interfaccia grafica.

10.1 Le proprietà dell' ambiente applicativo

Si intende per ambiente l' insieme di proprietà e metodi che permettono ad un' applicazione di sapere quali sono le sue impostazioni standard, quali attributi delle sue classi sono persistenti, su quale database è gestita la persistenza e quali sono le relazioni fra tabelle del database e classi dell' applicazione denominate *business class*. Gli oggetti necessari a creare questo ambiente devono essere creati in un package dedicato che conterrà almeno due classi, una che gestisce i valori di default delle proprietà dell' applicazione, l'altra che fornisce i servizi che implementano la persistenza.

10.2 Proprietà

Le proprietà sono valori, gestiti come stringhe di caratteri, che permettono ad un programma di conoscere le sue impostazioni. Per esempio le proprietà che devono sempre essere definite per utilizzare i servizi di persistenza sono:

- il driver che gestisce il database;
- l' indirizzo del database;
- l' utente che si connette al database (per i driver che lo gestiscono);
- la password dell' utente.
- Il nome del file di definizione del database
- Il nome del file di definizione del mapping delle properties
- Parametri di inizializzazione delle pooled connection

Le proprietà possono essere scritte in un qualsiasi file testo che sarà il file di inizializzazione dell' applicazione. L' esempio che segue mostra la definizione delle proprietà di base:

```
# database connection definition
url=jdbc:mysql://localhost/mydb
driver=com.mysql.jdbc.Driver
user=myuser
```

```
password=mypass

# database definition
xmldbfile=/home/user/myapp/db.xml
# properties mapping definition
xmlmapperfile=/home/user/myapp/properties.xml

# min number of open connection to db
min_connection=1
# maxi number of open connection to db
max_connection=1
# pooled connections manager settings
client_retries=3
retry_timeout=3000
connection_timeout=300000
controller_delay=60000
```

Nel file di inizializzazione è possibile definire qualsiasi altra proprietà gestita dal programmatore utile alla definizione dell' applicazione.

Nota: i valori assegnati alle proprietà fanno riferimento alla sintassi necessaria ad interfacciarsi ad una connessione al database mySql. Per maggiori informazioni su come identificare l' URL di un database ed il suo driver consultare la documentazione relativa al package java.sql e la documentazione del database che si intende utilizzare.

10.3 Servizi

I servizi che implementano la persistenza permettono di:

- creare, inizializzare, modificare, cancellare il database dell' applicazione;
- connettersi al database;
- identificare quali proprietà delle classi che costituiscono l' applicazione devono essere rese persistenti;
- identificare su quali tabelle devono essere salvate le proprietà persistenti;
- identificare quali relazioni, intese come dipendenze padre - figlio, ci sono fra le classi che costituiscono l' applicazione
- associare i campi che mantengono in relazione le tabelle del database alle relazioni fra le classi.

La classe da implementare per gestire questi servizi è un' estensione della classe astratta `JtApplicationServer`, dalla quale eredita i metodi di gestione dei servizi dell' applicazione; più importanti sono:

Metodi	Nota
<code>public void startServices()</code>	Avvia i servizi dell' applicazione, ovvero carica le proprietà di sistema e apre le connessioni al database
<code>public void stopServices()</code>	Chiude l' applicazione terminando le connessioni al database
<code>public String getProperty(propertyName)</code>	Restituisce il valore di una qualsiasi proprietà scritta nel file di inizializzazione
<code>public void createDB()</code>	Crea tutte le tabelle definite nel mapping del database

10.4 Le Business classes

Possiamo definire come business class una classe che descrive la logica applicativa di un particolare problema di un' applicazione, ad esempio un classe anagrafica deve avere determinate proprietà (nome, cognome ecc.) alcune delle quali devono essere obbligatorie, altre devono rispondere a determinati comportamenti ecc..

Le business classes demandano la gestione della persistenza alla superclasse *JtPersistenceEntity* la quale provvede ad interpretare il mapping e ad salvare i valori delle proprietà sul database o a ripristinare valori dal database alle proprietà.

Dai vincoli esposti nella spiegazione delle definizioni tramite XML si evincono alcune regole per il design delle business class:

- Le proprietà che gestiscono la persistenza devono essere procedute dalle preposizioni *get* e *set* (getter/setter methods)
- Le proprietà set devono accettare un solo parametro del tipo specificato nel mapping, il relativo metodo get deve ritornare un valore dello stesso tipo.
- Le proprietà get e set devono essere public

<PENDING.....>

11. Un bocciata di... realtà

L' applicazione MyApplication gestisce, in modo molto semplicistico, la gestione delle fatture e del relativo elenco clienti. L' esempio proposto è stato testato con successo con MySQL.

