Sommario

Indice generale

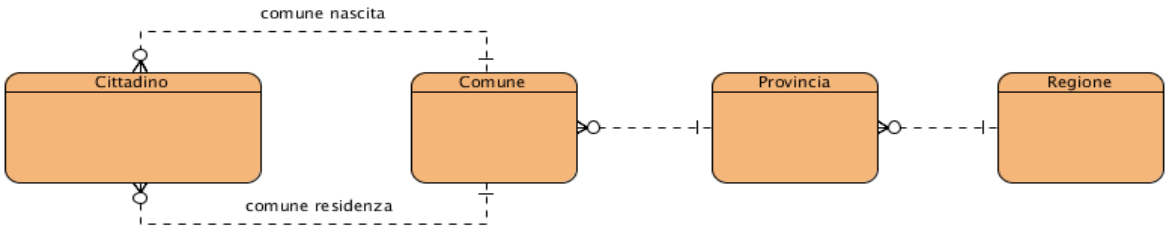
1 Descrizione generale della Reference Application

Lo scopo di questo documento è di descrivere le scelte implementative per la realizzazione di un'applicazione di riferimento che descrive l'utilizzo dei tools MDD.
I tools utilizzati nella reference application sono GUIGEN per la generazione della user interface e DATAGEN per la generazione del layer di accesso ai dati.

1.1 Database di riferimento

Il database di riferimento utilizzato per la realizzazione dell'applicazione è preesistente ed è il db delle reference application di Efesto. Di tale database sono state utilizzate le sole tabelle relative ai dati anagrafici del Cittadino; Le entità utilizzate, con le relative tabelle, sono le seguenti:

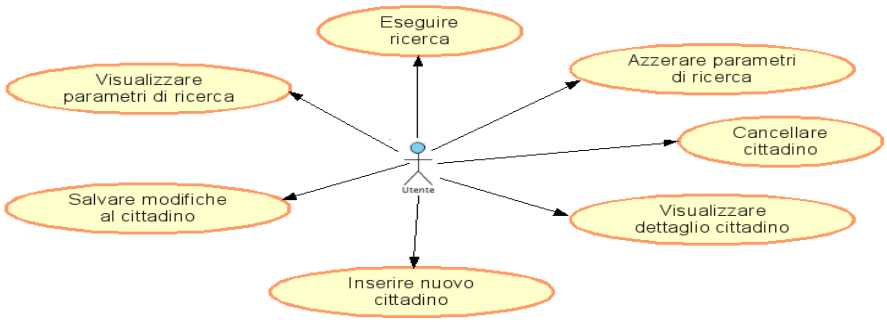
- **Cittadino** (tabella REFAPP_T_CITTADINO): contiene le informazioni anagrafiche di un generico cittadino e la relativa retribuzione
- **Comune** (tabella REFAPP_D_COMUNE): contiene le informazioni relative ad un generico comune. È referenziata dall'entità Cittadino per la specifica dei comuni di nascita e residenza.
- **Provincia** (tabella REFAPP_D_PROVINCIA): contiene le informazioni relative alle singole provincie. È referenziata dall'entità "Comune" per l'associazione Provincia-Comune
- **Regione** (tabella REFAPP_D_REGIONE): contiene le informazioni relative alle singole regioni. È referenziata dall'entità Provincia per l'associazione Provincia-Regione.



1.2 Struttura delle schermate

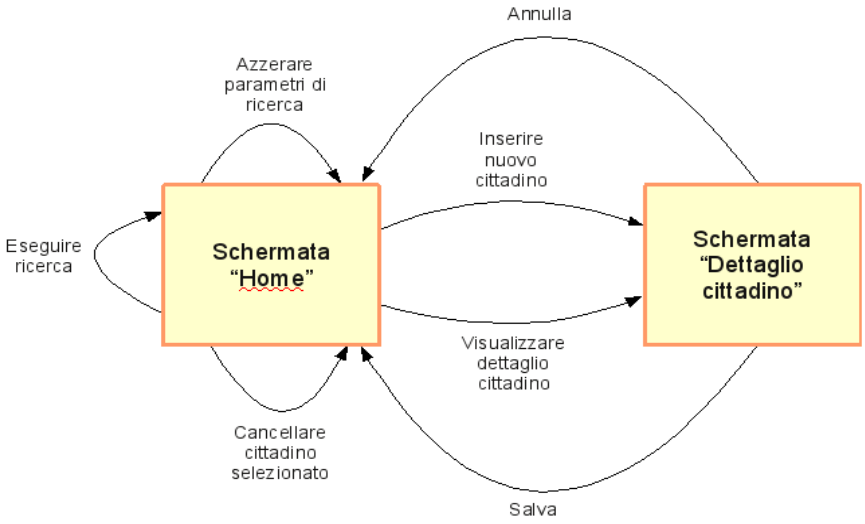
1.2.1 Funzionalità implementate

La reference application implementa, sulle tabelle relative all'anagrafica del cittadino, le principali funzioni di CRUD. I casi d'uso implementati sono descritti dal seguente diagramma:



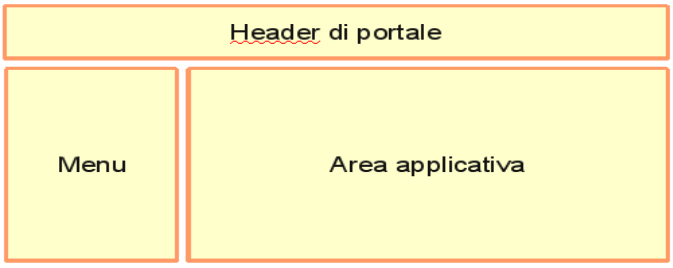
1.2.2 Interazione e navigazione fra le schermate

La Reference Application è stata costruita su due schermate distinte: la prima, chiamata “Home” fornisce le funzionalità di ricerca e navigazione sui record dei cittadini; la seconda, chiamata “Dettaglio Cittadino” fornisce le funzionalità di visualizzazione di dettaglio di un cittadino, di modifica dei dati esistenti e di inserimento di un nuovo record.
Le due schermate interagiscono fra loro a seconda dei comandi (o casi d'uso) eseguiti dall'utente. Il seguente diagramma di navigazione mostra le interazioni fra le due schermate.



1.2.3 Struttura generale delle schermate

Una tipica struttura[1] delle applicazioni gestionali è quella illustrata in figura:



dove:

- l'**header di portale** ha lo scopo di fornire informazioni di contestualizzazione sul singolo portale (RUPAR, Comune Torino,...). È dipendente dal cliente e viene generato in modo automatico dai tools MDD. Non viene considerato all'interno di questo esempio.
- L'area relativa al **menu** ha lo scopo di fornire uno spazio in cui visualizzare il menù dell'applicazione. La posizione del menù può variare a seconda del portale utilizzato (ad esempio può essere visualizzato in orizzontale al di sotto dell'header).

[1] La struttura effettiva da adottare nelle applicazioni in progetti reali dipende da vari fattori tra i quali: il portale di pubblicazione, la tipologia di utenza, ...

- All'interno dell'**Area Applicativa** vengono inseriti tutti i pannelli relativi alle funzionalità delle applicazioni.

1.2.4 Struttura della schermata Home

La schermata "home", ha lo scopo di fornire le funzioni per ricercare le informazioni relative al cittadino e di richiamare la schermata di dettaglio per la visualizzazione/modifica/inserimento dei dati relativi ai cittadini. A parte le componenti generalizzate (menu e header), è stata modellata la seguente struttura di pannelli:

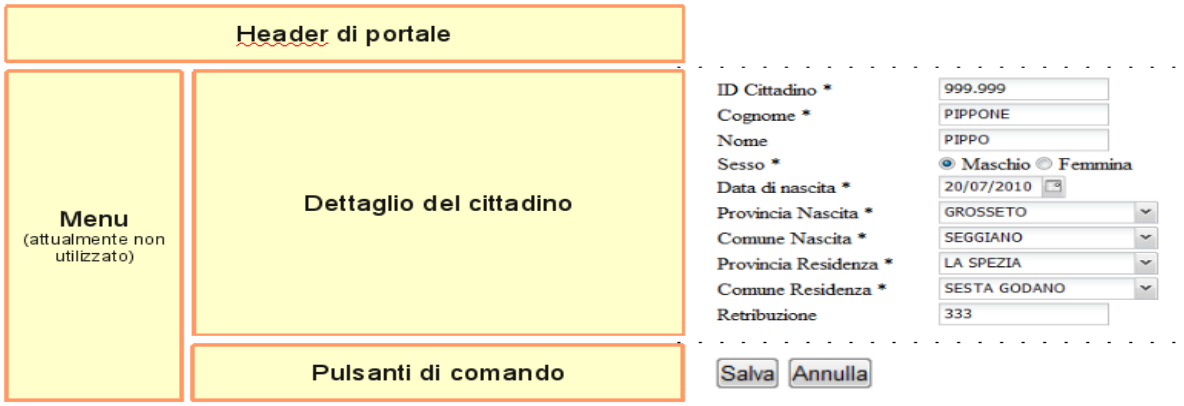
- **Pannello "Parametri di Ricerca"**: ha lo scopo di visualizzare i campi in cui specificare i parametri per l'esecuzione della ricerca sull'anagrafica dei cittadini. La ricerca avviene in modalità AND (tutti i campi per i quali sia impostato un valore concorrono al criterio di ricerca e sono da considerarsi in AND logico fra loro). A scopo esemplificativo sono stati previsti i seguenti parametri di ricerca (e i relativi widget per l'impostazione dei filtri corrispondenti):
  - **Regione**: rappresenta la regione di residenza. È un campo di tipo intero che contiene il codice identificativo della regione. Per semplificare l'utilizzo dell'applicazione da parte dell'utente viene impostato tramite una combo-box contenente l'elenco delle regioni italiane (chiave, descrizione). L'elenco viene caricato dal DB (tabella REFAPP_D_REGIONE).
  - **Provincia:** rappresenta la provincia di residenza. È un campo di tipo intero che contiene l'identificativo della provincia. Per semplificare l'utilizzo dell'applicazione da parte dell'utente viene impostato tramite una combo-box contenente l'elenco delle provincie relative alla regione selezionata in precedenza (le due combo-box sono quindi sincronizzate fra loro). L'elenco viene caricato dal DB (tabella REFAPP_D_PROVINCIA).
  - **Cognome:** è un campo di testo, di tipo stringa, destinato a contenere il cognome (o una sua parte) del cittadino da ricercare. All'interno di questo campo è possibile utilizzare i caratteri jolly di oracle (%) in quanto la ricerca avviene in modalità "LIKE".
  - **Retribuzione:** campo di tipo *long* destinato a contenere il valore della retribuzione da ricercare. Nell'esempio attuale, la ricerca sulla retribuzione è *puntuale*; non sono previsti range
- **Pannello "Comandi di Ricerca":** contiene i pulsanti di comando per pilotare la ricerca:
  - **Cerca:** esegue la ricerca con i parametri specificati e aggiorna la tabella dei risultati

- **Azzera parametri:** azzera tutti i parametri di ricerca e annulla eventuali risultati di ricerche precedenti

- Pannello "Risultato della ricerca": contiene una tabella che presenta i risultati della ricerca. La tabella, essendo implementata in modalità "ricca", consente all'utente di scegliere quali colonne visualizzare e l'ordinamento delle colonne. Inoltre, la tabella è paginata ed è possibile esportare i dati in CSV e PDF.
- Pannello "Comandi relativi alla tabella": Contiene i pulsanti di comando che operano sul record selezionato sulla tabella. Al termine dell'esecuzione dei singoli comandi, viene eseguito il refresh della tabella. Attualmente non è stata prevista la selezione multipla di più record. I comandi sono:
  - Cancella: esegue la cancellazione fisica del record selezionato.
  - Inserisci: consenti di inserire un nuovo record; viene mostrata la schermata di dettaglio (vuota) tramite la quale inserire le informazioni del nuovo cittadino
  - Modifica: consente di visualizzare il dettaglio di un cittadino e di eseguire eventuali modifiche sui dati anagrafici dello stesso. Viene mostrata la schermata di dettaglio con precaricati i dati del cittadino selezionato.

