Seconda Università degli Studi di Napoli
Facoltà di Ingegneria

Tesi di Laurea in

Ingegneria Informatica

NETSHOP:
un centro commerciale virtuale

Relatore:
Ch.mo Prof.
Antonio d'Acierno

Candidato:
Luigi Luongo
Matr: 834/486

Anno Accademico 2006/2007

... alla mia pazientissima famiglia
... alle persone che mi vogliono bene
... all'amore della mia vita Silvia

Ringraziamenti

Eccomi oramai giunto, dopo quattro anni, al termine di questa grande avventura ricca di difficoltà e di emozioni, che ha saputo farmi amare il mondo dell'ingegneria ed in particolare quello dell'informatica. In questi anni di studio ho dedicato tutto me stesso al raggiungimento di questo obiettivo, aggiungendovi costanza, instancabilità, caparbietà e tenacia che comunque sarebbero state vane senza le persone che qui ho il piacere ed il dovere di ringraziare.

I miei ringraziamenti vanno innanzitutto al *Prof. Ing. Antonio d'Acierno*, il quale ha accolto la mia proposta per questo lavoro di tesi, rendendosi sempre disponibile per qualsiasi chiarimento e dandomi sempre buoni consigli per migliorare il lavoro che stavo facendo. Grazie a lui, ho avuto la possibilità di affrontare ed approfondire nuovi temi legati allo sviluppo di applicazioni per Internet, per me di estremo interesse.

Un grazie va alla mia pazientissima famiglia, *Mamma*, *Papà*, *sorella* e *fratello*, che mi ha dato la possibilità di realizzare questo sogno, senza mai ostacolarmi nelle decisioni ma dandomi sempre il massimo appoggio. A loro devo chiedere scusa, per non essere stato molto presente nella famiglia.

Un grazie va ai miei amici e compagni di università con i quali ho condiviso questa esperienza, un grazie particolare al mio amico *Damiano* con il quale ho condiviso intensamente i primi anni di università.

Un grazie enorme va alla mia fidanzata *Silvia*, che ho avuto l'immensa fortuna di conoscere circa dieci anni fa e che diventata l'amore della mia vita. Si è ritrovata "catapultata" nel mio mondo, fatto di moltissime ore di studio, di sacrifici e di preoccupazioni, dandomi sempre la forza di rialzarmi nei momenti di rabbia e di sconforto.

Vorrei infine ringraziare il *signore Dio* che mi ha dato e mi da la forza per poter affrontare le tante difficoltà della vita e la mia nonna *Giovanna*, che da lassù mi ha guidato in questi anni di studio.

Grazie di cuore a tutti.

Indice

Introduzione

Il lavoro di tesi svolto si pone come obiettivo lo sviluppo di una applicazione Web per il commercio elettronico, dal nome NetShop, che consenta l'utilizzo e la gestione di un centro commerciale virtuale. Per la realizzazione di tale applicazione Web viene utilizzato il framework Jakarta Struts, il quale si basa sul pattern MVC.

Nella presente tesi, si parte da una descrizione del pattern MVC e del modo in cui ha contribuito all'evoluzione dell'architettura delle Web applications. Successivamente, viene dato ampio spazio al framework Jakarta Struts, descrivendone tutte le classi che ne costituiscono l'architettura e tutte le funzionalità messe a disposizione per realizzare un'applicazione Web.

Infine si passa alla descrizione dello sviluppo dell'applicazione Web, attraverso le varie fasi. Vengono presentate la fase di analisi e progettazione, che hanno previsto la realizzazione dei principali diagrammi UML (Use Case, Class, Activity, Sequence) oltre al modello concettuale e logico della base dati. A queste fasi, fa seguito la fase di implementazione, che ha previsto la realizzazione della Web application utilizzando il framework Struts.

Per quanto concerne gli strumenti adottati si è fatto ampio uso del software Sybase PowerDesigner per lo sviluppo dei diagrammi UML, dell'ambiente IDE Eclipse 3.2 e dei plug-in di Exadel Studio per l'implementazione, del Web Container Apache Tomcat 5.5 come ambiente di esecuzione per l'implementazione, di MySQL 5 come DBMS e di MySQL GUI Tool 1.2 per la creazione e la gestione della base dati ed infine di Macromedia Dreamweaver 8 e Fireworks 8 per la creazione della parte grafica.

Capitolo 1

Il pattern MVC

1.1 Introduzione

Un pattern è un modello che permette di definire la soluzione di un problema specifico che si ripresenta, di volta in volta, in un contesto diverso.

Il pattern MVC si basa sull'idea di separare i dati dalla loro rappresentazione, poiché mantenere un forte accoppiamento tra essi comporta che la modifica dell'uno, implica automaticamente un'aggiornamento dell'altro. Esso, quindi, prevede che un sistema software sia realizzato secondo un'architettura a livelli, stabilendo un disaccoppiamento fra dati e rappresentazione, mediante la definizione di tre elementi noti come: *Model*, *View* e *Controller*.

