



UNIVERSITA' DEGLI STUDI ROMA TRE

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

Manuale

Spring Rich Client

a cura di

Marco Piolo

Anno Accademico 2006-2007

Progetto di Sistemi Informatici

Indice

1	Introduzione	1
1.1	Introduzione	1
2	Il background	1
2.1	Le applicazioni desktop	1
2.1.1	I componenti e la gestione degli eventi	1
2.2	Il framework Spring	5
2.2.1	Layers Overview	5
2.2.2	Inversion Of Control	7
3	Il framework Spring Rich Client	14
3.1	Introduzione	14
3.1.1	Obiettivi	14
3.1.2	Architettura del framework	15
3.2	Componenti dell'applicazione	16
3.3	Command	18
3.4	Form	19
4	Una applicazione con Spring Rich Client	22
4.1	Introduzione	22
4.2	I Contesti Applicativi	22
4.2.1	ApplicationCtx	24

4.2.2	CommandsCtx	31
Bibliografia		35

Capitolo 1

Introduzione

1.1 Introduzione

Quella che segue è una introduzione a Spring Rich Client, un framework per la gestione di Interfacce Grafiche Utente(GUI) basato sulle metodologie del framework Spring, di cui verrà fatto in seguito un breve compendio.

Questo manuale è stato prodotto nell'ambito del corso di Progetto di Sistemi Informatici sotto la supervisione del prof. Paolo Merialdo.

Il manuale è organizzato in 4 parti, la prima(questa) in cui c'è una breve introduzione a quelli che saranno i temi trattati, la seconda che descrive il contesto e il background in cui è stato sviluppato il framework, la terza in cui vengono introdotte le metodologie di Spring Rich Client e l'ultima in cui viene mostrata una piccola guida pratica per iniziare a progettare con il framework descritto.

Quello che è descritto in questo manuale è riferito alla versione 0.2.1 del framework e alla versione 2.0 di Spring.

Capitolo 2

Il background

2.1 Le applicazioni desktop

2.1.1 I componenti e la gestione degli eventi

Prima di passare a dettagliare le caratteristiche del framework è bene dare una breve introduzione a quelle che sono le caratteristiche di una applicazione desktop, visto che parte delle metodologie del framework servono per gestire al meglio tali caratteristiche.

Una interfaccia grafica di una applicazione è, di norma, basata su:

componenti un certo numero di oggetti grafici, detti appunto componenti; ad esempio bottoni etichette, immagini e quant'altro

contenitori un certo numero di componenti, utilizzati per raggruppare altri componenti; ad esempio finestre pannelli etc...

oggetti di supporto una serie di altri oggetti di supporto come gestori del layout e gestori degli eventi

Per quanto riguarda la tecnologia Java, il supporto alla grafica è dato dalle tecnologie AWT e SWING, che sono implementate dai package `java.awt` e `javax.swing`. Il primo esempio di classi Java per gestire GUI è AWT la cui

prima release si ebbe nel 1995. Oltre ad avere poche funzionalità per quanto riguarda l'aspetto grafico(un esempio su tutti, quello di non poter creare una struttura ad albero), AWT presentava anche una forte dipendenza dalla piattaforma su cui venivano fatte girare le applicazioni. Con l'avvento di J2SE 1.2, gli sviluppatori Java rilasciarono il toolkit SWING, che aggiungeva alle caratteristiche di AWT numerosi Widget, un supporto *plugin-like* per gestire il *look-and-feel*¹ e maggior supporto per rendere il tutto meno dipendente dalle piattaforme.

Con riferimento alla tecnologia Swing, una interfaccia grafica è di norma composta da un *contenitore principale*, che può essere una applet (`javax.swing.JApplet`), una frame (`javax.swing.JFrame`) o una finestra di dialogo (`javax.swing.JDialog`); un *pannello* (`javax.swing.JPanel`) che è l'unico elemento contenuto nel contenitore principale ed è quello destinato a fare da contenitore intermedio per altri componenti; una serie di *elementi atomici*, che vengono effettivamente visualizzati nell' interfaccia grafica come bottoni (`javax.swing.JButton`), etichette (`JLabel`), campi di testo (`JTextField`) etc ...

Se da una parte l'utente può accedere, in maniera visiva, ai dati presentati dall'applicazione, dall'altra esso può richiedere, con gli strumenti di input della piattaforma che sta utilizzando, dati e informazioni all'applicazione.

Nelle interfacce grafiche, le azioni intraprese dagli utenti sono rappresentati da eventi: un *evento* rappresenta un interazione tra l'utente e l'interfaccia grafica. Quando si verifica un evento, l'applicazione (leggi JVM) crea l'oggetto associato all'evento,cerca il responsabile della gestione dell'evento e fa eseguire a quest'ultimo l'azione associata all'evento; quest'ultima azione dipende ovviamente dal tipo di evento che è stato generato.

Il componente grafico che ha generato l'evento (ad esempio un `JButton`) è chiamato *sorgente* dell'evento, e l'oggetto che è responsabile della gestione

¹per Look And Feel si intende il posizionamento dei componenti all'interno dei contenitori e la gestione della colorazione dell'applicazione

dell'evento stesso è chiamato *listener*.

In definitiva per quanto riguarda la gestione degli eventi sono di interesse tre oggetti:

1. l'oggetto che rappresenta l'evento che si è verificato; gli eventi sono rappresentati da oggetti polimorfi di tipo `EventObject`
2. la sorgente dell'evento, che è un componente dell'interfaccia grafica
3. l'ascoltatore dell'evento; gli ascoltatori di eventi sono oggetti che implementano l'interfaccia `EventListener`

Gli oggetti ascoltatori vanno realizzati definendo classi che implementano opportune interfacce, relative alle tipologie di eventi di interesse. La classe per un ascoltatore contiene la definizione di un metodo per ciascuna tipologia di evento che può verificarsi, che descrive le azioni che vanno eseguite in corrispondenza del verificarsi di tale evento. Ovviamente è possibile che una stessa classe sia responsabile della gestione di più tipologie di eventi, ossia che implementi più interfacce, e questo è proprio il caso più frequente.