1.2.5 Struttura della schermata di dettaglio

Lo scopo della schermata di dettaglio è di fornire gli strumenti per l'inserimento dei dati relativi ad un nuovo cittadino o di modificare i dati relativi ad un cittadino selezionato in precedenza.
A parte le componenti generalizzate (menu e header), sono stati inseriti i seguenti pannelli:



- **Pannello "Dettaglio del cittadino"**: contiene i campi di dettaglio del cittadino:
  - **ID Cittadino:** campo di testo (*TextField*) associato all'identificativo del cittadino (intero) . Allo stato attuale, questo campo obbligatorio, deve essere inserito manualmente dall'utente in fase di inserimento, in quanto sul database di test non è stata definita una sequence per tale campo.
  - **Cognome:** campo di testo (*TextField*), obbligatorio, utilizzato per la visualizzazione, inserimento e/o modifica del campo cognome del cittadino (stringa).
  - **Nome:** campo di testo (*TextField*), opzionale, utilizzato per la visualizzazione, inserimento e/o modifica del campo nome del cittadino.
  - **Sesso:** insieme di radio-button, che permette di scegliere tra due opzioni: "M" per maschio e "F" per femmina. Contiene l'indicazione del sesso del cittadino.
  - **Data Nascita:** campo di tipo *Calendar*, associato alla data di nascita del cittadino.
  - **Provincia Nascita:** C*omboBox* che permette di visualizzare o impostare la provincia di nascita del cittadino. L'elenco delle opzioni è alimentato con l'insieme di tutte le provincie italiane e il

valore corrispondente alla selezione è rappresentato dall'identificativo della provincia selezionata (campo intero).
  - **Comune Nascita:** *ComboBox* che permette di visualizzare o impostare il comune di nascita del cittadino. L'elenco delle opzioni è alimentato in base alla provincia selezionata in precedenza e il valore della selezione è rappresentato dall'identificativo del Comune selezionato.
  - **Provincia Residenza:** come per il campo "provincia nascita" ma associato alla provincia di residenza.
  - **Comune Residenza:** come per il campo "comune nascita" ma associato alla provincia di residenza.
  - **Retribuzione:** campo di testo (*TextField*) associato alla retribuzione del cittadino (*long).*
- **Pannello "Pulsanti di comando"**: contiene i seguenti pulsanti:
  - **Salva:** salva il nuovo record (se si tratta di un nuovo inserimento) o i dati modificati di un record visualizzato (se si tratta di modifica). La schermata **Dettaglio Cittadino** è condivisa fra le operazioni di inserimento e modifica del cittadino.
  - **Annulla:** annulla l'operazione in corso e torna alla pagina **Home**.

2 Predisposizione dell'ambiente

In questo paragrafo vengono elencati i passi necessari per la predisposizione degli ambienti necessari per lo sviluppo e l'esecuzione della reference application.

Per ottenere un ambiente completo utilizzabile per l'installazione, il build e l'esecuzione della reference application è necessario compiere i seguenti passi:

1. predisposizione della postazione di sviluppo MDD
2. predisposizione dell'ambiente di esecuzione

2.1 Predisposizione dell'ambiente di sviluppo MDD

L'ambiente di sviluppo MDD è basato sull'I.D.E Eclipse, le cui funzionalità sono aumentate tramite una serie di plugin specifici sviluppati in CSI che permettono di modellare le differenti parti di applicazione e generare il codice corrispondente.

I prerequisiti per una postazione di lavoro sono i seguenti:
- JDK 1.5 (o superiori)
- almeno 1Gb di memoria RAM (consigliati 2Gb)
- almeno 1Gb di spazio su disco

Le attività necessarie per la predisposizione della postazione sono invece le seguenti:
1. installazione del *bundle* base di Eclipse
2. installazione delle estensioni di eclipse di MDDTools

La documentazione di dettaglio che descrive la modalità di distribuzione dei tool MDD, le istruzioni di installazione oltre ad altre utili informazioni, sono disponibili all'URL:
http://dsp.csi.it/dsp/opencms/dsp/it/standard/add/MDD_materiale.html

Si consiglia di fare riferimento a tali istruzioni. Per semplicità di seguito si riporta un riassunto delle attività da compiere.

2.1.1 Installazione del bundle base di *eclipse*

Il bundle base di *eclipse* fornito dal CSI è un package preconfezionato dal CSI che contiene:

- il runtime dell'IDE *Eclipse* (attualmente nella versione 3.5 - "galileo")
- le estensioni necessarie per lo sviluppo secondo le tecnologie e il framework al momento in vigore in CSI:
  - strumenti di base per lo sviluppo in linguaggio Java
  - strumenti aggiuntivi per lo sviluppo di applicazioni J2EE (Web Tools Project)
  - integrazione embedded con server J2EE Jboss (per WLS è disponibile la modalità tramite debug remoto)
  - integrazione con SVN (subclipse)
- le tecnologie base per il model-driven-development, sulle quali sono basati i tool MDD
  - emf
  - openarchitectureware

Il bundle base è disponibile per il download all'url:
http://dsp.csi.it:81/std/strumenti/mddtools/eclipse-base/galileo-base-20100501.jar

(N.B.: il link di download è anche disponibile nella sezione "distribuzione dello strumento" della pagina:
http://dsp.csi.it/dsp/opencms/dsp/it/standard/add/MDD_materiale.html)

Per l'installazione del bundle è sufficiente scompattare in una directory a piacere l'archivio scaricato.
N.B: in caso si verificassero problemi durante lo scompattamento utilizzare il tool "jar" per lo scompattamento.

2.1.2 Installazione dei plugin MDDTools

Una volta installato il bundle base è necessario installare i plugin che costituiscono la suite di tool MDD.
Nella reference application sono utilizzati i seguenti plugin (nelle versioni indicate):

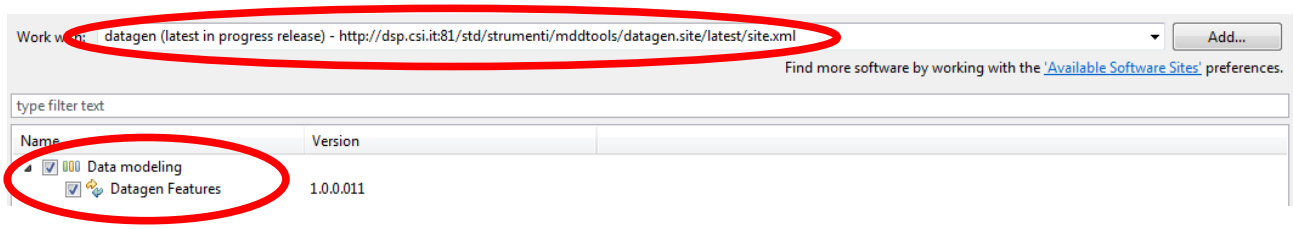
- *datagen v.1.0.0* (modellazione e generazione dello strato di accesso ai dati),
- *guigen v.1.5.0* (modellazione e generazione dello strato di user application),
- *mddtools v1.1.0* (documentazione e strumenti ausiliari).

I plugin sono disponibili nel formato dell'update-site eclipse. Le istruzioni di dettaglio per l'installazione dei plugin sono descritte nella sezione "installazione" della pagina disponibile all' URL:
http://dsp.csi.it/dsp/opencms/dsp/it/standard/add/MDD_materiale.html

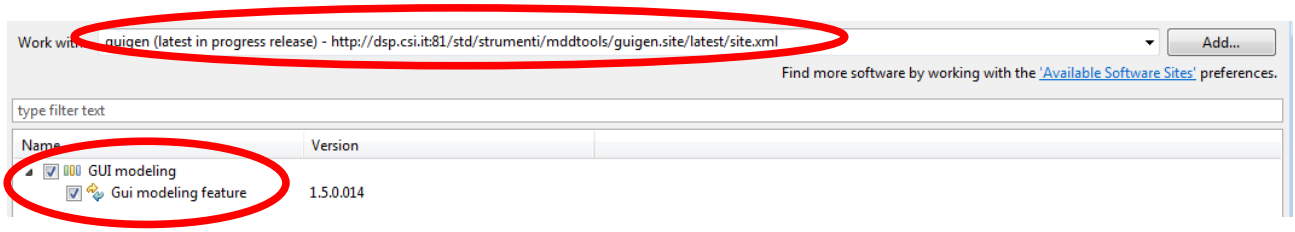
Alla stessa pagina (ma nella sezione "distribuzione dello strumento") sono riportati gli URL da utilizzare per l'impostazione degli update-site di download dei tool, con in aggiunta una importante indicazione della differenza tra gli update site di tipo "stable" (versione stabile rilasciata) e "latest" (l'ultima versione in sviluppo non ancora rilasciata ufficialmente).

In sintesi è necessario aggiungere ad eclipse i seguenti update-site ed istallare gli elementi evidenziati (attenzione!!! le versioni mostrate in questa guida possono differenziare da quelle effettivamente disponibili)

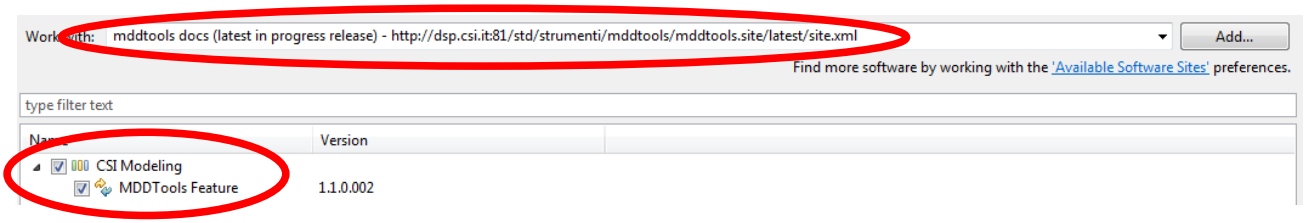
DATAGEN – tools per la generazione del layer di accesso ai dati