Il Model (Modello) incapsula lo stato dell'applicazione. E' responsabile della gestione dei dati e del comportamento (data & behaviour) dell'applicazione ed espone le funzionalità di quest'ultima. Esso coordina la logica dell'applicazione (*business logic*), l'accesso alle basi di dati (*data access*) e tutte le parti critiche nascoste del sistema. E' indipendente dalle specifiche rappresentazioni dei dati sullo schermo e dalle modalità di input dei dati stessi da parte dell'utente.

La View (Vista) ha il compito di visualizzare lo stato del Model e lo presenta all'utente anche in forme diverse, in relazione al dispositivo utilizzato per accedere al sistema (es. personal computer, cellulare, ...). Ciò vuol dire

che, pur partendo dagli stessi dati, è possibile effettuare rendering diversi ed ottenere viste multiple dello stesso modello. Permette di poter inviare l'input dell'utente al Controller e di poter richiede delle modifiche al Model.

Il Controller (Controllo) definisce il meccanismo mediante il quale il Model e la View comunicano. Realizza la connessione logica tra l'interazione dell'utente con l'interfaccia applicativa e i servizi della *business logic* nel back-end del sistema. Esso va a mappare l'azione (*action*) richiesta dall'utente con le operazioni del Model. E' responsabile della scelta di una tra molteplici viste dello stesso modello, in base al tipo di dispositivo utilizzato dall'utente per accedere al sistema ma anche in relazione alla localizzazione geografica dell'utente stesso.

Una qualsiasi richiesta (*request*) fatta al sistema viene acquisita dal Controller, che individua all'interno del Model il gestore della richiesta (*request handler*). Ottenuto il risultato dell'elaborazione (*response*), il Controller stesso determina a quale View passare i dati per la presentazione degli stessi all'utente.

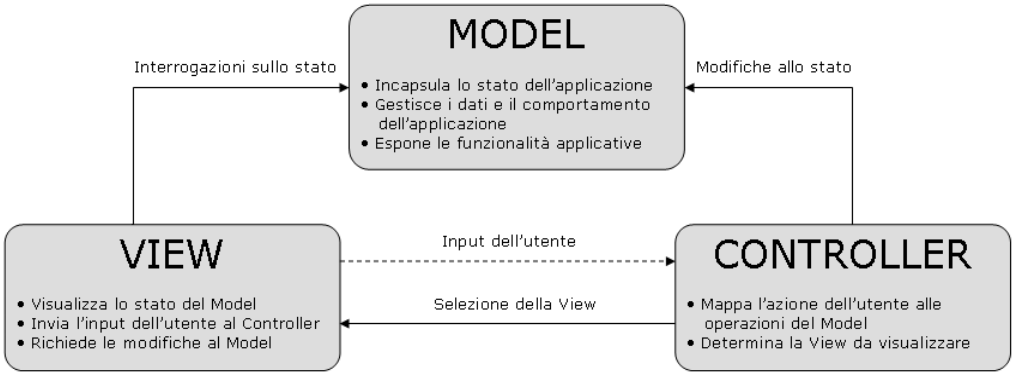

Figura 1.1: Model, View e Controller

1.2 Evoluzione dell'architettura delle Web Applications

Considerando il *Java* come uno dei migliori linguaggi per lo sviluppo delle applicazioni Web, attraverso l'uso delle *Servlets* e delle pagine *JSP (Java Server Pages)*, l'architettura delle Web applications ha subito una notevole evoluzione nel corso degli anni, seguendo un iter di questo tipo:

1. Assenza del pattern MVC;
2. Utilizzo del pattern MVC secondo il Model 1 (Page - Centric);
3. Utilizzo del pattern MVC secondo il Model 2 (Servlet - Centric);
4. Web Application Frameworks (es. Struts).

Tale evoluzione ha previsto un aumento della complessità e della robustezza di ciascuna applicazione e può essere schematizzata nel modo seguente, sino all'introduzione del Model 1:

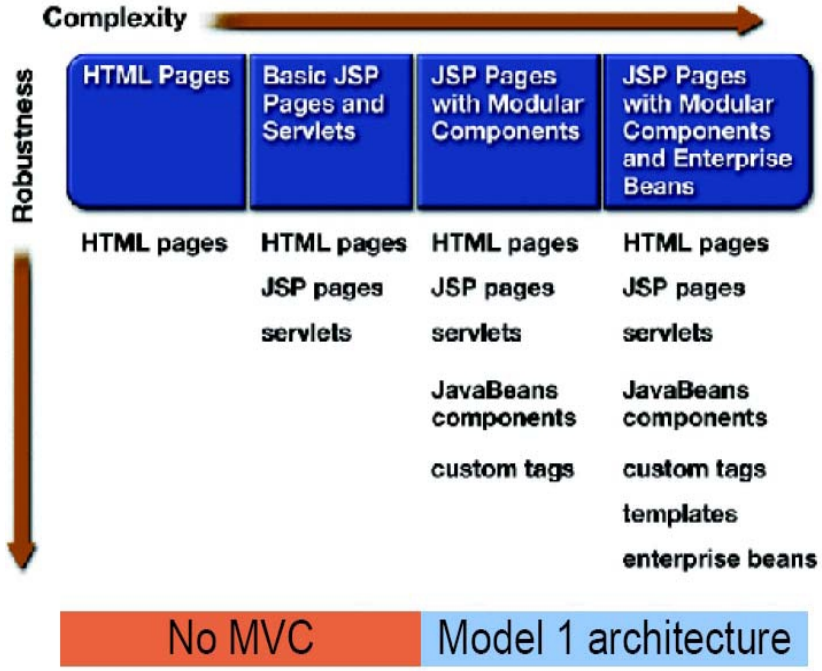

Figura 1.2: Evoluzione dello sviluppo di una Web Application

1.2.1 Model 1 (Page - Centric Architecture)

Il Model 1 del pattern MVC è detto anche Page - Centric, poiché l'architettura della Web application è basata sulle pagine JSP. Il sistema complessivo è composto da una serie di pagine JSP legate fra loro, ciascuna delle quali gestisce tutti gli aspetti principali dell'applicazione tra cui la presentazione, il controllo ed i processi di business. E' evidente che la *business logic* ed il controllo sono fortemente accoppiati all'interno di ciascuna pagina JSP, attraverso l'utilizzo di *JavaBeans*, *Scriptlets* ed Espressioni. I tre elementi del pattern, Model, View e Controller, pur essendo distinti, sono inglobati all'interno di una stessa struttura che in questo caso è rappresentata da una pagina JSP.

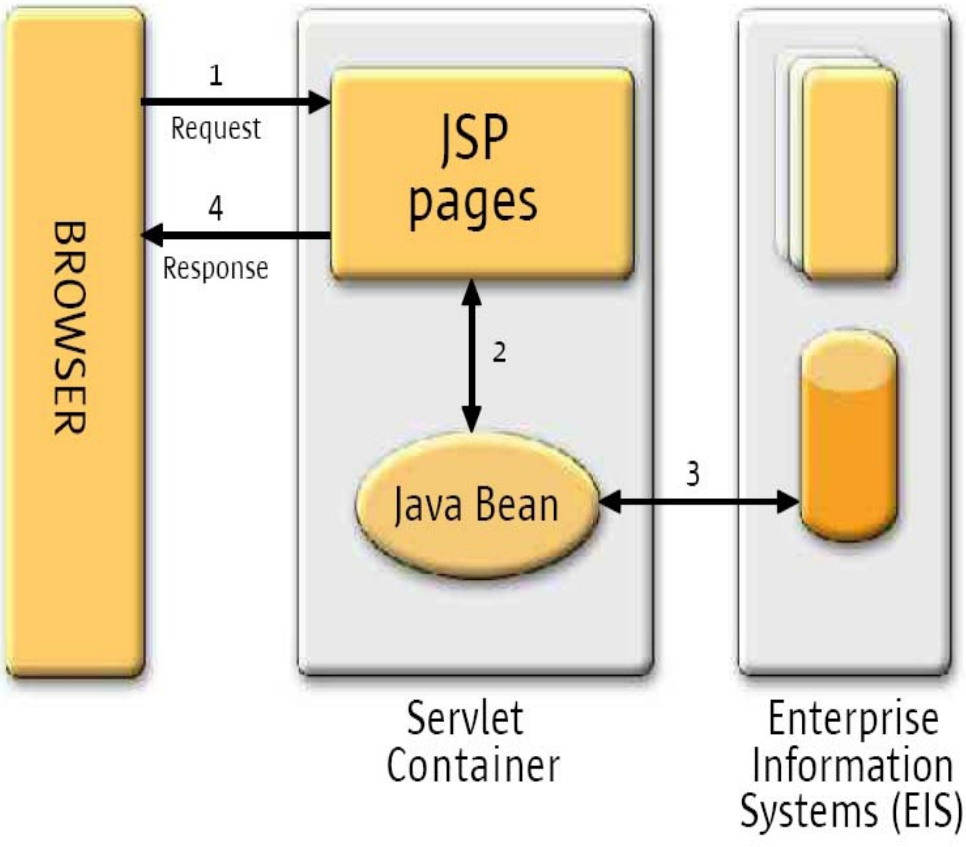

Figura 1.3: MVC Model 1 (Page - Centric Architecture)

Il modello prevede uno scenario di interazione di questo tipo:

1. Il browser invia una richiesta (request) di una risorsa al Web Server, nella maggior parte dei casi per la visualizzazione di una pagina JSP.