Quello che segue è un brevissimo esempio basato su SWING: si ha una finestra con al suo interno un pannello che contiene un bottone; il pannello è gestore dell'evento associato allo schiacciamento del bottone stesso ed è responsabile di cambiare la stringa di testo del bottone.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class EsSwing8 {
    public static void main(String [] v){
        JFrame f = new JFrame("Esempio"); //creazione frame
        Container c = f.getContentPane();
        EsPanel p = new EsPanel(); //creazione pannello
        c.add(p); //aggiunta componenti
        f.pack();
        f.show(); //la finestra viene mostrata
    }
}
```

```
} } }
```

Questo è il main dell'applicazione, in esso viene creata la frame, il pannello, il quale viene successivamente aggiunto alla finestra e il tutto viene poi mostrato all'utente.

```
// Il codice del pannello
public class EsPanel extends JPanel implements ActionListener{
    private JLabel l;

    public EsPanel(){
        super();
        l = new JLabel("Tizio");
        add(l);
        JButton b = new JButton("Tizio/Caio");
// Tizio/Caio è l'etichetta del pulsante
        b.addActionListener(this); //associazione all'evento
// registra lo stesso oggetto panel come
// ascoltatore degli eventi
        add(b); }
    public void actionPerformed(ActionEvent e){
        if(l.getText().equals("Tizio"))
            l.setText("Caio");
        else
            l.setText("Tizio"); } }
```

Nel listato precedente c'è la specifica del pannello: nel costruttore c'è il richiamo alla superclasse, la valorizzazione dell'etichetta e la specifica del bottone.

Inoltre si noti il fatto che la classe implementa l'interfaccia ActionListener: questo fa sì che il pannello sia il gestore dell'evento associato al bottone;². Nel caso in questione il testo dell'etichetta viene alternato in *tizio* e *caio*.

²quest'associazione viene specificata con l'overriding del metodo `void addActionListener(ActionListener al)`

2.2 Il framework Spring

2.2.1 Layers Overview

Lo sviluppo del framework Spring Rich Client è basato fortemente su Spring, un altro framework ben più sviluppato e più utilizzato, di cui è nato come progetto parallelo per lo sviluppo di interfacce grafiche.

I maggiori benefici vengono dati soprattutto dalla Dependency Injection/Inversion Of Control che permette una leggera configurazione dell'applicazione e maggior resistenza ai cambiamenti, in linea con i principi della OOP.

Prima di trattare in maniera approfondita le caratteristiche di Spring Rich Client è bene dare un'idea generale di quali siano gli obiettivi che Spring (o meglio, gli sviluppatori di tale framework) vuole perseguire (e che in realtà ha perseguito):

- Eliminare alcune responsabilità dallo sviluppatore e portarle al framework
- Eliminare la proliferazione di singleton in una applicazione, in quanto essi ostacolano il testing
- Spostare, per quanto possibile, la configurazione dell'applicazione da numerosi file specifici in diversi formati alla configurazione naturale del framework, con l'aiuto dell'Inversion Of Control
- Promuovere l'uso di Best-Practice introducendo la programmazione verso interfacce, piuttosto che verso classi
- Mantenere una bassissima dipendenza dell'applicazione dalle API del framework stesso
- Fornire il supporto per numerose tecniche di accesso ai dati, tra i quali JDBC o supporto per altri framework ORM

Per ottenere questi scopi, il framework contiene varie funzioni e caratteristiche, che sono organizzate in sette moduli(package) la cui organizzazione è mostrata in figura:

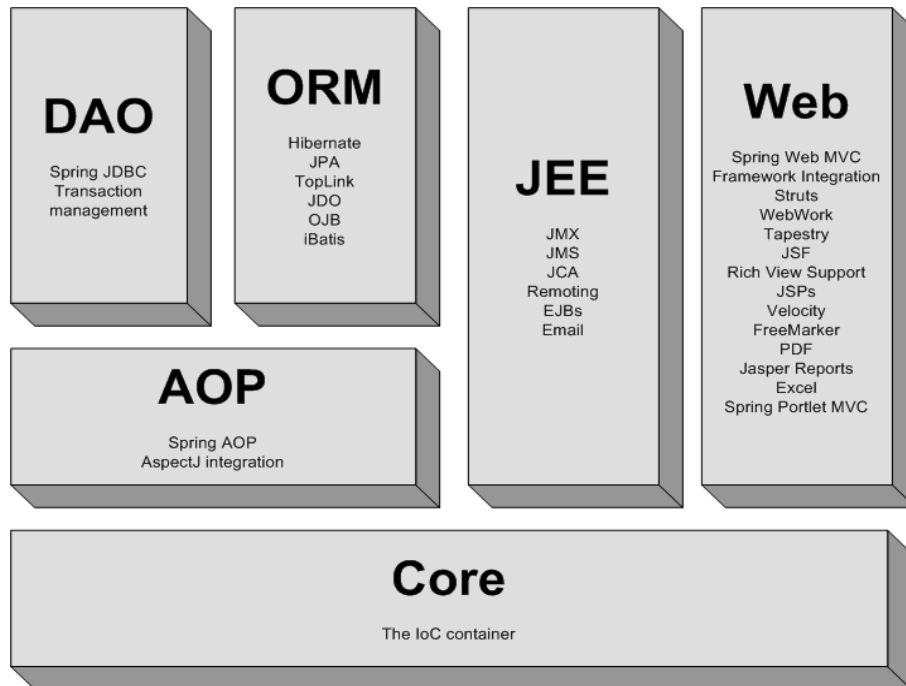


Figura 2.1: L'architettura a strati del framework spring

Il package **Core** è quello che contiene la parte centrale del framework, che offre le funzionalità di gestione dei bean³ tramite l'Inversione di Controllo (discusse in dettaglio nel prossimo paragrafo). Queste funzionalità sono offerte per la maggior parte dall'interfaccia `BeanFactory` che tramite un pattern Factory rimuove il bisogno di gestire manualmente oggetti singleton, e permette di disaccoppiare la configurazione delle dipendenze dalla logica applicativa dell'applicazione.

Il package **DAO** fornisce un livello di astrazione per JDBC e rimuove la necessità di scrivere codice ripetitivo e di gestire errori specifici dei diversi

³Un bean è un tipico oggetto java puro e semplice, caratterizzato da variabili d'istanza, e metodi getter/setter

vendor. Inoltre viene fornito il supporto per la gestione programmatica delle transazioni per tutti i POJO (*Plain Old Java Object*).

Il package **ORM** offre il supporto ai più diffusi ORM come Hibernate JDO e iBatis. Tutto questo viene fatto in maniera molto modulare e in combinazione con le altre funzionalità del framework.

Il package **AOP** di Spring offre il supporto all' AOP, che è utilizzata come complemento all' IOC e alla OOP per avere un maggiore disaccoppiamento del codice.

Per quanto riguarda la parte web del framework ci sono i due package *Web* e *Web MVC*.

Nel package **Web** troviamo funzionalità di integrazione orientate al web come ad esempio l'inizializzazione di contesti applicativi in maniera dichiarativa tramite *listener*. Inoltre viene fornito un ottimo framework MVC. Questo è il package da utilizzare quando si vuole integrare l'applicazione Spring ad esempio con Struts o con JSF.

Infine nel package **JEE** sono presenti funzioni di supporto per applicazioni di livello Enterprise come ad esempio funzioni di Remoting e supporto per EJB.

2.2.2 Inversion Of Control

Uno dei problemi maggiori che sta affliggendo il mondo a oggetti (questo ovviamente vale per tutti i tipi di applicazioni OO, anche per quelle con interfaccia web) è l'interazione tra diversi componenti durante lo sviluppo di applicazioni web. Come si può far interagire ad esempio una architettura di un controller web con l'interfaccia di un database, quando questi potrebbero essere stati sviluppati da team differenti?

Per risolvere problemi di questo tipo stanno emergendo nuove idee e correnti di pensiero, e nuovi framework sono nati per ovviare a queste prob-

lematiche. I framework nati per ovviare a queste problematiche vengono chiamati *Lightweight containers*.

Uno dei principi basilari che vengono utilizzati da questi *contenitori leggeri* è l'Inversione del Controllo. Il termine Inversione di Controllo ha però molteplici significati, in funzione di che cosa si intende per controllo. Il controllo può essere quello della creazione di oggetti Java, quello della gestione delle interfacce utente di una applicazione e molto altro...

In questo caso il controllo è riferito alla gestione delle dipendenze che gli oggetti Java hanno tra di loro.

Si potrebbe avere un caso di questo tipo:

```
class MovieLister {
    public Movie[] moviesDirectedBy(String myDir) {
        List allMovies = finder.findAll();
        for (Iterator it = allMovies.iterator(); it.hasNext();) {
            Movie movie = (Movie) it.next();
            if (!movie.getDirector().equals(myDir))
                it.remove();
        }
        return (Movie[]) allMovies.toArray(new Movie[allMovies.size()]);
    }
}
```

Nell'esempio in questione si ha una semplice classe *MovieLister* che offre il metodo *moviesDirectedBy(String myDir)* che restituisce l'array contenente tutti i film diretti da un particolare regista, in questo caso passato come parametro al metodo.

Il metodo in sé non fa niente di particolare, si fa ridare la lista dei film da un oggetto *finder*, itera su questa lista rimuovendo quelli che non sono diretti da quel particolare regista e restituisce l'array equivalente.

In realtà quello che interessa è l'oggetto *finder* o meglio, come la classe *MovieLister* si collega con tale oggetto. L'obiettivo è quello di aver il metodo *moviesDirectedBy(String arg)* completamente indipendente dal modo in cui i film vengano memorizzati. Si potrebbe astrarre il concetto di *finder* in una interfaccia *MovieFinder* che offra il metodo *findAll()* :

```
public interface MovieFinder {  
    List findAll();  
}
```

Adesso è tutto abbastanza ben disaccoppiato. Ad un certo punto però la classe `MovieLister` dovrà interagire per forza con l'oggetto finder concreto, quello che cercherà i film. In questo caso il finder viene passato, ad esempio, nel costruttore della classe `Lister`:

```
class MovieLister...  
    private MovieFinder finder;  
    public MovieLister() {  
        finder = new ColonDelimitedMovieFinder("movies1.txt");  
    }
```

Nel caso in questione il finder è un oggetto che cerca i film in un file di testo rappresentato dalla stringa che viene passata come parametro del costruttore del *ColonDelimitedMovieFinder*.

Finchè i film vengono memorizzati in un file di testo chiamato *movies1.txt* non sorgono problemi. Ma basti pensare all'evoluzione futura di questo piccolo componente software: se i film venissero memorizzati in un file chiamato in maniera diversa? oppure venissero memorizzati in file XML? o tramite un RDBMS? Si avrebbe sempre una dipendenza del `Lister` dall'apposita implementazione del `Finder`.

Si hanno cioè le dipendenze mostrate in figura:

Sarebbe desiderabile che il `MovieLister` lavorasse in maniera totalmente indipendente dall'implementazione del finder. Inoltre sarebbe comodo che la giusta implementazione venga aggiunta *a posteriori*, fuori dalle responsabilità del progettista, per garantire estrema flessibilità e modularità.

In una applicazione reale, potrebbero esserci decine di queste componenti, e quindi decine di componenti da astrarre in interfacce e da assemblare tra di loro.

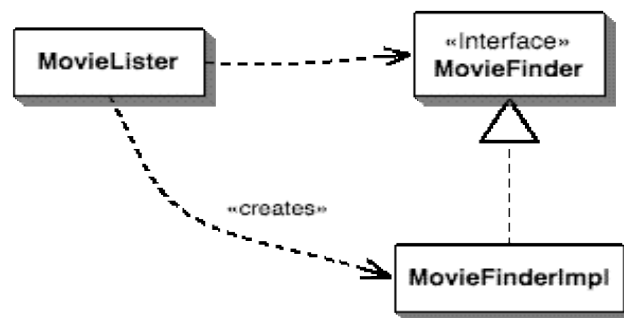


Figura 2.2: Le dipendenze che coesistono tra il Lister e il Finder

Il nucleo del problema è quindi come poter assemblare tutti questi *plug-in* in una applicazione reale.

Per risolvere questo problema, tutti i contenitori leggeri di nuova generazione stanno utilizzando Inversion Of Control.

L'idea di base è quella di avere un oggetto esterno a cui delegare la responsabilità di gestire il controllo del collegamento degli oggetti tra loro, eliminando quest'ultimo dalle stesse classi software.

L'approccio dei contenitori leggeri è fare in modo che l'utente di un *plug-in* segua qualche convenzione che permette all'oggetto esterno, di iniettare l'implementazione del *plugin* dove necessario.

Per questo motivo Martin Fowler in un suo articolo nel 2004 ha creato un pattern più specifico: **Dependency Injection** che segue la regola del *Hollywood Principle*: Don't call me, I'll call you.

L'obiettivo è avere le dipendenze mostrate in figura:

Per realizzare la Dependency Injection ci sono 3 modi:

- Constructor Injection: iniezione per costruttore
- Setter Injection: iniezione tramite metodi setter
- Interface Injection: iniezione tramite uso di metodi astratti, meno utilizzato e non supportato da Spring

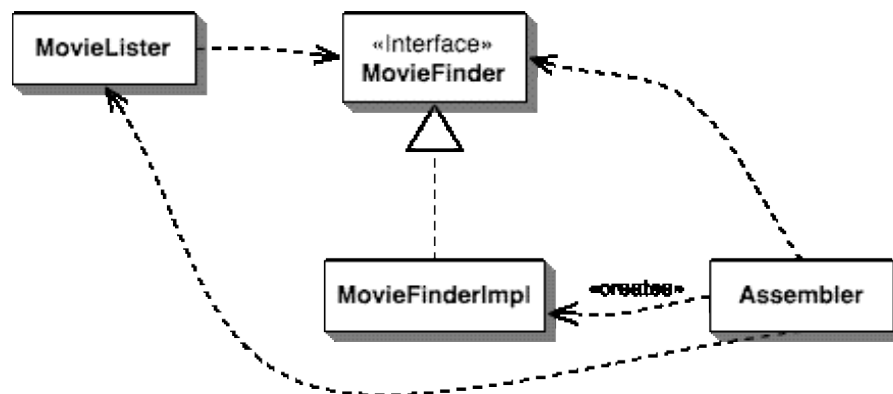


Figura 2.3: Le dipendenze dell'esempio utilizzando la Dependency Injection

Constructor Injection Per realizzare l'iniezione di dipendenza posso passare l'apposita implementazione del finder nel costruttore del lister e posso passare il nome del file nel costruttore del finder:

```

class MovieLister...
    private MovieFinder finder
    public MovieLister(MovieFinder finder){
        this.finder = finder;
    }

class ColonMovieFinder...
    private String fName;
    public ColonMovieFinder(String fName){
        this.fName = fName;
    }
  
```

La configurazione delle classi viene fatta attraverso files XML.

```

<beans> <bean id="MovieLister" class="spring.MovieLister">
  <constructor-arg>
    <ref bean="MovieFinder"/>
  </constructor-arg>
</bean> <bean id="MovieFinder" class="spring.ColonMovieFinder">
  <constructor-arg>
    <value>movies1.txt</value>
  </constructor-arg>
</bean> </beans>

```

Nel file XML si specifica che si stanno utilizzando due bean *MovieLister* e *MovieFinder* che come costruttori hanno rispettivamente il finder e una Stringa che rappresenta il nome del file.⁴

Setter Injection Per attuare la Dependency Injection posso farlo anche tramite metodi setter. E' sufficiente mettere un costruttore vuoto e dei metodi setter per i campi che si vuole modificare.

```

class MovieLister... private MovieFinder finder; public void
setFinder(MovieFinder finder) { this.finder = finder; } class
ColonMovieFinder... public void setFilename(String filename) {
  this.filename = filename;
}

```

Inoltre si va a modificare il file XML con la configurazione:

```

<beans> <bean id="MovieLister" class="spring.MovieLister">
  <property name="finder">
    <ref local="MovieFinder"/>
  </property>
</bean> <bean id="MovieFinder" class="spring.ColonMovieFinder">
  <property name="filename">
    <value>movies1.txt</value>
  </property>
</bean> </beans>

```

⁴di default i bean sono singleton, per non avere singleton è necessario impostare l'attributo singleton a false

In maniera simile alla Constructor Injection si specifica la configurazione dei bean.

Un test potrebbe essere il seguente:

```
public void testWithSpring() throws Exception { ApplicationContext
ctx =
    new FileSystemXmlApplicationContext("spring.xml");
MovieLister lister = (MovieLister) ctx.getBean("MovieLister");
Movie[] mov = lister.moviesDirectedBy("Sergio Leone");
assertEquals('Once upon a time in the west', mov[0].getTitle());
}
```

Come si è visto la configurazione avviene dentro files XML⁵ in maniera dichiarativa. E' presente ovviamente il supporto per specificare multioggetti, come file di Properties, Liste, Mappe e Set.

⁵è presente un DTD di riferimento su <http://www.springframework.org/dtd/spring-beans.dtd>

Capitolo 3

Il framework Spring Rich Client

3.1 Introduzione

3.1.1 Obiettivi

L'obiettivo del progetto *Spring-richclient* è di fornire una modalità di sviluppo per coloro che necessitano piattaforme e metodologie *best-practice* per costruire applicazioni Swing in maniera rapida.

In particolare il target è articolato come segue:

1. Fornire un modo elegante per progettare applicazioni Swing altamente configurabili e che siano conformi allo standard per le GUI, ispirandosi alle metodologie del framework Spring.
2. Favorire l'integrazione con progetti legati alle interfacce grafiche. Ad esempio, JGoodies e TableLayout sono progetti già validi, non si ha bisogno di crearne di nuovi.
3. Aderire ai principi di Spring e dell'OO, ossia programmare verso interfacce, documentare e testare.

4. Supporto per gestione di finestre multiple, configurazione delle viste, che possono essere definite nel contenitore e configurate per essere caricate all'avvio
5. Fornire supporto per la validazione degli input, barre degli strumenti, gestione delle tabelle e quant'altro ...
6. Fornire quel livello di astrazione che ad oggi manca nelle applicazioni Swing e che ne rende complesso lo sviluppo

Per perseguire tali obiettivi la strategia di Spring Rich Client è basarsi fortemente sulle idee e sulle metodologie di Spring, descritte in precedenza, che hanno dimostrato la loro validità, essendo Spring ad oggi uno dei framework maggiormente utilizzati per lo sviluppo di applicazioni.

3.1.2 Architettura del framework

Il framework Spring Rich Client è organizzato in 6 moduli dipendenti l'uno dall'altro come mostrato in figura:

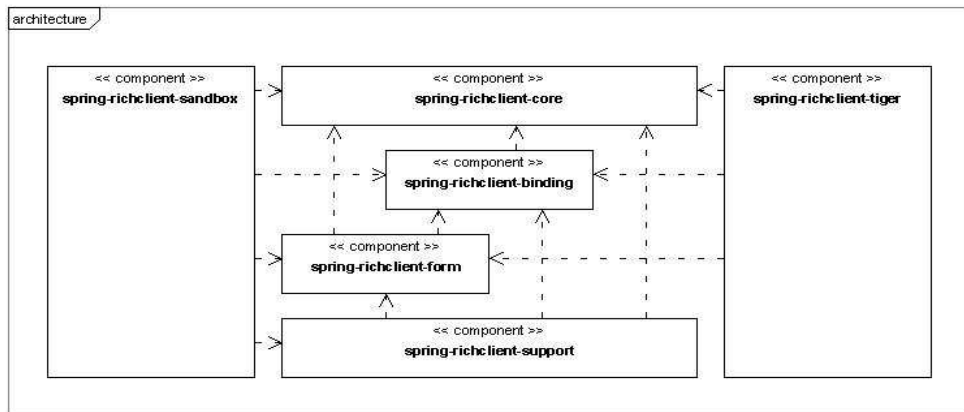


Figura 3.1: Architettura del framework

1. Il modulo **Sandbox** è utilizzato per le versioni ancora non definite del codice e quindi ancora in fase di test e di sviluppo.
2. Il modulo **Core** è il vero e proprio motore del framework ed è quello che utilizza le potenzialità di Spring Framework(Dependency Injection).
3. Il modulo **Binding** si occupa del binding, ossia di effettuare il legame tra gli oggetti del modello e quelli della vista.
4. Il modulo **Form** offre funzionalità e supporto per le form di input.
5. Il modulo **Support** fornisce un supporto *framework-wide*.
6. Il modulo **Tiger** serve per la gestione di alcune funzionalità di Java introdotte a partire dalla versione 5

Nell'applicazione questi moduli collaborano tra di loro in maniera non invadente e senza appesantire troppo l'applicativo per fornire allo sviluppatore quel livello di astrazione che manca nelle applicazioni basate su Swing e che proprio a causa di questa di questa mancanza sono complesse da sviluppare.

3.2 Componenti dell'applicazione

L'applicazione con Spring Rich Client(SRC) è così sviluppata:

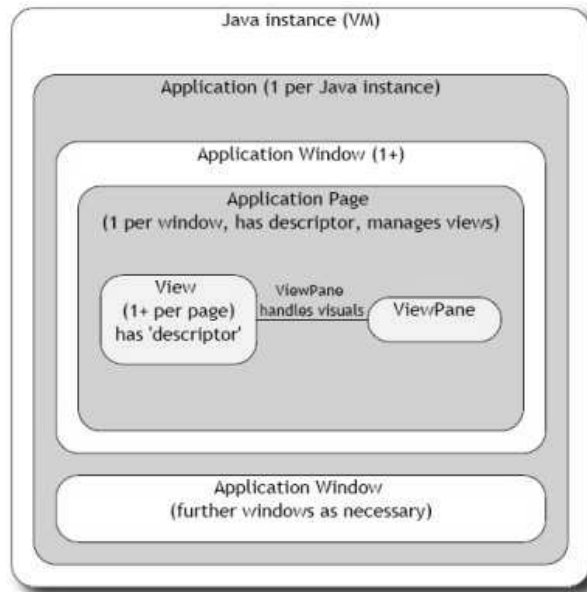


Figura 3.2: Organizzazione dell'applicazione

Si ha una applicazione per ogni JVM che gira sul calcolatore, ed ogni applicazione al suo interno ha (in generale) diverse finestre, implementate tramite componenti di tipo `JFrame`, la quale agisce come *Factory* per la creazione dei componenti che sono al suo interno.

Per ogni finestra dell'applicazione si ha una sezione chiamata *ApplicationPage*. Questa zona della finestra contiene al suo interno un singleton *ApplicationPageDescriptor* la cui responsabilità è quella di incapsulare delle regole sul modo di inizializzare il Layout per quella Page.

Ogni *ApplicationPage* contiene al suo interno delle viste (*View*) che sono le sezioni vere e proprie adibite al *docking*¹. Inoltre ogni *View* ha associato un proprio *ViewDescriptor* tramite il quale viene individuato il modo di rappresentare la vista all'interno della pagina. Le view possono anche essere aperte in diverse finestre e in diverse Page, l'istanza di una view rimane unica all'interno del suo *ViewDescriptor*.

¹per docking si intende la rappresentazione delle informazioni

Nella pratica la vista è rappresentata da una classe Java (che estende `AbstractView`) e in cui si rappresentano i dati tramite Swing nella maniera in cui si preferisce. Nelle stesse classi si implementano inoltre gli appositi Listener che dovranno essere utilizzati per le interazioni *Utente-Applicazione*. Le viste vengono configurate nel contesto dell'applicazione tramite files XML e iniettati tramite Dependency Injection a run-time.

Le viste possono essere posizionate in varie posizioni della pagina e impostate in strutture particolari: *stacked*, *tabbed* etc. . .

Inoltre la stessa vista può essere aperta in pagine diverse di finestre differenti, ma per ogni vista è ammesso soltanto un `ViewDescriptor`. Non si possono avere dunque due diversi layout per una stessa vista contemporaneamente.

3.3 Command

Nelle applicazioni Swing è necessario avere un Command Framework che aiuti l'utente nella creazione, gestione e configurazione per barre dei menu, degli strumenti, popup menu e tutta una serie di altre funzionalità che si usano nelle applicazioni rich-client. Spring Rich Client fornisce un proprio Command Framework (nel package `org.springframework.richclient.command`). Le principali caratteristiche di questo framework sono:

Actionable Commands Questi sono comandi che vengono generalmente utilizzati per implementare comandi locali, che hanno cioè senso soltanto in alcune viste dell'applicazione.

Targetable Commands In questo tipo di comandi il target del comando viene scelto dinamicamente, in base alla vista attualmente attiva. Si può pensare ad essi come a dei comandi globali condivisi da tutta l'applicazione.

Command Groups I comandi possono essere raggruppati insieme per essere usati in toolbar, menu a tendina etc

Command Configuration I comandi vengono configurati in maniera programmatica per quanto riguarda le icone, le etichette e tutto quello che riguarda il rendering del comando

Ulteriori Command Ulteriori funzionalità includono la tecnica dell'interception, parametrizzazione dei comandi e aggiornamento automatico dei comandi Swing in risposta ad un determinato comando

Come quasi tutto in Spring(e derivati) la definizione di questi comandi avviene nel contesto, anzi più precisamente nei contesti applicativi:

```
<bean id=" lifecycleAdvisor "
      class="com . mypackage . SimpleLifecycleAdvisor ">
  <property name="windowCommandBarDefinitions"
    value="commands-context . xml" />
  <property name="startingPageId "
    value="initialView " />
  <property name="eventExceptionHandler "
    value="com . mypackage . myExceptionHandler " />
</bean>
```

Nel listato si vede come le definizioni della barra dei comandi(definite nel file `commands-context.xml`) vengano iniettate per mezzo della Dependency Injection nel gestore del ciclo di vita dell'applicazione. Questo fa in modo che gli eventi vengano propagati dal contesto principale al contesto dei comandi in modo che i bean definiti in quest'ultimo possano processare gli eventi quando richiesto.

Ulteriori approfondimenti ed esempi saranno fatti nel capitolo 4.

3.4 Form

Come in tutti i tipi di applicazioni(desktop ma anche web-oriented) la gestione dell'input dell'utente è di notevole importanza e quindi il supporto per le form deve essere ampio e facile da utilizzare.

Spring Rich Client mette a disposizione diverse funzionalità che aiutano lo sviluppatore, ad esempio un sistema efficiente di binding tra la form UI e l'oggetto del modello a cui la form è riferita. Inoltre si ha un sistema di validazione a run-time che si basa su regole definite in apposite classi e iniettate tramite Dependency Injection.

Sono disponibili diverse implementazioni di form, sia più semplici che composite, ad esempio se si hanno dei wizard con più step e ogni step comprende una form è possibile raggruppare tutti gli step in una unica form logica.



Figura 3.3: Un esempio di form composita tratto dall'applicazione PetClinic: si vede che la form è divisa in due sottoform tramite due tab

Per la costruzione delle form si utilizzano appositi FormBuilder in cui si specificano tutti i campi della UI passando come parametro lo stesso nome dei campi della classe del modello a cui la form va legata. Questo è sufficiente per effettuare il binding tra vista e modello.

Interessante in questo ambito è la tecnica dell'*Interception*, utilizzata come supporto quando si utilizza una form. Tale metodologia è utilizzata per avere funzionalità quali: completamento automatico, segnalazione di errori di validazione o di informazioni varie.

Le regole di validazione vengono specificate in opportune classi, che poi verranno iniettate a tempo di esecuzione, le regole di validazione. Basta specificare il campo della form a cui la regola è riferita e scegliere il tipo di validazione che si vuole attuare: è possibile scegliere limitazioni alfabetiche, numeriche e anche date da espressioni regolari.

Un esempio di utilizzo verrà fatto nel capitolo 4.

Capitolo 4

Una applicazione con Spring Rich Client

4.1 Introduzione

Quello che segue è un breve (ma sufficientemente esaustivo) HowTo per iniziare a sviluppare applicazioni tramite l'uso di Spring Rich Client. Verrà presa in esempio una applicazione che realizza i servizi di una segreteria didattica di una università. Sarà dunque di interesse la gestione di corsi, docenti, dispense ed avvisi.

E' bene sottolineare che questo è soltanto un esempio e in quanto tale estendibile o semplificabile. Ovviamente verranno presi in considerazione soltanto gli aspetti rilevanti per quanto riguarda il framework Spring Rich Client: non ci saranno, ad esempio, considerazioni sullo strato di persistenza o sulla gestione del modello.

4.2 I Contesti Applicativi

Nelle applicazioni *Spring-Based* sono di notevole importanza i contesti applicativi, definiti in files XML e nei quali avviene la maggior parte della

configurazione dell'applicazione.

Per chiarezza ed ordine è bene suddividere i vari contesti applicativi in base alle loro funzionalità. In questo caso si è scelto di utilizzare 4 contesti (e quindi 4 files XML). E' importante dire che questa scelta è del tutto arbitraria e non vincolante. Potrebbe anche venire configurato tutto in un singolo file XML, ma in questo modo si avrebbe una gestione dell'applicazione del tutto improponibile. Nel dettaglio si ha:

1. **applicationCtx.xml**: in questo files sono definiti tutti i bean che sono parte integrante dell'applicazione rich client, ad esempio gestori degli eventi, le viste, le classi adibite ai wizard di creazione degli oggetti, le posizioni dei bundle per le icone e i messaggi e altro ancora...
2. **businessCtx.xml**: in questo file ci sono le specifiche per la parte di business, del tutto indipendenti da Spring Rich Client, come ad esempio le specifiche per il DataSource o per le implementazioni per le classi DAO
3. **commandsCtx.xml**: in questo file vengono definite tutte le barre dei comandi e degli strumenti
4. **securityCtx.xml**: in quest'ultimo file ci sono regole per gestire la sicurezza in fase di Login ¹

Il punto di ingresso con l'applicazione è dato in maniera molto semplice dal seguente comando definito nella Main-Class dell'applicazione e che si limita a caricare tutti i contesti applicativi e ad agire da Launcher:

¹In questo contesto c'è la definizione dei bean necessari ad ACEGI, framework per autenticazione e autorizzazione sviluppato dalla comunità Spring

```
public class Launcher {

    /** Creates a new instance of Launcher */
    public Launcher() {
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String [] args) {

        String rootCtx = "it/uniroma3/diaRich/ctx/";

        String startupCtx = rootCtx+"startupCtx.xml";
        String applicationCtx = rootCtx+"applicationCtx.xml";
        String businessCtx = rootCtx+"businessCtx.xml";
        String commandsCtx = rootCtx+"commandsCtx.xml";
        String securityCtx = rootCtx+"securityCtx.xml";
        try {
            new ApplicationLauncher(startupCtx ,
                new String []{ applicationCtx , businessCtx , securityCtx });
        } catch (Exception e){
            System.err.print(e);
            e.printStackTrace();
            System.exit(1);
        } }
    }
```

L'avvio dell'applicazione viene fatto semplicemente creando l'oggetto **ApplicationLauncher** e passando come parametro i contesti applicativi necessari. In questo caso è stato inserito anche uno *startupCtx*, nel quale è possibile definire uno splash-screen da visualizzare mentre viene caricata l'applicazione. Di seguito verranno analizzati in dettaglio i due contesti applicativi di interesse per Spring Rich Client: l'**applicationCtx** e il **commandsCtx**.

4.2.1 ApplicationCtx

La prima parte di questo contesto è formata dai bean che danno vita all'applicazione, ossia il bean dell'applicazione stessa, un bean che ne descrive le

caratteristiche, e uno che ne gestisce il ciclo di vita:

```
<bean id="application"
      class="org.springframework.richclient.application.Application">
  <constructor-arg index="0">
    <ref bean="applicationDescriptor"/>
  </constructor-arg>
  <constructor-arg index="1">
    <ref bean="diaRichLifecycleAdvisor"/>
  </constructor-arg>
</bean>