GUIGEN – tools per la generazione della componente di front-end applicativo

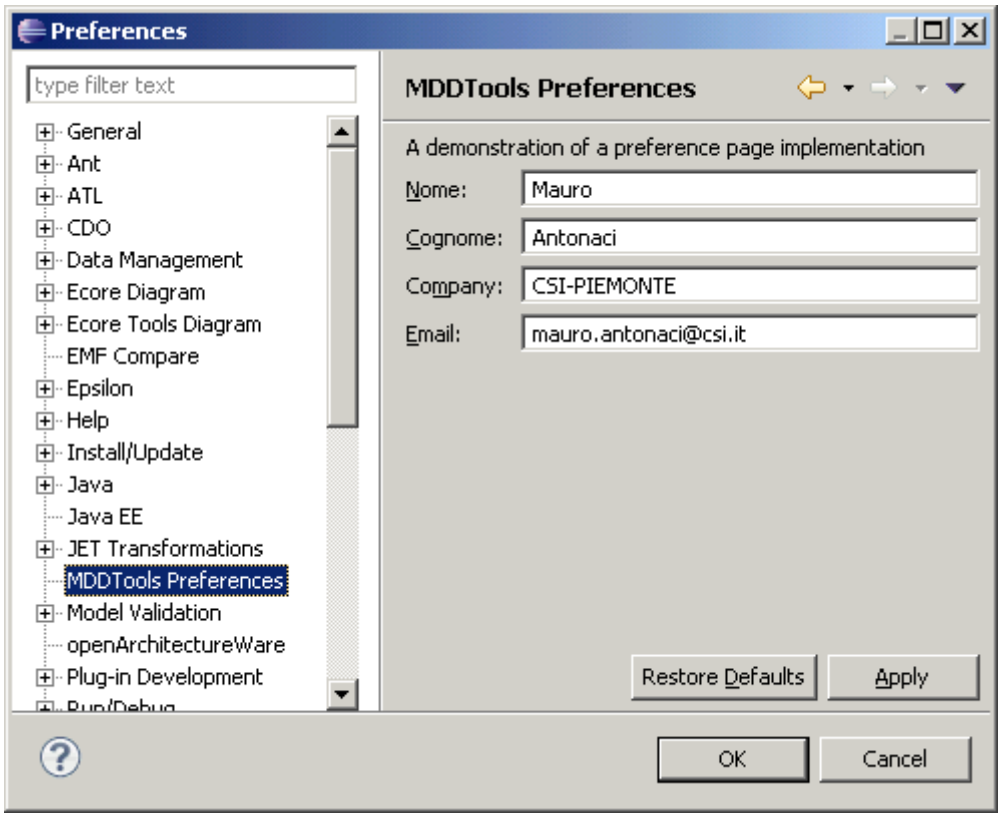


Documentazione dei tools MDD + funzionalità condivise



I tools per la generazione dei servizi non sono necessari per questa reference application per cui non ne è richiesta l'installazione.

La suite MDDTools contiene un sistema di raccolta di dati statistici sull'utilizzo dei tool, finalizzata al miglioramento dello strumento. Per tale motivo è necessario, al termine dell'installazione dei plugin, indicare i propri riferimenti nella finestra di configurazione del plugin MDDTools, accessibile tramite il menu window → preferences, e selezionando in seguito la voce “mddtools” nell'albero di navigazione.
All'interno della form è necessario inserire i propri riferimenti, che saranno utilizzati per la comunicazione diretta di novità inerenti gli strumenti mddtools.



2.2 Predisposizione degli ambienti di runtime

L'applicazione è destinata ad un ambiente di runtime costituito da:

- Application server Jboss 4.2 (è possibile, modificando la target platform sul modello guigen, rigenerare il progetto in modo che possa essere eseguito su Weblogic 9)
- DBMS Oracle 10

2.2.1 Predisposizione dell'application server

Si consiglia di utilizzare gli ambienti di sviluppo standard

2.2.2 Predisposizione del DBMS

Si consiglia di utilizzare gli ambienti di sviluppo standard. In aggiunta viene fornito uno script di creazione minimale del DB utilizzato dall'applicazione.

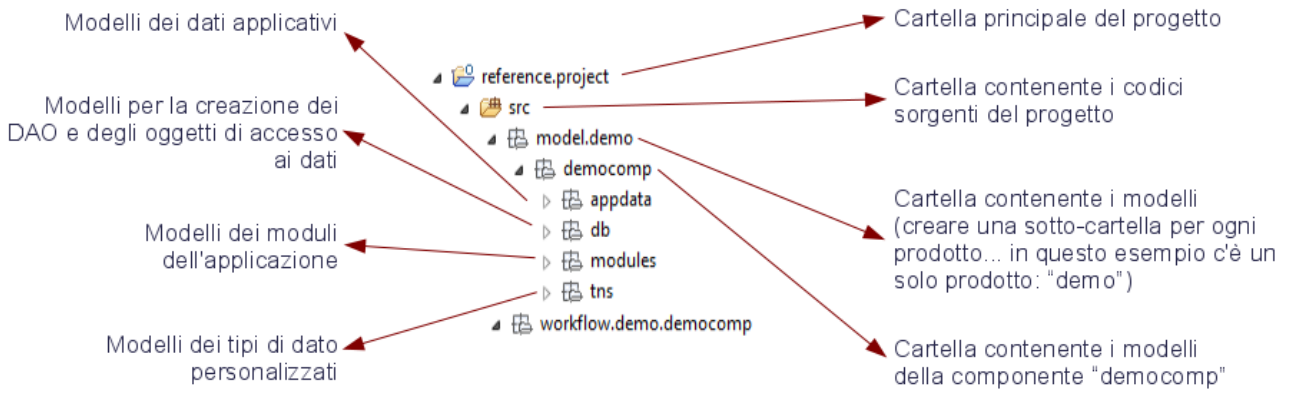
2.3 Creazione dell'alberatura di progetto

La distribuzione della reference application contiene due progetti:

- il progetto *generatore*
- il progetto *generato*

All'interno di Eclipse è necessario creare due progetti:

- Un progetto Java Standard che conterrà i codici sorgenti dell'applicazione (codice generato e codice scritto a mano). Tale progetto dovrà essere configurato per utilizzare l'alberatura standard per le applicazioni CSI (vedi documentazione standard). Questo progetto lo chiameremo **masterDetail**.
- Un progetto di tipo **OpenArchitecturware** che conterrà i modelli e i workflow per la generazione automatica del codice. Tale cartella dovrà essere configurata secondo quanto specificato nella documentazione MDD creando le cartelle che conterranno i modelli, i data type e i workflow di generazione del codice. Per la reference application la struttura delle cartelle da creare è la seguente:



3 Sviluppo della reference implementation tramite MDD

In questo capitolo verranno descritte le attività principali che permettono di ottenere l'applicazione funzionante sviluppata con gli strumenti MDD. Il focus del capitolo è quello di fornire al lettore una vista di insieme delle attività necessarie e della loro sequenza, senza addentrarsi nei dettagli di documentazione (per i quali si rimanda ai manuali di utilizzo). Verrà inoltre focalizzata l'attenzione del lettore su alcune scelte di modellazione evidenziando, ove necessario, quali altre alternative sarebbe stato possibile utilizzare e le motivazioni della scelta.

3.1 Overview delle attività necessarie

L'architettura applicativa finale dell'applicazione in oggetto (le cui specifiche sono state descritte al capitolo 1) è costituita da un unico componente software di tipo web application che accede direttamente al DB senza utilizzare nessuno strato di intermediazione di tipo SOA.

Nota: la scelta di **non utilizzare nessuno strato SOA** deriva essenzialmente dalle seguenti considerazioni:

1. le specifiche dell'applicazione non prevedono l'interazione con servizi esterni di terze parti
2. non vi è evidenza della necessità di costruire uno strato di servizi sui dati che possano essere utilizzati come back-end dell'applicazione in oggetto e messi anche a disposizione di altri potenziali fruitori.

In queste condizioni la presenza di uno strato SOA non è indispensabile. **Nei progetti reali è necessario tenere presenti questi criteri per decidere se utilizzare questa architettura semplificata o introdurre anche uno strato SOA.**

Gli artefatti che è necessario produrre per realizzare l'applicazione secondo l'architettura appena descritta sono:

1. sul progetto generatore:
   a) il modello dello schema del DB, da utilizzare nella modellazione dello strato di accesso ai dati (estensione: rdbmdl)
   b) il modello dello strato di accesso ai dati (estensione: datagen)
   c) i modelli dello strato di front-end applicativo (estensione: guigen)
2. sul progetto generato:
   a) il codice prodotto dal generatore guigen (web application J2EE, struts2 + spring)
   b) il codice prodotto dal generatore datagen (strato di bean spring che implementano i DAO e che si innestano nella struttura di classi java dell'applicazione generata da guigen)

Ad un primo livello di dettaglio le attività necessarie per lo sviluppo di un'applicazione che realizza le specifiche descritte in 1 con l'architettura di cui sopra possono essere riassunte come segue:

1. sul progetto generatore:
   a) modellazione dello **schema del DB**
   b) modellazione dello **strato di accesso ai dati**
   c) modellazione dell'**applicazione di front end**
2. sul progetto generato:
   a) implementazione della **business logic** e del "**collante**" tra front end ed accesso ai dati tramite codifica manuale.
   b) implementazione di **logica custom di accesso ai dati** ove necessario

Il processo di modellazione/sviluppo MDD è tipicamente un processo iterativo incrementale: non esiste dunque un ordine totale di esecuzione dei passi sopra elencati (ovvero ad es. non è detto che si debba prima completare l'attività di modellazione dell'accesso ai dati e successivamente quella di modellazione del front-end applicativo).
Esistono invece delle priorità/propedeuticità parziali: con questo termine si intende la necessità di effettuare parte di una specifica attività prima di poter realizzare parte di un'altra attività (es. prima di poter codificare manualmente la logica di business di una certa funzione utente, è necessario modellare, e poi generare, la porzione di applicazione relativa; prima di poter modellare la logica di accesso ad una determinata tabella è necessario modellare la struttura di tale tabella, etc..).

Nel contesto di quest'applicazione (che ha tra le caratteristiche di contorno quella di dover accedere ad un **DB preesistente** e non creato ex-novo) un **percorso efficace** (e che corrisponde con quello seguito per lo sviluppo effettivo di questa reference) potrebbe essere il seguente:

1. estrarre tramite **reverse engineering** il modello completo dello schema DB (operazione da compiersi *una-tantum* mediante un apposito wizard);
2. iniziare ad impostare il **modello del front-end applicativo**, iniziando a modellare **una schermata alla volta** e generando l'applicazione parziale;
3. man mano che, durante le attività di implementazione della logica di business di una funzione/schermata, si presenta la necessità di accedere ad una data tabella del DB, **modellare lo strato di accesso** a quella tabella e generare il codice corrispondente
4. implementare la logica applicativa inserendo nelle apposite regioni protette il codice che implementa la logica di business e che interagisce dove necessario con lo strato DAO;
5. ripetere i punti a partire dal (2) aggiungendo di volta in volta schermate/funzionalità o apportando correzioni.

Nota: come si può notare l'attività #1 dipende strettamente dal **contesto** di questo test-case, ovvero dalla **presenza pregressa di un DB**. Nel caso in cui il sistema prevedesse la realizzazione anche di un nuovo schema DB l'attività di reverse engineering non sarebbe più presente; al suo posto anche la modellazione dello schema DB potrebbe seguire, ad esempio, la stessa logica incrementale descritta per il front end. E' importante in definitiva notare come sia sempre necessario (e possibile) **adattare la sequenza con cui vengono svolte le varie attività di modellazione/sviluppo al contesto del progetto effettivo**.

Nei paragrafi successivi saranno descritte le singole attività. Si tenga presente durante tutta la lettura delle sezioni successive le osservazioni effettuate in questo paragrafo relativamente alla successione dell'esecuzione delle varie attività.