Per utilizzare un altro database server o comunque personalizzare secondo le esigenze, occorre modificare le proprietà di connessione nel file di inizializzazione *JtProperty.ini*:

Proprietà	Nota
<code>url=jdbc:mysql://localhost/mydb</code>	L' url del database server secondo le specifiche del driver JDBC utilizzato, per semplicità si consiglia comunque di mantenere uguale il nome del database (mydb).
<code>driver=com.mysql.jdbc.Driver</code>	Il driver JDBC utilizzato
<code>user=myusr</code> <code>password=mypwd</code>	L' utente e la password di connessione al database.

Per fare un esempio una connessione a SQL Server potrebbe essere impostata in questo modo:

```
url=jdbc:inetdae7://SQLS:1433/mydb
driver=com.inet.tds.TdsDriver
user=myusr
password=mypwd
```

Se viene utilizzato un database server diverso da MySQL è necessario modificare la configurazione del datasource nel file XML di definizione del database db.xml:

```
<datasource>MYSQL</datasource>
```

I valori possibili sono:

Valore	Nota
<code>SQL SERVER</code>	database Microsoft Sql Server
<code>MySQL-MAX</code>	database server MySQL con la gestione delle transazioni
<code>ORACLE</code>	database server Oracle
<code>INTERBASE</code>	database server Borland Interbase
<code>DB2-AS/400</code>	database server DB2 su AS/400
<code>DB2-UDB</code>	database server DB2 UDB

Un'altra configurazione da curare nel *JtProperty.ini* è la locazione dei file XML di definizione del database e del mapping delle business class:

```
xmldbfile=/home/user/myapp/db.xml
xmlmapperfile=/home/user/myapp/properties.xml
```

L'esempio proposto descrive la locazione dei file su un sistema linux, su sistemi operativi Microsoft Windows la configurazione potrebbe essere:

```
xmldbfile=d:/myapp/db.xml
xmlmapperfile=d:/myapp/properties.xml
```

Vediamo ora brevemente la struttura dell'applicazione.

Il database mydb è composto dalle seguenti tabelle:

- *Fattura* - la testata delle fatture
- *Dettaglio* - il dettaglio delle fatture
- *Clients* - l'elenco dei clienti

L'applicazione implementa le seguenti business classes:

- *Fattura* - descrive i dati della fattura e implementa un contenitore di oggetti relativi rappresentanti il dettaglio specifico della fattura
- *DettaglioFattura* - descrive il dettaglio della fattura, ogni riga di dettaglio fa riferimento ad un articolo gestito da un tipo dati complesso implementato tramite la classe *Articolo*.
- *Cliente* - descrive una entità cliente (probabilmente destinatario di una fattura)

L'ambiente applicativo è gestito da:

- *MyNewApplicationServices* - Contiene i servizi di mapping, connettività ecc.

Infine una classe di test che genera registrazioni nel database ed effettua letture:

- *TestMyApplication*

Diamo un'occhiata ad una parte del main della classe *TestMyApplication*:

```
// inizializza l'applicazione
MyNewApplicationServices application = new MyNewApplicationServices();
application.setPropertyFilePath("/home/uneco/local_repository/gneco/");
application.setPropertyFileName("JtProperty.ini");
try {
    // start dell'applicazione
    application.startServices();
    // crea le tabelle nel database
    application.createDb();
    // crea clienti
    creaClienti();
    // crea fatture
    creaFatture();
<...>
} catch (Throwable th) {
    System.out.println(th);
} finally {
    application.stopServices();
}
```

Come appare evidente viene istanziato un oggetto della classe `MyNewApplicationServices`, il servizio di inizializzazione dell' applicazione, il quale provvederà a registrare il database, il mapping delle business class e ad aprire una pool di connessioni al database. Per fare ciò è necessario fornire all' oggetto il nome ed il path del file di proprietà (nell' esempio `JtProperty.ini`).

Lo startup dell' applicazione avviene tramite il metodo `startServices()` il quale provvede a caricare le proprietà ed ad effettuare la connessione al database.

Il metodo `createDB` permettere di creare le tabelle definite nel database, è evidente che questo servizio viene utilizzato un tantum in fase di deployment dell' applicazione.

In seguito vengono richiamati due metodi della classe `TestMyApplication` per inserire dati nelle tabelle appena create.

Nella parte rimanente del main vengono proposte alcune tecniche di *retrieve* dei dati utilizzando criteri di selezione personalizzati.

Il test dell' applicazione è semplice:

- creare un database MySQL denominandolo **mydb**
- Installare i sorgenti proposti nell' ambiente di sviluppo, attenzione: i sorgenti risiedono nel package `jtfc.samples`
- Montare nell' ambiente di sviluppo la libreria contenente il motore di persistenza JTFC (la distribuzione ufficiale è disponibile sul sito www.jtech.it) e la libreria dei driver JDBC del database. Nota bene: se non viene utilizzato alcun ambiente di sviluppo, le librerie menzionate devono essere specificate nel classpath.
- Compilare i sorgenti
- Lanciare la classe `TestMyApplication`