2. Il Web Server, tipicamente, funge da Web Container e quindi da Servlet Container in quanto va ricordato che una pagina JSP, una volta richiesta, viene sempre trasformata in una corrispondente Servlet. All'interno della pagina JSP, ci sono i tag che ne definiscono la presentazione all'utente e quindi l'aspetto grafico (GUI) , ma anche gli elementi per l'esecuzione delle elaborazioni. Queste ultime possono essere eseguite all'interno della pagina stessa, attraverso codice Java immerso nei tag (Scriptlets) oppure mediante dei componenti esterni, ai quali si fa riferimento da tale pagina.

3. I componenti in questione sono tipicamente dei JavaBeans, i quali effettuano una qualsiasi computazione, comunicando eventualmente con il back-end del sistema, ad esempio per l'accesso a basi di dati.

4. Il risultato di ciascuna elaborazione sarà così integrato all'interno della pagina HTML prodotta, che sarà inviata nella risposta (response) al browser.

Da quanto detto, si evince che Model, View e Controller sono praticamente integrati all'interno di ciascuna pagina JSP e che non c'è una netta separazione fra essi. I limiti principali di questo modello possono essere i seguenti:

- viene incoraggiata una struttura a spaghetti delle pagine JSP, poiché la *business logic* si perde all'interno di ciascuna pagina e la navigazione nella Web application complessiva viene fatta pagina per pagina;

- è molto difficile eseguirne il debug, poiché tutti gli errori riportati dal Web Container, fanno riferimenti al codice compilato della pagina JSP

in una Servlet e quindi sono difficili da individuare all'interno della pagina stessa.

Tutto ciò rappresenta un primo passo verso il modello definitivo del pattern MVC.

1.2.2 Model 2 (Servlet - Centric Architecture)

Il Model 2 del pattern MVC è detto anche Servlet - Centric, poiché l'architettura della Web application si basa fortemente sull'utilizzo di una Servlet. Il sistema complessivo è composto da una Servlet principale che svolge il ruolo di Controller, da una serie di pagine JSP che rappresentano la View ed infine da un insieme di JavaBeans che costituiscono il Model. I tre elementi del pattern MVC sono quindi nettamente separati tra loro, pur mantenendo la possibilità di comunicare per scambiarsi informazioni. In particolare, le pagine JSP si occupano esclusivamente della presentazione dei dati all'utente, senza contenere un minimo di *business logic*. La Servlet master ha il compito di acquisire tutte le richieste provenienti dalla rete e funge da dispatcher, inoltrando ciascuna di esse verso il corrispondente handler per permetterne la gestione. Al termine dell'elaborazione, è la stessa Servlet che determina a quale pagina JSP restituire il controllo, eseguendo il redirecting. Infine, i JavaBeans incapsulano le funzionalità dell'applicazione, interagiscono con il back-end del sistema ed eseguono tutte le elaborazioni richieste. Su tale modello si basa il framework Struts.

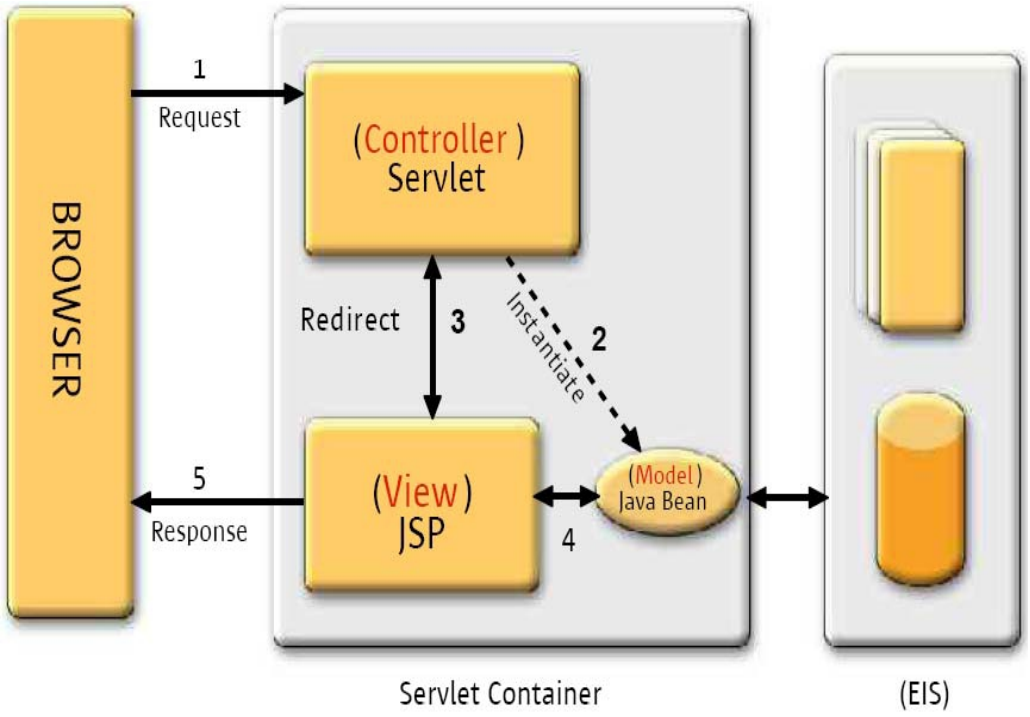

Figura 1.4: MVC Model 2 (Servlet - Centric Architecture)

Il modello prevede uno scenario iterativo di questo tipo:

1. Il browser invia una richiesta (request) di una risorsa al Web Server, nella maggior parte dei casi per la visualizzazione di una pagina JSP.