3.2 Implementazione del layer di accesso ai dati

Da un'analisi del DB esistente si evince che le tabelle a cui accedere, per l'implementazione delle funzioni oggetto dell'applicazione, sono:

- REFAPP_T_CITTADINO
- REFAPP_D_REGIONE
- REFAPP_D_PROVINCIA
- REFAPP_D_COMUNE

In questo contesto le attività da compiere per la creazione del layer di accesso ai dati sono:

1. effettuare il reverse engineering della base dati al fine di ottenere il modello dello schema relazionale (*rdbmdl*)
2. Creazione del modello dello strato di accesso ai dati (*datagen*)
3. Generazione del codice di accesso ai dati tramite il generatore DATAGEN
4. Personalizzazione del codice generato (ove necessario)
5. Creazione dei *manager* ad uso delle componenti di business dell'applicazione

3.2.1 Reverse engineering della base dati

La modellazione dello strato di accesso ai dati è costituita da due modelli:

- il modello dello schema relazionale da accedere (.rdbml)
- il modello dell'accesso vero e proprio a tali dati (.datagen)

Il modello dello schema relazionale a cui accedere può essere realizzato in due modi differenti:

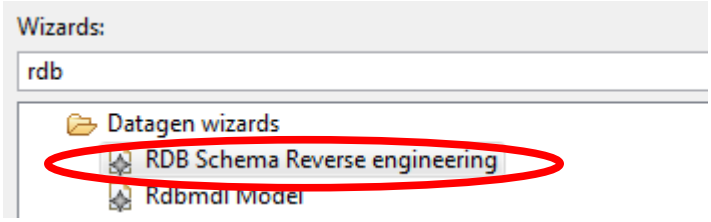
- creazione del modello da zero tramite il wizard **Rdbml Model** e successiva progettazione/modellazione manuale dell'intero schema[2];
- esecuzione del reverse engineering di un database esistente;

Nota: Per la reference application, esistendo già un database di riferimento da utilizzare come repository dei dati, l'unica scelta possibile è la seconda (reverse engineering).

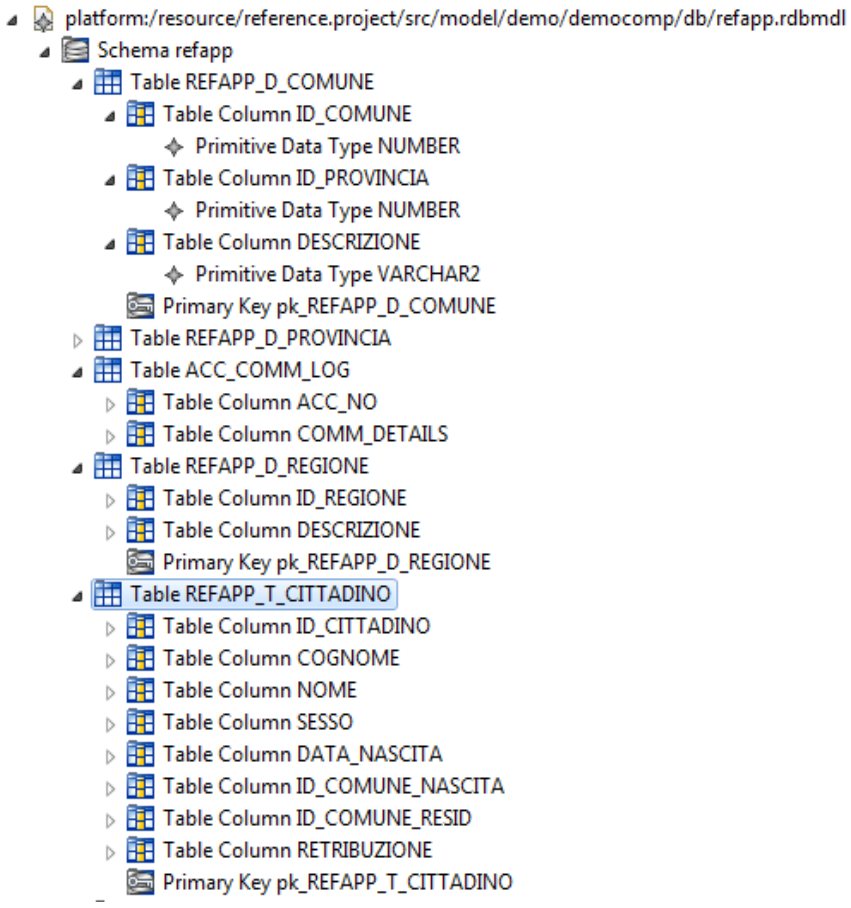
2 In futuro sarà possibile generare gli script SQL di creazione dello schema a partire da questo modello

Le operazioni da eseguire sono le seguenti:

1 – Selezionare da menu: File/New/Other
2 – selezionare il wizard “RDB Schema Reverse Engineering” (presente nella sezione “datagen wizards”)



3 – specificare la cartella di destinazione del modello (es. /reference.project/src/model/demo/democomp/db)
4 – specificare come nome del file: es. refapp.rdbmdl
5 – specificare come nome dello schema (nell'esempio il codice da utilizzare è: REFAPP)
6 – inserire la connessione JDBC ed eventuali user e password
7 – al termine del reverse engineering, all'interno della cartella db, sarà presente il modello relazionale del database, secondo il metamodello *rdbmdl*. Nella figura seguente è mostrata una porzione del modello, relativa alle tabelle da utilizzare nell'applicazione (non tutti i rami dell'albero sono stati esplosi per la visualizzazione; nella distribuzione è disponibile il modello completo):



Per la comprensione di dettaglio dei contenuti del metamodello si rimanda alla documentazione del metamodello distribuita con gli strumenti MDDTools.

3.2.2 Creazione del modello dello strato di accesso ai dati

La creazione del modello di accesso ai dati richiede di specificare, quali DAO (DataAccessObject) si desidera modellare e, per ogni DAO, quali strumenti (funzioni di accesso) si vogliono mettere a disposizione dell'utilizzatore.

Tipicamente nel modello saranno modellati solo i DAO corrispondenti alle tabelle interessate dalle funzionalità applicative.

Per ciascun DAO, poi, occorrerà modellare l'insieme degli strumenti da mettere a disposizione, che saranno da scegliere tra:

- **Inserter**: strumenti per l'inserimento di nuovi record nella tabella associata al DAO;
- **Finder**: strumenti per la ricerca e la lettura dati dalla tabella associata al DAO;
- **Updater**: strumenti per l'aggiornamento dei dati di record contenuti nella tabella associata al DAO;
- **Deleter**: strumenti per la cancellazione di record contenuti nella tabella associata al DAO.

NOTA: Non per tutti i DAO saranno necessari tutti gli strumenti e non è necessariamente una buona pratica definire a priori più strumenti di quelli effettivamente necessari.
Nell'esempio, per i DAO regione, provincie e comune saranno necessari solo i finders in quanto l'applicazione accederà a queste tabelle in sola lettura. Per il DAO cittadino, invece, saranno necessarie tutte le tipologie di strumenti in quanto si dovrà poter modificare i dati esistenti e e si dovrà poter inserire nuovi record.

Nei prossimi paragrafi saranno passati velocemente in rassegna i vari DAO.

3.2.2.1 DAO regione

Nel flusso di sviluppo incrementale descritto in precedenza la necessità di accedere ai dati della tabella delle regioni emerge analizzando la schermata principale, dove si evidenzia la necessità di caricare la combo-box regione. Da questa necessità emerge l'esigenza di modellare il DataAccessObject di nome “regione”, associato alla tabella delle regioni italiane.

L'analisi della logica di utilizzo dei dati gestiti da questo DAO evidenzia inoltre che l'unico utilizzo possibile è quello del caricamento totale di tutti i record della tabella Regioni. Questa esigenza si concretizza nella necessità di mettere a disposizione degli utilizzatori del DAO (e quindi di modellare) un particolare tipo di finder: quello di tipo “Find All” che è previsto dal metamodello.

Nota: lo strumento ***findAll*** è un tipico esempio di strumento da mettere a disposizione **solo quando effettivamente necessario**, per motivi che risultano abbastanza ovvi nel momento in cui la tabella di riferimento ha una cardinalità di dati molto alta. Che senso ha fornire un *findAll* in un DAO che gestisce i cittadini del Piemonte? E, soprattutto, che problemi di runtime può creare un utilizzo scorretto di tale strumento? In queste condizioni è preferibile non mettere a disposizione lo strumento.

Per completezza (e perché tipicamente questa necessità è molto ricorrente nella tipologia di applicazioni in oggetto) si può modellare anche il *finder* di tipo “Find By Pk” che, ricercando per chiave primaria, può

essere utilizzato per recuperare, in caso di necessità, il nome della regione dato il suo codice (ad esempio per assolvere ad esigenze di reportistica nelle quali il solo codice della regione non risulti sufficiente).

Nota: è facile rendersi conto che il momento in cui il modellatore decide di inserire un finder **può variare molto a seconda del flusso logico con cui vengono svolte le varie attività**: può essere adottata una strategia più ***proattiva*** che tende a modellare in un certo istante un numero di elementi maggiori rispetto a quelli di cui si sia manifestata l'evidenza sino a quel momento, oppure può essere adottata una strategia più ***reattiva,*** che invece tende a rimandare la modellazione di elementi fino al momento in cui se ne manifesta l'evidenza effettiva. Non esiste una strategia vincente in assoluto: entrambe sono percorribili e portano al risultato ottimale, anche se per percorsi differenti; potrebbe però risultare maggiormente adeguata l'adozione dell'una piuttosto che dell'altra a seconda, ad esempio, di elementi organizzativi: se ad esempio l'organizzazione del gruppo di lavoro prevedesse una suddivisione dei compiti tale per cui chi si occupa della parte di front-end e chi si occupa della parte di accesso ai dati sono persone distinte, allora potrebbe essere più naturale un approccio *proattivo;* in caso contrario (ovvero una suddivisione non per strati ma per funzionalità) risulterebbe più adeguato un approccio *reattivo*. La scelta va valutata attentamente caso per caso considerando i pro e contro delle due modalità, così come avviene peraltro anche negli sviluppi tradizionali.

3.2.2.2 DAO provincia

La necessità di accedere ai dati delle province italiane (e la conseguente necessità di modellare il DAO relativo) si evidenzia durante l'analisi della funzionalità di dettaglio/modifica dei dati del cittadino:

- è necessario recuperare l'insieme di tutte le provincie italiane (indipendentemente dalla regione) per il caricamento delle combo-box "provincia nascita" e "provincia residenza" nella schermata di dettaglio del cittadino. Per recuperare tali informazioni è sufficiente modellare il finder di tipo "Find All";
- è necessario recuperare tutte le province appartenenti ad una data regione per il caricamento della combo-box "provincia" nella schermata "Home". Per tale funzionalità è necessario modellare un finder di tipo custom (*customFinder*)che chiameremo "findByReg", in grado di restituire l'elenco delle provincie dato l'identificativo della regione.

Dopo la generazione del codice sarà necessario inserire nella classe del DAO "provincie", e più precisamente nel metodo java relativo a tale finder (che si chiamerà proprio *findByReg)*, la query da eseguire (vedi paragrafi successivi).

Per completezza si può inserire anche il finder "Find By Pk" che, ricercando per chiave primaria, può essere utilizzato per recuperare, in caso di necessità, il nome della provincia dato il suo codice.

3.2.2.3 DAO comune

L'esigenza di recuperare l'elenco di tutti i comuni appartenenti ad una data provincia si evidenzia nella schermata di dettaglio del cittadino, ove tali dati sono necessari per il caricamento delle combo-box "comune nascita" e "comune residenza". Per mettere a disposizione degli utilizzatori del DAO tale funzionalità è necessario modellare un finder di tipo custom (*customFinder*), che chiameremo "findByProv" in grado di restituire l'elenco dei comuni dato l'identificativo della provincia.

Dopo la generazione del codice sarà necessario inserire nella classe del DAO "comuni", e più precisamente nel metodo java relativo a tale finder (che si chiamerà proprio *findByProv)*, la query da eseguire (vedi paragrafi successivi).