<bean id="applicationDescriptor"
      class="org...DefaultApplicationDescriptor">
  <property name="version">
    <value>1.0</value>
  </property>
  <property name="buildId">
    <value>20061225001</value>
  </property>
</bean>

<bean id="diaRichLifecycleAdvisor"
      class="it.uniroma3...util.DiaRichLifecycleAdvisor">
  <property name="startingPageId">
    <value>corsiView</value>
  </property>
  <property name="windowCommandBarDefinitions">
    <value>it/uniroma3/diaRich/ctx/commandsCtx.xml</value>
  </property>
</bean>
```

Come si vede nel listato, si hanno 3 **bean**, il primo rappresenta l'applicazione stessa in cui vengono iniettati le implementazioni per una breve descrizione (il secondo **bean**) e il gestore del ciclo di vita (il terzo **bean**) che si occuperà di caricare i comandi e di presentare all'utente la pagina iniziale specificata nello *startingPageId*: il valore *corsiView* fa riferimento ad un bean che rappresenta una vista, ed è il seguente:

```
<bean id="corsiView"
class="org...richclient.application.support.DefaultViewDescriptor">
  <property name="viewClass" value="it...diaRich.gui.view.CorsiView"/>
  <property name="viewProperties">
    <map>
      <entry key="diaRich">
        <ref bean="diaRich"/>
      </entry>
    </map>
  </property>
</bean>
```

Come sottolineato in precedenza, questo bean rappresenta la classe Java che serve a rappresentare i dati ottenuti dalla persistenza. La variabile che viene iniettata è necessaria infatti proprio per avere accesso alla persistenza.

La classe Java che rappresenta la vista è la seguente:

```
public class CorsiView extends AbstractView implements
ApplicationListener {

private DiaRich diaRich;

protected JComponent createControl(){
  //in questo metodo, da sovrascrivere, verranno estratti i dati
  //dallo strato di persistenza e presentati nella maniera più
  //opportuna
}

public void onApplicationEvent(ApplicationEvent applicationEvent){
  //ascoltatore/i degli eventi, per gestire le interazioni
  //user/application }

  //eventuali metodi accessori
}
```

Nell'applicationCtx vengono settati anche i bean dedicati alla gestione dei bundle per icone, testi, caption e figure:

```
<bean id="applicationObjectConfigurer"
      class="org.springframework...DefaultApplicationObjectConfigurer">
  <constructor-arg index="0">
    <ref bean="messageSource"/>
  </constructor-arg>
  <constructor-arg index="1">
    <ref bean="imageSource"/>
  </constructor-arg>
  <constructor-arg index="2">
    <ref bean="iconSource"/>
  </constructor-arg>
</bean>

<bean id="messageSource"
      class="org...context.support.ResourceBundleMessageSource">
  <property name="basenames">
    <list>
      <value>it.uniroma3.diaRich.gui.messages</value>
      <value>org...richclient.application.messages</value>
    </list>
  </property>
</bean>

<bean id="imageSource"
      class="org...richclient.image.DefaultImageSource">
  <constructor-arg index="0">
    <ref bean="imageResourcesFactory"/>
  </constructor-arg>
  <property name="brokenImageIndicator">
    <value>images/alert/error_obj.gif</value>
  </property>
</bean>

<bean id="imageResourcesFactory"
      class="org...context.support.ResourceMapFactoryBean">
  <property name="locations">
    <list>
      <value>org/springframework/image/images.properties</value>
      <value>it/uniroma3/diaRich/gui/images.properties</value>
    </list>
  </property>
```

```
<property name="resourceBasePath">
  <value>it/uniroma3/diaRich/gui/images/</value>
</property>
</bean>

<bean id="iconSource"
  class="org...richclient.image.DefaultIconSource">
  <constructor-arg index="0">
    <ref bean="imageSource"/>
  </constructor-arg>
</bean>
```

E' stato definito un oggetto atto a configurare tutti i widget dell'applicazione, a cui vengono iniettati gli oggetti che rappresentano le varie *sorgenti* per icone, immagini, e messaggi, tutti definiti in bundle per poter facilitare cambiamenti e internazionalizzazione dell'applicazione. Se infatti si vuole cambiare i messaggi e le etichette o le icone o quant'altro è sufficiente andare a modificare i file di properties, senza dover andare a modificare il codice. Questo è senz'altro un grande vantaggio in termini di comodità d'uso e di resistenza ai cambiamenti.

Una ulteriore componente definito in questo file sono gli Wizard per la creazione di oggetti:

```
<bean id="newDocenteWizard"
  class="it.uniroma3.diaRich.gui.wizard.NewDocenteWizard">
  <property name="diaRich">
    <ref bean="diaRich"/>
  </property>
</bean>
```

In questo caso si tratta di un wizard per la creazione di un oggetto di tipo Docente. La dichiarazione di questo bean nel contesto applicativo servirà quando, nella configurazione delle barre dei comandi e degli strumenti (*commandsCtx*), si dovrà indicare che un determinato pulsante corrisponderà alla creazione di un nuovo oggetto. Questo verrà esemplificato nel prossimo paragrafo.

La classe che si occupa dello wizard è la seguente:

```
public class NewDocenteWizard extends AbstractWizard implements
ActionCommandExecutor{

private DiaRich diaRich;
private WizardDialog wizardDialog;
private CompoundForm wizardForm;

protected boolean onFinish () {
//quando si è terminato di compilare la form si inserirà l'oggetto
//nello strato di persistenza e si pubblicherà l'evento associato
Docente docente = getNewDocente(); //prende il docente dalla form
diaRich.inserisciDocente(docente);
getApplicationContext().publishEvent(new
LifecycleApplicationEvent(LifecycleApplicationEvent.CREATED,
docente)); return true; }

public void execute() {
//Si effettua il binding tra Docente e la form
wizardDialog = new WizardDialog(this); //creazione del wizard
wizardForm = new CompoundForm(); //creazione della form
wizardForm.setFormObject(new DocenteBean()); //binding
wizardDialog.showDialog(); //viene mostrata la finestra di dialogo
}

public void addPages() { addPage(new FormBackedWizardPage(new
DocenteForm( FormModelHelper.createChildPageFormModel( //form
wizardForm.getFormModel())))); }
```

Come si vede dal listato, il wizard è composto da una form che a sua volta è composta da una sola pagina. La classe per la form è la seguente:

```
public class DocenteForm extends AbstractForm{

protected JComponent createFormControl(){

    TableFormBuilder fBuilder = new TableFormBuilder(getBindingFactory());
    this.nameField = fBuilder.add("nome")[1]; fBuilder.add("cognome");
    fBuilder.row();
    fBuilder.add("email"); fBuilder.add("orario");
    return fBuilder.getForm();
}}}
```

Il binding tra la form(UI) e l'oggetto docente(Model) viene fatto tramite i nomi dei campi della form, che devono essere gli stessi dei campi della classe del modello. Allo stesso modo, se l'oggetto (del modello) di tipo docente viene creato valorizzando alcuni campi della classe, nella form si avranno inizialmente dei valori già inseriti.

Un interessante strumento di supporto alle form è la validazione, che viene fatta in maniera (quasi) del tutto automatica. Le regole di validazione vengono infatti definite in una apposita classe Java, che viene anche definita nel contesto applicativo e che il framework utilizza quando carica tutti i contesti in fase di avvio dell'applicazione. Questa è la definizione nell'applicationCtx:

```
<bean id="rulesSource"
      class="it.uniroma3.diaRich.business.util.DiaRichRulesSource"/>
```

Come si vede dal listato la definizione è semplicissima, basta specificare la locazione della classe che si occuperà della validazione. Un esempio di come potrebbe essere implementata la classe è il seguente:

```
class DiaRichRulesSource{
public DiaRichRulesSource() {
    super();
    addRules(createDocenteRules());
    //posso aggiungere altre regole..
} private Rules createDocenteRules() {
    return new Rules(Docente.class) {
        protected void initRules() {
            add("nome", getNameValueConstraint());
            add("cognome", getNameValueConstraint());
            add(not(eqProperty("nome", "cognome")));
        }
    };
}
private Constraint getNameValueConstraint() {
    return all(new Constraint[] {
        required(), maxLength(25), regexp("[a-z_A-Z]*", "alphanumeric")});
}}
```

Qui si è specificato che per la classe Docente, o meglio, quando si crea un oggetto della classe Docente, nella form, sia per il campo nome, che per il campo cognome, si utilizzeranno le regole restituite dal metodo *Constraint getNameValueConstraint()*. In questo metodo è stato definito che la lunghezza massima del valore del campo è di 25 caratteri, e che tale valore sarà dato da un'espressione regolare di caratteri compresi tra a-z e A-Z, ossia dovrà essere un valore strettamente alfabetico.

Inoltre si è anche specificato che il valore del nome dovrà essere diverso da quello del cognome. Oltre a queste si hanno disponibili molti altri criteri di validazione.

La validazione è fatta a *run-time* e se si sta violando anche una soltanto delle regole definite non viene attivato il pulsante di *Submit* della form e inoltre compaiono dei messaggi accanto alla form che specificano che si stanno violando le regole (*Ajax-Style*).

4.2.2 CommandsCtx

L'altro contesto fondamentale nelle applicazioni sviluppate con Spring Rich Client è il contesto in cui vengono definiti tutti i comandi per le barre degli strumenti e le barre dei menu.

La configurazione è estremamente semplice, basta infatti definire nel contesto tutti i menu e la loro configurazione e sarà poi compito di Spring incollare il tutto e presentarlo all'utente.

Innanzitutto dovranno essere definiti i comandi condivisi da tutta l'applicazione: taglia, copia, annulla. . .

```
<bean id="windowCommandManager"
      class="org...application.support.ApplicationWindowCommandManager">
  <property name="sharedCommandIds">
    <list>
      <value>saveAsCommand</value>
      <value>propertiesCommand</value>
      <value>renameCommand</value>
      // altri command
    </list>
  </property>
</bean>
```

Questi comandi, per l'appunto condivisi, sono utilizzabili globalmente nell'applicazione, quindi da qualsiasi vista e in qualsiasi punto del sistema è possibile utilizzarli.

Si deve poi specificare la configurazione della barra del menu e i comandi che faranno parte di essa:

```
<bean id="menuBar"
      class="org...command.CommandGroupFactoryBean">
  <property name="members">
    <list>
      <ref bean="fileMenu"/>
      <ref bean="editMenu"/>
      <ref bean="windowMenu"/>
      <ref bean="helpMenu"/>
    </list>
  </property>
</bean>
```

In questo caso si è scelto di inserire nella barra dei menu, quattro componenti: un menu per il file, un menu di modifica, un menu di opzioni per la finestra, e un menu di help. Come si vede dal listato quei riferimenti sono verso altri bean, quindi verso altri oggetti complessi che devono essere definiti. Il fileMenu, sarà, ad esempio, fatto in questo modo:

```
<bean id="fileMenu"
      class="org ... command.CommandGroupFactoryBean">
  <property name="members">
    <list>
      <ref bean="newMenu"/>
      <value>separator</value>
      <value>saveAsCommand</value>
      <value>propertiesCommand</value>
      <value>separator</value>
      <bean class="org ... command.support.ExitCommand"/>
    </list>
  </property>
</bean>
```

Avrà al suo interno un riferimento verso un menu, *newMenu*, dal quale sarà possibile creare nuovi oggetti (docenti, corsi...), un separatore, dei comandi di utilità, nuovamente un separatore, e poi un comando di uscita dall'applicazione.

Nel *newMenu* avremo i comandi relativi alla creazione di nuovi oggetti:

```
<bean id="newMenu"
      class="org ... command.CommandGroupFactoryBean">
  <property name="members">
    <list>
      <ref bean="newDocenteCommand"/>
      <ref bean="newCorsoCommand"/>
      <ref bean="newDispensaCommand"/>
      <ref bean="newAvvisoCommand"/>
    </list>
  </property>
</bean>
```

Questi command saranno ovviamente legati alle classi della logica applicativa che si occupano della creazione dei nuovi oggetti:

```
<bean id="newCorsoCommand"
      class="org...command.TargetableActionCommand">
  <property name="commandExecutor">
    <ref bean="newCorsoWizard" />
  </property>
</bean>
```

In questo caso si sta specificando che la classe che si occuperà di eseguire quel comando è la classe specificata dal bean *newCorsoWizard*, i cui dettagli sono stati descritti nel paragrafo precedente.

Quello che è stato descritto è la base per iniziare a sviluppare applicazioni desktop basate su Spring Rich Client.

Bibliografia

[1] Spring reference documentation. <http://www.springframework.org/>

[2] Spring Rich Client Wiki <http://spring-rich-c.sourceforge.net/>