2. Il Web Server, tipicamente funge da Web Container e quindi da Servlet Container, in quanto è in esecuzione su di esso, la master Servlet della Web application. Quest'ultima funge da Controller ed acquisisce la richiesta, sulla base della quale individua l'handler che dovrà gestirla. In particolare, vengono istanziati una serie di JavaBeans, costituenti il Model, che eseguono le elaborazioni richieste ed eventualmente interagiscono con il back-end del sistema, accedendo ad una base di dati.

3. Al termine della computazione, il controllo ritorna alla Servlet che sulla base del risultato, determina la View e quindi la pagina JSP verso la

quale eseguire il redirecting, la quale estrae dai JavaBeans i risultati e li visualizza all'utente.

4. Infine, la pagina HTML prodotta viene trasmessa al browser come risposta (response) definitiva.

Da quanto detto si evince che Model, View e Controller sono nettamente separati fra loro e che il Controller funge da tramite per la comunicazione tra i primi due. Da ciò, scaturisce anche che il debug può essere eseguito in maniera piuttosto semplice su ciascun JavaBeans, estrapolandolo dal sistema complessivo.

Per quanto riguarda la struttura del Controller, è possibile pensare di utilizzare una sola Servlet come detto sino ad ora, oppure più Servlets. La scelta dipende dal livello di granularità della Web application ed è possibile pensare a tre diverse soluzioni:

- una sola master Servlet;
- una Servlet per ciascun caso d'uso dell'applicazione oppure per ogni macrofunzionalità offerta;
- una combinazione delle due precedenti soluzioni con una master Servlet per gestire le funzioni comuni a tutti gli ambiti dell'applicazione, che delega alle Servlets figlie la gestione di particolari macrofunzionalità del sistema.

1.2.3 Web Application Deployment Descriptor

Il Web Application Deployment Descriptor è un file XML, tipicamente chiamato *web.xml*, all'interno del quale ci sono tutte le informazioni di configurazione dell'applicazione, che vengono utilizzate dal Web Container in fase di start-up della stessa.

Nell'ambito dell'utilizzo di un framework, le informazioni principali che costituiscono tale file sono le seguenti:

- definizione della Servlet che funge da Controller e mapping delle richieste su di essa;
- dichiarazione dei parametri iniziali che sono passati alla Servlet in fase di start-up;
- configurazione delle librerie di tag (*Tag Libraries*) da adottare;
- elenco dei file di benvenuto dell'applicazione (*Welcome File List*);
- utilizzo di eventuali filtri per il filtraggio delle richieste prima che vengano passate alla Servlet Controller.

Per la definizione della Servlet che funge da Controller dell'applicazione, viene utilizzato il tag <*servlet*>, mediante il quale vengono specificate le seguenti informazioni:

- il nome da assegnare alla Servlet, mediante il quale fare riferimento ad essa nel resto del file (<*servlet-name*>);
- la classe che definisce la Servlet (<*servlet-class*>).

Il passo successivo è quello di fare in modo che, ciascuna richiesta di un certo tipo, ad esempio verso risorse con una particolare estensione, vengano smistate dal Web Container verso la Servlet specificata. Tale operazione è nota come *mapping*, in quanto viene eseguita una vera e propria mappatura fra le tipologie di richieste e la Servlet. Per questo scopo, viene utilizzato

il tag <*servlet-mapping*> all'interno del quale sono specificate le seguenti informazioni:

- il nome della Servlet a cui inoltrare le richieste (<*servlet-name*>);
- elenco delle estensioni che devono avere le richieste per essere inoltrate alla Servlet suddetta (<*url-pattern*>).

Considerando l'utilizzo di un framework come Struts, la Servlet che funge da Controller ha un parametro iniziale che è rappresentato dal file di configurazione XML dell'applicazione. In questo modo, nella fase di start-up di quest'ultima, la Servlet del framework esegue l'operazione di parsing e caricamento del contenuto del file, all'interno della memoria. Per questo scopo, viene adottato il tag <*init-param*>, mediante il quale è possibile specificare un qualsiasi tipo di parametro iniziale della Servlet, oltre all'applicazione particolare suddetta. All'interno del tag, possono essere definite le seguenti informazioni:

- nome del parametro (<*param-name*>);
- valore del parametro (<*param-value*>).