Per completezza si può inserire anche il finder "Find By Pk" che, ricercando per chiave primaria, può essere utilizzato per recuperare, in caso di necessità, il nome del comune dato il suo codice.

Nota: è opportuno non modellare il metodo *findAll* in quanto l'elenco dei comuni italiani ha una cardinalità > 8000.

3.2.2.4 DAO cittadino

Il cittadino è l'oggetto principale del business di questa applicazione demo: è pertanto scontata la presenza di un DAO associato a tale archivio.

Dall'analisi delle funzionalità emergono le seguente esigenze di accesso alla tabella dei cittadini:

- nella funzionalità di ricerca emerge l'esigenza di effettuare l'estrazione dei record che soddisfano i criteri di selezione impostati nella schermata "home". Per assolvere a questa esigenza è necessario modellare un finder di tipo custom, che chiameremo "findByLoc" che eseguirà la query secondo i parametri di ricerca specificati
- nella funzionalità di dettaglio emerge l'esigenza di estrarre un singolo cittadino in base alla chiave primaria specificata. Per assolvere a questa esigenza è sufficiente modellare un finder che opera per chiave primaria (*findByPK*).

- Dalla funzionalità di inserimento di un nuovo cittadino discende la necessità di modellare un *inserter;*
- Dalla funzionalità di editing di un cittadino discende la necessità di modellare un *updater*. Tra gli *updater* disponibili si è scelto di utilizzare "Update Row" che consente di fare l'update di un intero record.
- Dalla funzionalità di cancellazione fisica di un cittadino dagli archivi discende la necessità di modellare un *deleter*. Si è scelto di utilizzare il deleter standard "Delete by PK" che consente di eliminare un record data la sua chiave primaria.

Nota: essendo l'entità "cittadino" l'oggetto centrale del business di questa applicazione di esempio, ed essendo questa una applicazione *gestionale*, il set di strumenti che emergono come necessari è quello di tipico di una funzionalità di CRUD, ovvero:

- C: un inserter
- R: una serie di finder (dipendenti dai dati dell'entità in questione)
- U: una serie di updater (come minimo un update complessivo, con l'aggiunta di ulteriori updater di un sottoinsieme di colonne se necessario)
- D: un deleter per PK

Nel caso dei dati di localizzazione amministrativa (regioni/provincie/comuni) invece si tratta di un tipico caso di entità di decodifica gerarchiche che presentano un differente ma altrettanto ricorrente pattern di accesso:

- *findAll* della tabella superiore nella gerarchia (o eventualmente customFinder nel caso in cui i dati di tale tabella siano di cardinalità ingestibile cognitivamente e tecnicamente)
- *customFinder* di una tabella di livello *n+1,* finalizzato a risolvere un ramo della gerarchia dal livello *n* al livello *n+1*, fissato l'elemento del livello superiore (es. comuni data la provincia) con la possibile variante dell'aggiunta di un filtro ulteriore nel caso in cui la cardinalità del solo vincolo gerarchico sia ingestibile cognitivamente o tecnicamente

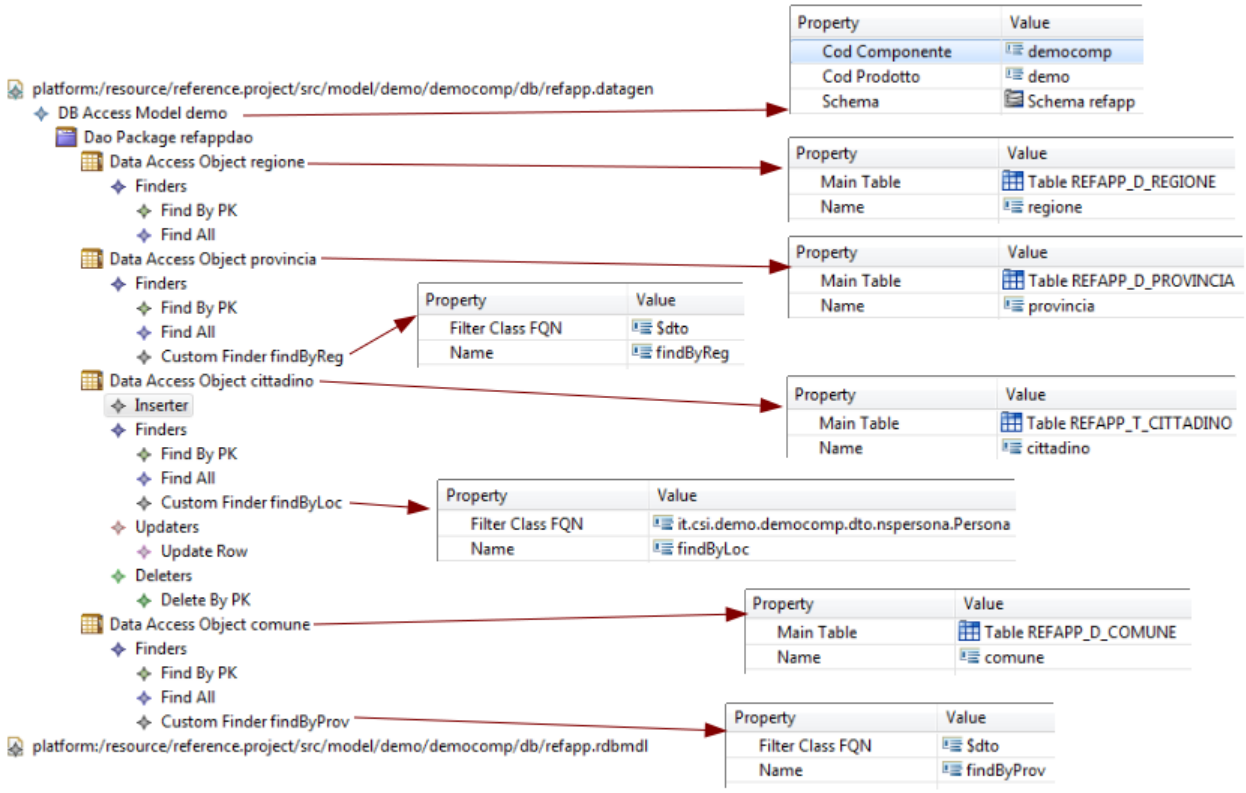
- findByPK per ciascun livello della gerarchia, tipicamente utilizzato per la risoluzione di relazioni tra oggetti di business e questi oggetti di decodifica (lookup; esempio cittadino → regione di residenza)

Riconoscere, durante l'analisi delle funzionalità, a quale categoria appartenga una certa entità fornisce una utile guida alla definizione *proattiva* degli strumenti che il corrispondente DAO dovrà mettere a disposizione.

E' altresì opportuno far notare che modellare il metodo *findAll* per questo DAO sarebbe una follia progettuale in quanto l'elenco dei cittadini italiani ha una cardinalità assolutamente ingestibile in caso di estrazione totale (oltre a non essere assolutamente sensata una potenziale funzionalità on-line che preveda l'estrazione dell'elenco completo di cittadini.

3.2.2.5 Il modello

Il modello viene istanziato tramite l'utilizzo del wizard di eclipse "new data gen model". Per l'utilizzo di questo wizard e per le specifiche di dettaglio della struttura di questo tipo di modello (meta-modello) vedere la documentazione contenuta nella distribuzione dei tools MDD. La figura seguente mostra il modello DAO relativo all'esempio (il modello completo è contenuto nella distribuzione della reference application):



3.2.3 Generazione del codice tramite DATAGEN

Una volta modellato (almeno un sottoinsieme iniziale dello) strato di accesso ai dati è possibile generare il codice che implementa tale layer. Per far ciò è necessario predisporre quello che lo strumento *openarchitectureware – OAW* definisce "workflow di trasformazione":

- nella cartella workflow.demo.democomp creare un nuovo file chiamato db.oaw e inserire al suo interno il contenuto della figura seguente (per il significato delle singole voci vedere la documentazione MDD disponibile nell'help on-line di eclipse (sezione MDDTools user's guide → datagen → cartridges):

```
<?xml version="1.0"?>
<workflow>
<!-- Clean de build directory -->
    <component class="org.openarchitectureware.workflow.common.DirectoryCleaner">
        <directory value="masterDetail/build" />
    </component>
    <cartridge file="it/csi/mddtools/datagen/workflow/basicDao.oaw"
      model="reference.project/src/model/demo/democomp/db/refapp.datagen"
      daoTargetProjectName="masterDetail"
      basePackage="it.csi.demo.democomp.business"
      daoBeansFileName="/conf/web/democomp/WEB-INF/dao-beans.xml"
    />
</workflow>
```

- selezionare il file **db.oaw** e specificare il comando "Run As / oAW Workflow". Il processo di generazione del codice verrà eseguito e, se non ci sono errori, nella cartella del progetto (masterDetail) troveremo il codice dei DAO appena creati e dei relativi DTO.

Nota: è necessario inserire tra le *dependencies* del progetto generatore anche il plugin *org.openarchitectureware.dependencies*. Il progetto di esempio è già configurato correttamente in tal senso.

3.2.4 Personalizzazione del codice dei DAO generato

Il codice che implementa la logica di accesso ai dati generato da *datagen* è parzialmente completo: ciò significa che in molti casi non è necessario intervenire manualmente, mentre in alcuni casi ben determinati è necessario intervenire in tal senso per completare l'implementazione.
I casi in cui non è necessario intervenire manualmente sono:

- il metodo *inserter*
- il metodo *findByPK*
- il metodo *findAll*
- il metodo *findByExample (qbe finder)*
- il metodo *updateRow*
- il metodo *deleteByPk*

Negli altri casi è necessario intervenire manualmente. Tipicamente l'intervento è finalizzato a definire/perfezionare lo statement SQL.
Nel caso della reference application è necessario procedere alla personalizzazione del codice in modo da implementare i custom finder modellati. Nel nostro caso sarà necessario modificare i seguenti file, presenti nella cartella /masterDetail/src/java/it/csi/demo/democomp/business/business/dao/refappdao/dao/impl

- ComuneDaoImpl.java → metodo findFindByProv()
- ProvinciaDaoImpl.java → metodo findFindByReg()
- CittadinoDaoImpl.java → metodo findFindByLoc()

le modifiche devono avvenire solamente all'interno delle **regioni protette** evidenziate nel codice. Qualsiasi modifica apportata al di fuori di esse verrà persa a fronte di una qualsiasi rigenerazione del codice.

Il codice relativo alle modifiche apportate è presente nel file ZIP contenente il progetto. Di seguito, come esempio, riportiamo uno screenshot del codice del metodo findFindByReg() del DAO Provincia.

```
public List<ProvinciaDto> findFindByReg(
                it.csi.demo.democomp.business.business.dao.refappdao.dto.ProvinciaDto input)
                throws ProvinciaDaoException {

        /*PROTECTED REGION ID(R371461634) ENABLED START*/
        // personalizzare la query SQL relativa al finder
        String sql = "SELECT * from REFAPP_D_PROVINCIA WHERE ";

        // personalizzare l'elenco dei parametri da passare al jdbctemplate (devono corrispondere in tipo e
        // numero ai parametri definiti nella querystring)
        sql += "ID_REGIONE = :idregione";
        //***aggiungere tutte le condizioni
        Map<String, Object> paramMap = new TreeMap<String, Object>();
        paramMap.put("idregione", input.getIdRegione());

        /*PROTECTED REGION END*/
```

3.3 Implementazione del front end applicativo

La user interface è stata progettata secondo quanto descritto nel capitolo 1 e modellata secondo il meta-modello *guigen*. In questo capitolo non verrà fornita una trattazione esaustiva di tale meta-modello (che è però disponibile nella documentazione di riferimento distribuita con i tool), ma saranno evidenziate alcune scelte eseguite nell'implementazione della reference application.

Le attività tipicamente necessarie per la realizzazione dell'applicazione sono:

- definizione dei tipi di dato strutturati necessari per l'applicazione (*ComplexType*)
- definizione dei dati applicativi da utilizzare a livello applicativo (*ApplicationData*)
- definizione del modello della user interface (*AppModule*, *ContentPanel*, ...)
- generazione del codice a partire dal modello
- personalizzazione del codice per la realizzazione della business logic

3.3.1 Definizione del modello della user interface

Per definire la user interface di un'applicazione modellata con *guigen* è necessario:

- individuare le differenti schermate che costituiscono l'applicazione (ogni schermata corrisponderà, nel modello, ad un oggetto di tipo *ContentPanel*); ciascuna schermata ha lo scopo di presentare all'utente un insieme di componenti di User Interface atti ad interagire con esso limitatamente all'insieme di funzionalità che quella schermata è destinata a realizzare;
- per ogni schermata:
  - definire la struttura gerarchica della user interface in termini di:
    - pannelli e sotto-pannelli

        - widget
    - associare ove necessario ai widget di controllo (es. pulsanti) la gestione degli eventi di user interaction e la logica che deve essere eseguita quando si scatena quell'evento. Questa modellazione si compone di:
        - *EventHandler* specifici di un determinato evento (es. click su pulsante, selezione di un elemento di una combobox, selezione di un tab, …)
        - Struttura di comandi che devono essere eseguiti a fronte di quell'evento. I comandi sono di vario genere:
            - comandi di variazione dello stato della user interface (accensione/spegnimento abilitazione /disabilitazione di widget, salto ad una differente schermata, refresh di una porzione di schermata, …)
            - comandi di esecuzione di logica di business (richiamo di un metodo java che esegue la business logic, gestione dello stato delle variabili applicative – es. backup, verifica variazioni, ripristino valore precedente, …)
            - comandi di flusso (es. esecuzione sequenza di comandi, esecuzione condizionale, …)
        - definire il binding tra componenti di user interface e variabili applicative *(Widget →* ApplicationData tramite *ApplicationDataBinding – vedere sezioni successive)*

Per una panoramica relativa ai "mattoncini" di base con cui è possibile definire la user interface si consulti la sezione apposita della guida di *guigen* distribuita con i tool (nell'help on-line di eclipse, sezione mddtools → guigen → UI Patterns).

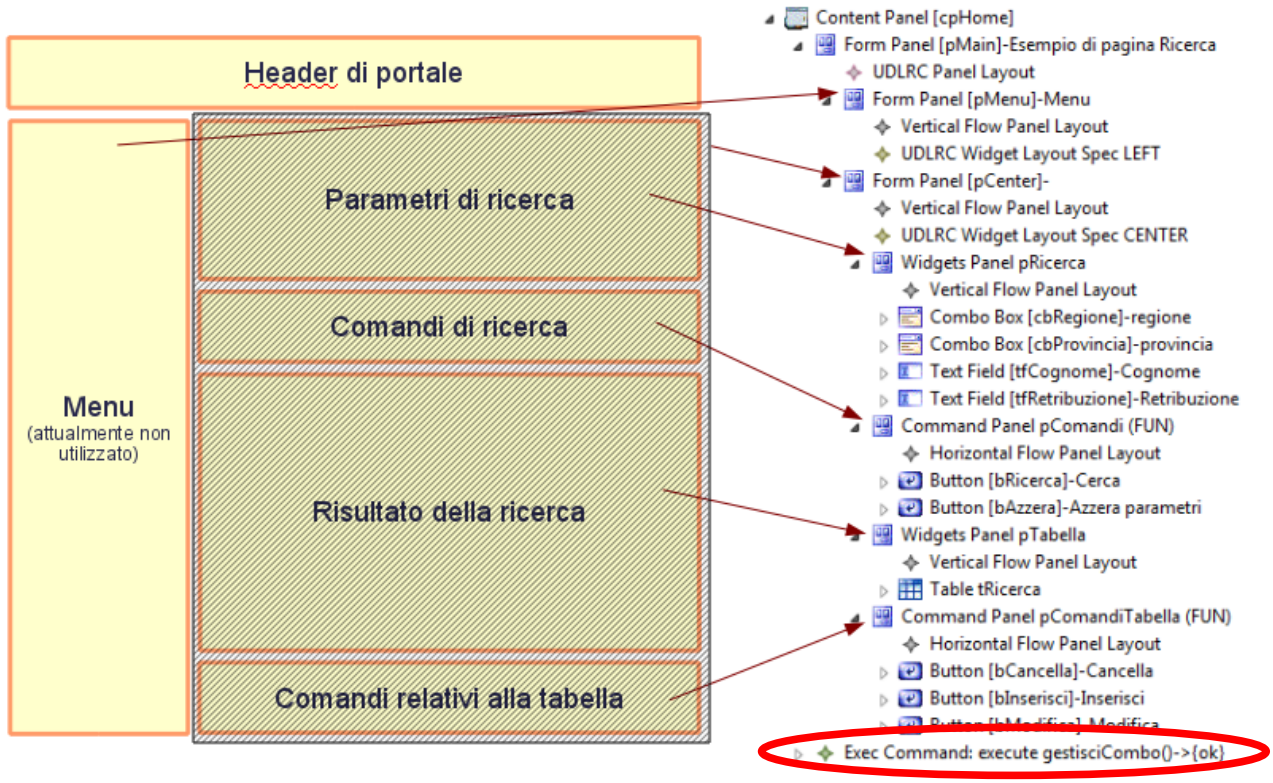
Per una descrizione approfondita degli elementi si consulti la guida di riferimento del meta-modello di guigen distribuita con i tool (nell'help on-line di eclipse, sezione mddtools → guigen → guida di riferimento metamodello).

Per assolvere alle specifiche descritte al capitolo 1 è stato necessario modellare due *ContentPanel*:

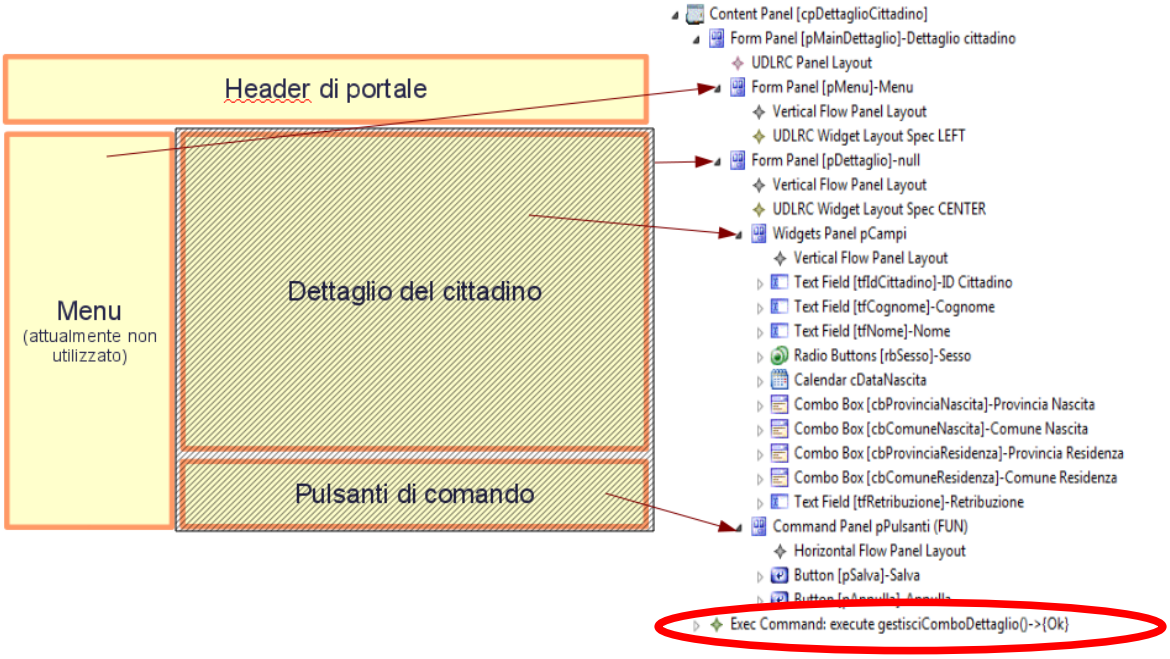
- *cpHome*: contiene la schermata home con le funzioni di ricerca
- *cpDettaglioCittadino*: contiene la schermata di dettaglio del cittadino con le funzioni di visualizzazione e salvataggio dei dati anagrafici

Per entrambi i *ContentPanel* si è utilizzato il layout a quadranti (UDLRC) per impostare le aree descritte nel capitolo 1

Il modello della schermata Home è visualizzato nella immagine seguente.



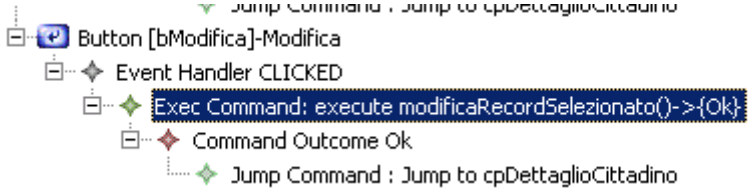
Il modello della schermata di dettaglio è invece rappresentato nella figura seguente:



3.3.2 Definizione della logica applicativa

Una volta definita la struttura della logica applicativa è necessario agganciare la logica applicativa agli eventi di user interaction e agli eventi impliciti (refresh della schermata, inizializzazione dell'applicazione).
Nel caso degli eventi espliciti è necessario modellare, in corrispondenza del W*idget* che recepisce l'evento dall'utente, un *EventHandler* specifico del tipo di evento che si vuole catturare (l'elenco di eventi disponibili dipende dal particolare *Widget*).
In entrambi i casi è successivamente necessario modellare la sequenza di comandi che devono essere eseguiti a fronte dell'evento (esplicito od implicito).

Nella figura seguente è mostrato un esempio di modellazione di evento esplicito e relativi comandi (



Tale frammento di modello definisce che, al verificarsi di un evento di "click" sul pulsante "bModifica" della schermata di dettaglio cittadino, deve essere eseguito il metodo di business logic "modificaRecordSelezionato()" che ha il compito di rendere persistenti le modifiche al cittadino che l'utente ha "comandato" agendo sui vari *Widget* della schermata stessa.
La logica vera e propria dovrà essere codificata a mano in linguaggio java come descritto nel paragrafo 3.3.6.

Nota: questo è un esempio molto semplice (ma anche molto frequente) di struttura di comandi. La complessità delle strutture di comandi definibili è molto superiore e può virtualmente prevedere una qualsiasi composizione di comandi atomici (purché sia coerente con le specifiche del meta-modello). Ad esempio è possibile modellare una logica che preveda:

- l'esecuzione di un metodo di business (*ExecCommand*)
- a seconda dell'esito esegui logiche differenti:
  - se l'esito è OK, esegui una sequenza di operazioni così costituita (*SequenceCommand*):
    - rendi abilitato un certo sottoinsieme dei *Widget* della schermata (*ONOFFCommand);*
    - rendi visibile un certo altro sottoinsieme di *Widget* della schermata (*VisibilityCommand*);
    - resta nella stessa schermata e restituisci il controllo all'utente (*NOPCommand*);
  - se l'esito è KO, resta nella stessa schermata visualizzando i messaggi di errore che la logica applicativa ha inserito nel contesto (*NOPCommand*)

Per una trattazione completa dei comandi disponibili si rimanda alla documentazione eclipse distribuita con i tool MDD.