Generalmente, una Web application prevede l'utilizzo dei normali tag HTML all'interno delle proprie pagine, ma anche eventualmente ulteriori tag appartenenti a librerie particolari, come quelle che vengono fornite con i framework Struts. Per poter rendere disponibili tali librerie, esse vanno specificate all'interno del file *web.xml* mediante l'utilizzo del tag <*taglib*>, specificando le seguenti informazioni:

- la locazione fisica del file *TLD (Tag Library Descriptor)* che contiene la descrizione dei tag della libreria (<*taglib-location*>);
- un URI associato alla libreria, per includerla all'interno delle pagine ed evitare di fare riferimento alla locazione fisica (<*taglib-uri*>).

Infine, è possibile specificare un elenco di risorse, tipicamente pagine, che il Web Container invia per una certa richiesta se quest'ultima non è completa, ossia prevede soltanto una parte dell'URL. Ciascun file viene definito con il tag <*welcome-file*>, all'interno del blocco <*welcome-file-list*>.

Capitolo 2

Il framework Struts

2.1 Introduzione

Un framework fornisce dei componenti generici che collaborano fra di loro, i quali possono essere estesi per fornire una serie di funzioni. I luoghi in cui il framework può essere esteso sono detti *punti di estensione*.

Struts è un framework *Open Source* per la realizzazione di Web applications secondo il pattern MVC. Esso è costituito da una serie di classi e di interfacce che ne costituiscono l'infrastruttura. Le *API (Application Program Interface)* delle classi permettono di:

- acquisire i dati dei form riempiti dall'utente durante la navigazione ed eseguire le elaborazioni su di essi;
- individuare gli handlers (gestori), detti anche *actions*, per esaudire ciascuna richiesta;
- definire la navigazione fra le pagine all'interno della Web application;
- supportare l'internazionalizzazione e l'accessibilità.

Inoltre, include i due seguenti framework integrati attraverso il meccanismo dei plug-in:

- *Tiles*: framework per la realizzazione del layout delle pagine con mattonelle (tiles) componibili;
- *Validator*: framework che fornisce un vasto insieme di validatori, per la validazione dei dati immessi dall'utente nei form.

Infine, mette a disposizione tre librerie di tag (Tag Libraries) :

- *Struts HTML*: per la realizzazione dell'interfaccia utente, con una serie di tag che permettono di introdurre i componenti per la costruzione di ciascuna pagina;
- *Struts Bean*: per poter accedere, direttamente da ciascuna pagina, alle proprietà dei JavaBeans che definiscono il Model dell'applicazione;
- *Struts Logic*: per definire dei costrutti di controllo (condizionali e ciclici) che permettono la costruzione dinamica di una pagina.

Ciò che rende Struts uno degli strumenti maggiormente utilizzati per lo sviluppo di Web applications, è soprattutto il fatto di essere Open Source. Infatti, milioni di sviluppatori in tutto il mondo partecipano al miglioramento del framework, realizzando classi aggiuntive che ne aumentano le potenzialità oppure migliorando quelle preesistenti. Nel tempo, tutto ciò ha reso Struts un strumento stabile e fortemente maturo con un curva di apprendimento (*learning curve*) piuttosto bassa.

Esso si basa fortemente sugli strumenti che costituiscono principalmente un'applicazione Web realizzata in Java: le pagine JSP e le Servlet. Infatti, il flusso di esecuzione all'interno di ciascuna Web application è gestito attraverso una Servlet, in esecuzione nel Web Container.

Ovviamente, alle numerose classi che costituiscono il framework, è possibile aggiungere ulteriori classi che definiscono la *business logic*, tipicamente realizzate attraverso dei JavaBeans.

La semplicità di configurazione della Web application da realizzare è garantita dalla presenza di un file XML, chiamato *struts-config.xml*, nel quale

vengono impostati tutti i parametri e gli elementi costitutivi dell'applicazione stessa.

Infine, dalla versione 1.1, è stata introdotta la possibilità di realizzare un'applicazione di grandi dimensioni suddividendola in moduli, ciascuno dei quali definisce una parte delle funzionalità complessive ed ha un proprio file di configurazione autonomo. In questo modo, si può gestire ciascun modulo separatamente dagli altri ma è ovviamente garantita la comunicazione fra di essi.

Tutti gli elementi che costituiscono il framework sono raggruppati in otto principali package:

- *action*, *actions*: contengono le classi che costituiscono il Controller e le classi che possono essere usate o estese all'interno della propria applicazione;
- *config*: contiene le classi che definiscono una rappresentazione in memoria, della configurazione dell'applicazione;
- *taglib*: contiene i Tag Handler, ossia le classi per la gestione delle librerie di tag di Struts;
- *tiles*: include le classi utilizzate dal framework Tiles;
- *upload*: contiene le classi per eseguire l'upload di file attraverso un browser Web;
- *util*: include delle classi general-purpose, con delle utility sfruttate dal framework;
- *validator*: contiene le classi specifiche di Struts per lo sviluppo di validatori personalizzati.