In questo punto è ancora utile notare che il comando rappresentato in figura prevede (ottimisticamente) un solo *CommandOutcome* (esito) a fronte del quale viene eseguito il passo successivo che consiste nel passaggio alla schermata "home", tramite il comando *JumpCommand*.

Un altro frequente punto di modellazione dei comandi è a fonte dell'evento implicito di *refresh* della schermata, che viene scatenato ad ogni (ri)caricamento della pagina (nel caso in cui il refresh sia conseguente ad un evento esplicito la logica di *refresh* viene eseguita prima della logica associata all'evento). Tipicamente questo è il punto in cui vengono gestite alcune problematiche ricorrenti:

- il caricamento di strutture a supporto della schermata (es. caricamento delle collezioni di dati che alimentano le *ComboBox* presenti nella schermata, anche nel caso in cui vi siano delle logiche di dipendenza tra i vari widget)
- la configurazione dinamica della schermata (es. abilitazione/disabilitazione, visibilità/invisibilità di widget)

Nell'esempio è possibile notare nelle figure che descrivono il modello delle due schermate la presenza di un elemento di tipo *ExecCommand* che è proprio destinato a gestire la valorizzazione delle *ComboBox* presenti nelle due schermate.

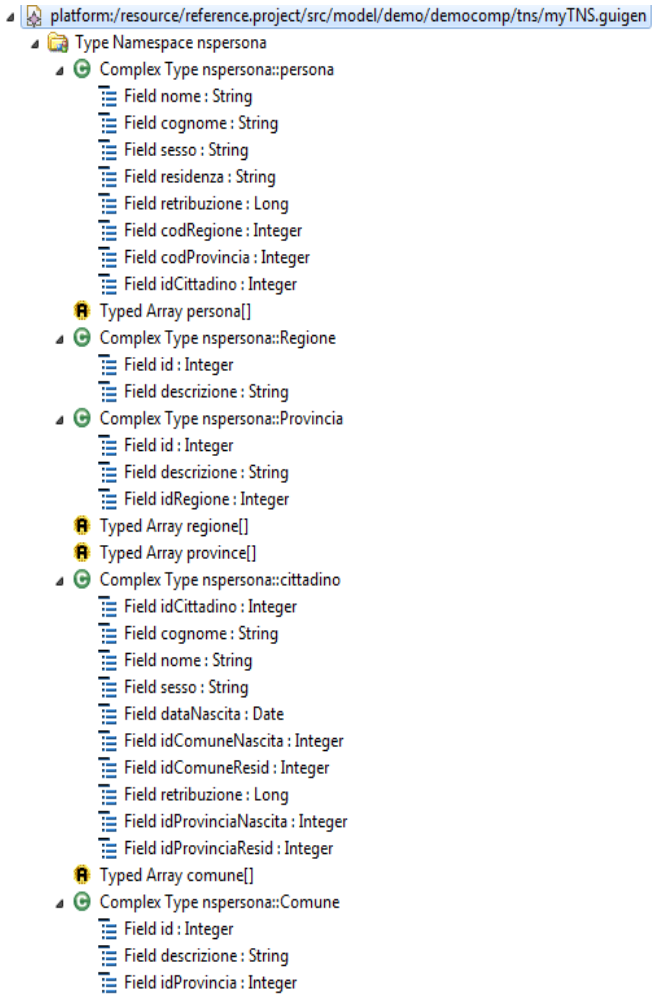
3.3.3 Definizione dei tipi di dato strutturati

Lo stato applicativo delle applicazioni modellate con *guigen* è mantenuto in contenitori assimilabili a variabili applicative (detti *ApplicationData*) che possono contenere dati di tipo semplice o strutturato (*ComplexType*). I tipi semplici sono nativamente a disposizione del modellatore, mentre i tipi complessi sono strettamente dipendenti dall'applicazione da realizzare e sono dunque oggetto di modellazione.
I tipi di dato personalizzati sono organizzati in partizioni denominate *TypeNamespace* (concetto assimilabile a quello di package). Ogni *TypeNamespace* deve essere modellato in un apposito file ed essere referenziato laddove serva (un tipo particolare di *TypeNamsepace* è il namespace “common” contenente i tipi base: il modello che descrive questo namespace è disponibile come libreria e deve essere referenziato ove sia necessario).
Per convenzione i namespace sono inseriti nella cartella /reference.project/src/model/demo/democomp/tns dove è stato creato il file myTNS.guigen che contiene il namespace di nome “nspersona”.

Nota: per motivi di praticità, vista la piccola dimensione della reference application, si è scelto di creare un unico file per i TNS personalizzati, che si va ad aggiungere al file del TNS “common”. In applicazioni reali è consigliabile la suddivisione dei TNS personalizzati in più file ognuno dei quali contenente TNS correlati fra loro. E' altresì verosimile che, con il diffondersi della metodologia, si diffonda la pratica del riuso in un'applicazione di strutture dato modellate al di fuori dello scope del progetto in corso di sviluppo: ad esempio potrebbe essere utile condividere strutture dato associabili a servizi trasversali (toponomastica?) oppure frammenti di modelli abilitanti all'integrazione con piattaforme (es. Flux/BPM).

Nell'esempio è stato creato il namespace **nspersona** atto a contenere i tipi di dato per la gestione del cittadino e dei dati anagrafici correlati.

Il modello (disponibile nella sua totalità nella distribuzione della reference application) è visualizzato nella immagine seguente.



Nota: i *ComplexType* rappresentano strutture dati di supporto alla gestione dello stato applicativo. E' importante specificare meglio in questa sede la loro collocazione all'interno dei vari strati (UI, business-logic, data-logic): la loro collocazione logica preferenziale è a cavallo tra lo strato di UI e lo strato di business logic.
Rappresentano, in pratica, il punto di contatto tra elementi di interfaccia grafica (schermate, pannelli, widget, eventi) e la logica di business.
La conseguenza di questa connotazione è che la struttura di tali oggetti modellati è principalmente finalizzata ad accogliere le informazioni oggetto di input/output tramite i componenti grafici di interfaccia. Sarebbe un errore progettuale progettare dei *ComplexType* molto somiglianti a entità di business, principalmente perché diventerebbe molto complicato, in quel caso, modellare il collegamento (*binding*) tra dati e user interface. Ecco, ad esempio, la motivazione della presenza nel tipo “persona” dell' id del comune di residenza e non di una più intuitiva proprietà complessa di tipo “comune”. Quanto avvicinarsi più o meno ad uno dei due fronti (UI o business logic) va valutato caso per caso, ma tenendo sempre presente come criterio guida quello della semplicità/fattibilità del binding con la user interface.

3.3.4 Definizione dei dati applicativi

I dati applicativi (*ApplicationData*) rappresentano i contenitori dei vari elementi informativi che, a runtime, conterranno lo stato dell'applicazione. Nell'astrazione fornita dal meta-modello *guigen* possono essere considerati a tutti gli effetti assimilabili all'insieme delle *variabili* applicative.
Sono tipicamente di tre tipologie corrispondenti al tempo di vita del dato in essi contenuto:

- scope USER_REQUEST: il valore è disponibile solo per una singola azione dell'utente
- scope SESSION: il valore è mantenuto per tutta la durata della sessione applicativa dell'utente
- scope PAGE: il valore è mantenuto per più azioni utente, ma solo finché il flusso applicativo resta sulla stessa schermata (*ContentPanel*).

Gli *ApplicationData* sono alimentati dallo strato di user interface al quale sono collegati tramite *ApplicationDataBinding* (da Widget a ApplicationData) e il loro valore può essere utilizzato/modificato nella business logic (solo le informazioni contenute negli application data sono passate allo strato di business logic).

Gli *ApplicationData* definiti per la reference application e il loro significato sono descritti nell'immagine seguente:

particolare attenzione va posta ai seguenti APPDATA:

- *selTabellaIdPersona*: viene utilizzato per memorizzare l'identificativo del cittadino selezionato nella tabella dei risultati della schermata home. Viene utilizzato nelle operazioni di delete del record selezionato (gestita dalla schermata home) e viene passato alla schermata di dettaglio per la visualizzazione dei dati del cittadino selezionato.
- *selettoreOperazione*: è una stringa che serve per parametrizzare la schermata di dettaglio del cittadino in modo da poter condividere il modello della schermata fra più operazioni. I valori consentiti sono:
  - UPD: indica che si tratta di un'operazione di visualizzazione/update. In tal caso, alla visualizzazione della schermata di dettaglio, devono essere precaricati i dati del cittadino selezionato. Alla pressione del pulsante “SALVA” sulla schermata di dettaglio verrà invocato il metodo di update sul manager GestorePersona.
  - INS: indica che si tratta di un'operazione di creazione di un nuovo record. La schermata di dettaglio viene visualizzata vuota. Alla pressione del pulsante “SALVA” sulla schermata di dettaglio verrà invocato il metodo di insert sul manager GestorePersona.

3.3.5 Generazione del codice

Per la generazione del codice è necessario creare un opportuno workflow nella cartella /reference.project/src/workflow/demo/democomp

Il file creato per la reference application si chiama newWorkflowFile.oaw e contiene i seguenti comandi (il significato degli stessi è presente nella documentazione ufficiale di MDD):

```
<?xml version="1.0"?>
    <workflow>

    <!-- test del modulo "home" -->
    <cartridge file="it/csi/mddtools/guigen/workflow/guigenCheck.oaw"
        model="reference.project/src/model/demo/democomp/modules/homepage.guigen"
        portal="neutral"
    />

    <!-- test del modello principale -->
    <cartridge file="it/csi/mddtools/guigen/workflow/guigenCheck.oaw"
        model="reference.project/src/model/demo/democomp/masterDetail.guigen"
        portal="neutral"
    />

    <!-- test del myAppdata Guigen -->
    <cartridge file="it/csi/mddtools/guigen/workflow/guigenCheck.oaw"
        model="reference.project/src/model/demo/democomp/appdata/myAppData.guigen"
        portal="neutral"
    />

    <!-- test del TNS Guigen -->
    <cartridge file="it/csi/mddtools/guigen/workflow/guigenCheck.oaw"
        model="reference.project/src/model/demo/democomp/tns/myTNS.guigen"
        portal="neutral"
    />

    <!-- generazione del codice -->
    <cartridge file="it/csi/mddtools/guigen/workflow/struts2Basic.oaw"
        model="reference.project/src/model/demo/democomp/masterDetail.guigen"
        targetProjectName="masterDetail"
        portal="neutral"
    />

</workflow>
```

Per la generazione del codice bisogna selezionare il file newWorkflowFile.oaw ed eseguire il comando “Run As / oAW Workflow”

Al termine della generazione del codice, nella cartella di progetto verrà costruito lo scheletro dell'applicazione con tutta la user interface e il codice di gestione del pattern MVC tramite struts2.

3.3.6 Personalizzazione del codice

La personalizzazione del codice deve avvenire nelle opportune regioni protette. Se questo non avviene, con le rigenerazioni successive del codice, verranno perse le modifiche effettuate.

Per ogni pannello definito nell'applicazione, viene generato un file denominato CPBE<nomepannello>.java all'interno inserire il codice di personalizzazione.

Per ogni comando inserito nel modello, all'interno di tali file, verrà predisposto un metodo che dovrà essere opportunamente personalizzato.