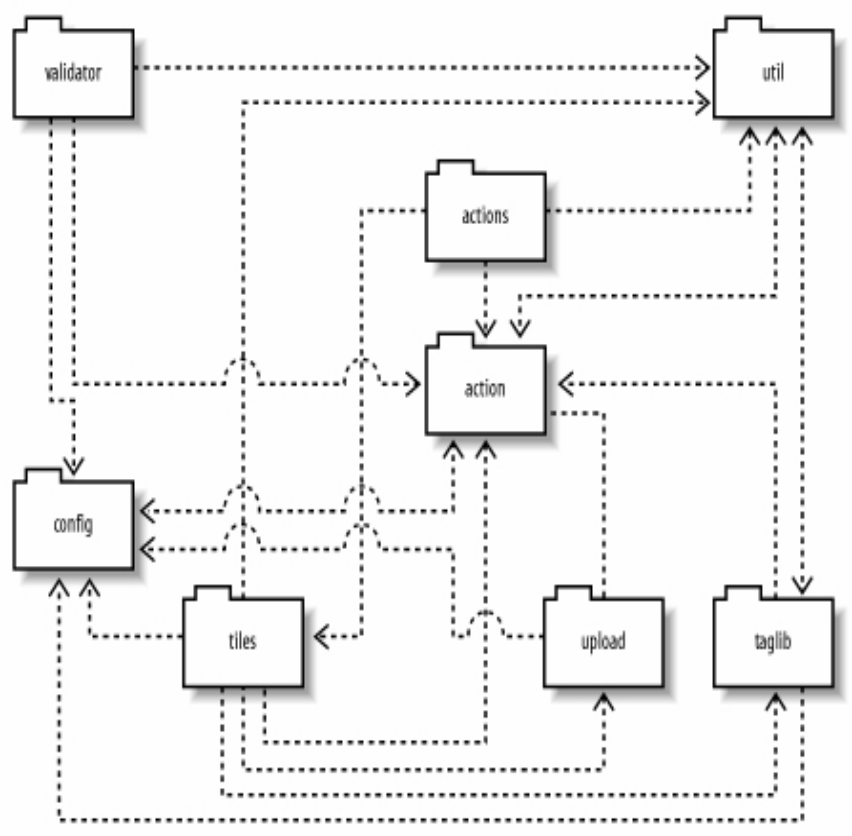

Figura 2.1: Architettura Packages Struts

2.2 Componenti del Controller, del Model e della View

Basandosi sul pattern MVC, è ovvio che Struts è caratterizzato da una serie di componenti e di classi, che compongono ciascuna delle tre parti del pattern stesso.

2.2.1 Componenti del Controller

Il Controller di Struts gestisce tutte le richieste in maniera centralizzata. Oltre ai numerosi vantaggi relativi alle funzionalità dell'applicazione, ne

scaturisce che alcuni servizi come la sicurezza, l'internazionalizzazione ed il logging sono concentrati nel Controller.

Facendo riferimento al pattern MVC, il Controller di Struts ha i seguenti compiti :

- ricevere tutte le richieste provenienti dai vari client;
- per ciascuna richiesta, determinarne il gestore, detto anche action, che avrà il compito di eseguire le elaborazioni richieste, utilizzando anche i JavaBeans che costituiscono la *business logic*;
- raccogliere i risultati delle elaborazioni per renderli disponibili al client;
- determinare la View alla quale passare il controllo per la visualizzazione dei risultati.

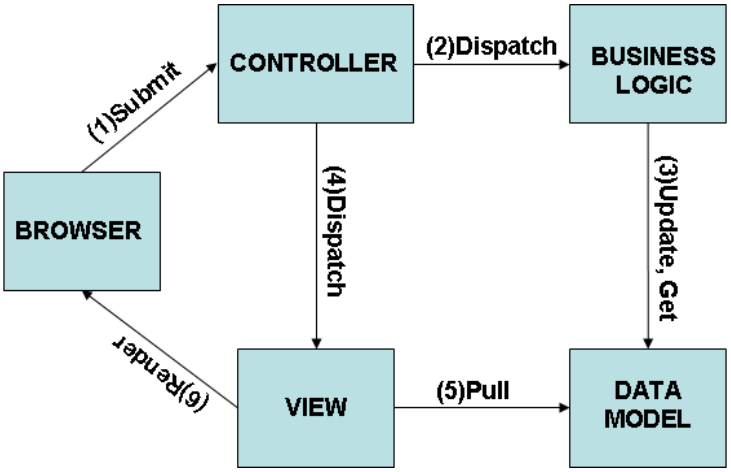

Figura 2.2: Pattern MVC

Struts prevede una serie di classi, legate fra loro, per la strutturazione del Controller.