Per la reference application, tali file sono presenti nella cartella
/masterDetail/src/java/it/csi/demo/democomp/business/miaapp

Sono stati generati i seguenti due file con i relativi metodi:

- **CPBECpHome.java:** contiene il codice per la gestione della schermata (pannello) home. I metodi predisposti sono:
  - **eseguiRicerca():** contiene il codice per l'esecuzione di una ricerca in base ai parametri inseriti nei campi di ricerca
  - **azzeraCampiRicerca():** contiene il codice per azzerare i campi di ricerca e il risultato della tabella
  - **cancellaRecordSelezionato():** contiene il codice per eseguire la cancellazione del record selezionato sulla tabella
  - **visualizzaRecordVuoto():** il suo compito è visualizzare la schermata di dettaglio in bianco per l'inserimento di un nuovo record
  - **modificaRecordSelezionato():** contiene il record che visualizza la schermata di dettaglio del cittadino e che precarica i dati del record selezionato sulla tabella
  - **gestisciCombo():** contiene il codice per la gestione della sincronizzazione delle combo-box

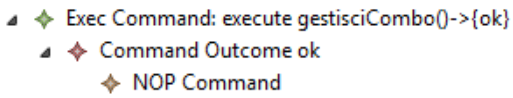
- **CPBECpDettaglioCittadino.java:** contiene il codice per la gestione della schermata (pannello) di dettaglio del cittadino. I metodi predisposti sono:
  - **salvaDatiCittadino()**: esegue il salvataggio dei dati del cittadino. A seconda che si tratti di un update o di un nuovo inserimento richiamerà funzioni differenti sul manager GestorePersona.
  - **tornaIndietro()**: ritorna alla schermata home annullando l'operazione di inserimento/modifica. Questo metodo, attualmente, non richiede personalizzazioni
  - **gestisciComboDettaglio()**: gestisce la sincronizzazione delle combo-box della schermata di dettaglio

3.3.6.1 Esempio: sincronizzazione delle combo-box nella schermata home

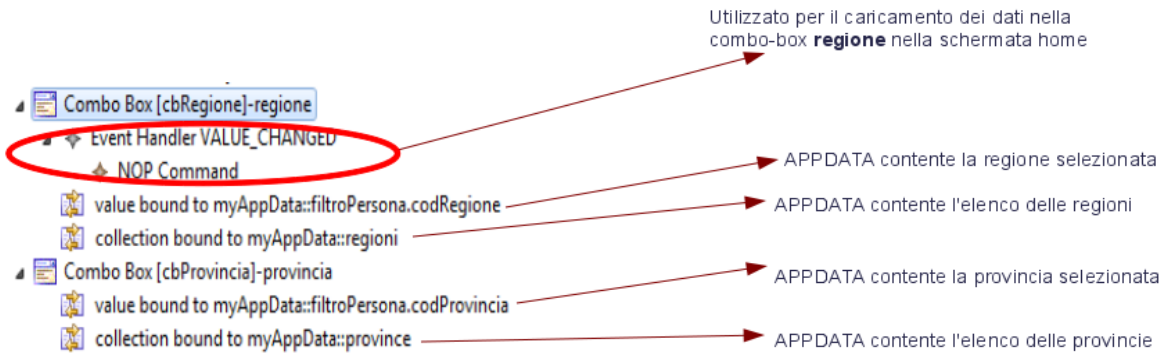
La sincronizzazione delle combo-box della schermata home richiede che, alla selezione di una regione dalla relativa combo-box, le provincie di tale regione, debbano essere caricate nella combo-box provincie.

Essendo state abilitate, come requisito, le feature ricche, è necessario indicare al sistema quando eseguire l'aggiornamento delle combo-box.

La prima cosa è aggiungere al pannello cpHome l'esecuzione del metodo gestisciCombo(). Questo verrà eseguito al caricamento del pannello



Quindi è necessario un handler nella combo-box regione, che specifica il refresh della pagina al cambiamento del calore sulla combo-box regione.



Il codice personalizzato, da inserire nel metodo gestisciCombo() è il seguente:

```
/*PROTECTED REGION ID(R348602115) ENABLED START*/
// inserire qui la logica applicativa da eseguire:
if (theModel.getAppDatafiltroPersona() == null) {
    theModel.setAppDatafiltroPersona(new Persona());
}

if (theModel.getAppDataregioni() == null) {
    theModel.setAppDataregioni(getGestoreLimitiAmm()
            .getAllRegioni());
    theModel.setAppDataprovince(new ArrayList<Provincia>());
} else {
    if (theModel.getAppDatafiltroPersona().getCodRegione() == 0) {
        theModel.setAppDataprovince(new ArrayList<Provincia>());
    } else {
        theModel.setAppDataprovince(getGestoreLimitiAmm()
                .getProvinceByRegione(
                        theModel.getAppDatafiltroPersona()
                                .getCodRegione()));
    }
}
// impostazione del result code
result.setResultCode("ok");
// modifica degli attributi del model (che verranno propagati allo strato
// di presentation). si puo' agire anche direttamente sull'attributo "session".

result.setModel(theModel);
return result;
/*PROTECTED REGION END*/
```

3.3.6.2 Esempio: riutilizzo della stessa schermata per il salvataggio e la modifica

Si è deciso di utilizzare la stessa schermata “Dettaglio Cittadino” sia per le operazioni di modifica che per le operazioni di inserimento di un nuovo record. A tale scopo si è utilizzato l'APPDATA selettoreOperazione che potrà contenere i seguenti valori:

- INS: indica che si stà eseguendo l'inserimento di un nuovo record
- UPD: indica che si stà eseguendo un update.

Tale appdata verrà impostato all'interno della schermata home, nel file **CPBECpHome.java** all'inerno dei due metodi **visualizzaRecordVuoto()** e **modificaRecordSelezionato().** Prima di visualizzare il dettaglio, il primo metodo dovrà pulire tutti gli appdata relativi al dettaglio del cittadino mentre il secondo dovrà precaricare i dati del cittadino selezionato.

Di seguito, come esempio, viene riportato il codice del metodo modificaRecordSelezionato()

```
/*PROTECTED REGION ID(R1283594808) ENABLED START*/
// inserire qui la logica applicativa da eseguire:
theModel.setAppDataselettoreOperazione("UPD");
// impostazione del result code
result.setResultCode("Ok");
// modifica degli attributi del model (che verranno propagati allo strato
// di presentation). si puo' agire anche direttamente sull'attributo "session".

Cittadino z = new Cittadino();
z = getGestorePersona().getCittadinoByPk(
        theModel.getAppDataselTabellaIdPersona());

//impostazione delle combo con i valori precedenti
theModel.setAppDataprovince(getGestoreLimitiAmm().getAllProvince());
theModel
        .setAppDataprovince2(getGestoreLimitiAmm().getAllProvince());
z.setIdProvinciaNascita(getGestoreLimitiAmm().getProvinciaByComune(
        z.getIdComuneNascita()));
z.setIdProvinciaResid(getGestoreLimitiAmm().getProvinciaByComune(
        z.getIdComuneResid()));
theModel.setAppDatacomuni(getGestoreLimitiAmm()
        .getComuniByProvincia(z.getIdProvinciaNascita()));
theModel.setAppDatacomuni2(getGestoreLimitiAmm()
        .getComuniByProvincia(z.getIdProvinciaResid()));
// fine impostazioni combo
theModel.setAppDatacittadino(z);
result.setModel(theModel);
return result;
/*PROTECTED REGION END*/
```

All'interno della schermata Dettaglio Cittadino, nel metodo **salvaDatiCittadino()**, all'interno del file **CPBECpDettaglioCittadino.java** verrà testato il valore dell'appdata e verrà eseguita l'operazione relativa.

```
/*PROTECTED REGION ID(R-1824915609) ENABLED START*/
// inserire qui la logica applicativa da eseguire:

if (theModel.getAppDatacittadino() != null) {
    Cittadino c = new Cittadino();
    c = theModel.getAppDatacittadino();
    if (theModel.getAppDataselettoreOperazione().equals("INS")) {
        getGestorePersona().salvaNuovoCittadino(c);
    }
    if (theModel.getAppDataselettoreOperazione().equals("UPD")) {
        getGestorePersona().modificaCittadino(c);
    }
    theModel.setAppDatapersone(getGestorePersona()
            .getElencoPersone(theModel.getAppDatafiltroPersona()));
}

// impostazione del result code
result.setResultCode("Ok");
// modifica degli attributi del model (che verranno propagati allo strato
// di presentation). si puo' agire anche direttamente sull'attributo "session".
result.setModel(theModel);
return result;
/*PROTECTED REGION END*/
```

3.4 Codifica manuale della business logic (strato business/manager)

3.4.1 Creazione dei manager

Per la gestione, a livello business, delle entità in gioco, è necessario creare degli opportuni manager, che consentono di parlare, verso i DAO con i DTO a livello dati (generati da DATAGEN) e verso la User Interface, con i DTO di presentation (generati da GUIGEN)

un'ottima posizione per i manager è la cartella /masterDetail/src/java/it/csi/demo/democomp/business/mgr

per la reference application sono stati implementati due manager:

- **GestoreLimitiAmm.java**: si occupa di gestire il business relativo alle entità Regione, Provincia e Comuni.
- **GestorePersona.java**: si occupa di gestire il business relativo all'entità Cittadino

I sorgenti dei due manager sono disponibili nello zip dell'applicazione.

I manager sono implementati come componenti Spring che vengono iniettati nell'applicazione. È quindi necessario configurare il file **applicationContext.xml**, presente nella cartella /masterDetail/conf/web/democomp/WEB-INF in modo da indicare a Spring l'esistenza dei due manager.

La modifica di tale file è consentita solo all'interno delle opportune regioni protette. La figura seguente mostra la configurazione utilizzata per la reference application

```
<!--PROTECTED REGION ID(R585863764) ENABLED START-->
    <!-- Inserire qui configurazioni aggiuntive da iniettare nel bean prinicipale -->
<bean id="GestoreLimitiAmm" class="it.csi.demo.democomp.business.mgr.GestoreLimitiAmm">
    <property name="regioneDao">
        <ref bean="regioneDao"/>
    </property>
    <property name="provinciaDao">
        <ref bean="provinciaDao"/>
    </property>
    <property name="comuneDao">
        <ref bean="comuneDao"/>
    </property>
</bean>

<bean id="GestorePersona" class="it.csi.demo.democomp.business.mgr.GestorePersona">
    <property name="cittadinoDao">
        <ref bean="cittadinoDao"/>
    </property>
    <property name="comuneDao">
        <ref bean="comuneDao"/>
    </property>
</bean>


<!--PROTECTED REGION END-->
```

3.4.2 Punti di attenzione: gestione delle transazioni

Il framework MDD non gestisce direttamente le transazioni. Queste devono essere abilitate in modo manuale dallo sviluppatore. La transazionalità viene gestita tramite il framework Spring. Per poterla utilizzare è necessario aggiungere al file dao-beans.xml le seguenti righe:

```
<tx:annotation-driven transaction-manager="txManager" proxy-target-class="true" />
<bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

la configurazione precedente è per application server Jboss. Se si vuole utilizzare WebLogic sostituire la seconda linea con la seguente:

```
<bean id="txManager" class="org.springframework.transaction.jta.WebLogicJtaTransactionManager"/>
```

A questo punto è necessario annotare con @Transactional le operazioni che devono essere sottoposte a transazionalità.

All'interno della reference application le uniche operazioni transazionali sono presenti all'interno del manager GestorePersona.java e sono rappresentate dai metodi:

- salvaNuovoCittadino()
- cancellaCittadino()
- modificaCittadino()