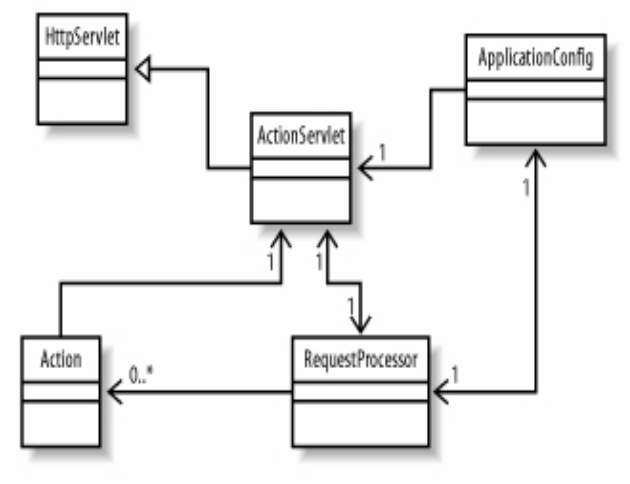

Figura 2.3: Class Diagram Controller Components

2.2.1.1 ActionServlet

La classe *ActionServlet* rappresenta la Servlet in esecuzione nel Web Container, alla quale arrivano tutte richieste dei client, dirette all'applicazione stessa.

Il framework da la possibilità allo sviluppatore di realizzare una classe che estenda *ActionServlet*, per poter definire un Controller personalizzato, che va comunque registrato nel Deployment Descriptor *web.xml*, per segnalare al Web Container quale Servlet avviare per gestire l'applicazione.

2.2.1.2 RequestProcessor

Dalla versione 1.1 del framework, alla classe *ActionServlet* è stata affiancata la classe *RequestProcessor*. Si è resa necessaria tale introduzione, in virtù del fatto che, proprio da tale versione, è stato introdotto il supporto di realizzazione di un'applicazione in più moduli separati. In questo caso, a partire dalla classe *ActionServlet*, vengono create più istanze della classe *RequestProcessor*, quanti sono i moduli dell'applicazione, in modo tale che ciascuno di essi abbia il proprio request handler, per la gestione delle richieste in ingresso.

Nel momento in cui arriva una richiesta alla *ActionServlet*, quest'ulti-

ma determina il modulo a cui essa è destinata ed istanzia la relativa classe *RequestProcessor*, verso la quale inoltra la richiesta.

2.2.1.3 Action

La classe *Action* può essere considerata il cuore del framework. Essa definisce un ponte tra il Controller ed il Model, in virtù del fatto che riceve dal primo la richiesta da esaudire ed utilizza le classi del secondo per espletarla.

Per la gestione di una richiesta da parte di una action, viene invocato su di essa il metodo *execute()*, all'interno del quale vengono istanziati gli oggetti che compongono la *business logic* ed eseguono l'elaborazione richiesta. Tale metodo restituisce un oggetto della classe *ActionForward*, che specifica la direzione verso la quale incanalare il flusso dell'applicazione. Al termine dell'esecuzione di tale metodo, il controllo ritorna al Controller che determina, sulla base dell'oggetto *ActionForward*, a quale View compete la visualizzazione.

```
public ActionForward execute(ActionMapping mapping,
  ActionForm form,
  HttpServletRequest request,
  HttpServletResponse response) throws Exception {

  ...
}
```

Si osservi che il metodo *execute()* riceve in ingresso l'oggetto *ActionMapping* che contiene il mapping delle action dell'applicazione e serve per individuare i forward, l'oggetto *ActionForm* che contiene i dati del form trasmesso dall'utente ed infine, gli oggetti *HttpServletRequest* ed *HttpServletResponse* relativi alla richiesta ed alla risposta.

La classe *Action* è definita *thread-safe*, nel senso che, nell'ambito di un'applicazione, viene creata sempre e solo un'unica istanza di ciascuna action. Ciò vuol dire che tutti i client condividono la medesima istanza di ciascuna di esse e possono invocare sulla medesima action il metodo *execute()* contemporaneamente, proprio perché la gestione avviene con thread separati.

L'elenco delle action istanziate è contenuto all'interno di una *HashMap*

della classe *RequestProcessor*. In virtù dell'approccio thread-safe con cui vengono gestite le action, è consigliabile non utilizzare variabili di istanza per ciascuna di esse, che altrimenti verrebbero condivise da tutti i client.

Inoltre, il framework mette a disposizione delle particolari classi che estendono il funzionamento della classe *Action*:

- *ForwardAction*;
- *DispatchAction*;
- *LookupDispatchAction*;
- *SwitchAction*.

ForwardAction

In moltissime situazioni, c'è la necessità di passare da una pagina JSP all'altra, senza il bisogno di utilizzare una *Action*. In generale, invocare direttamente una pagina JSP è un approccio non consigliato per molteplici motivazioni.

Il Controller ha il compito di selezionare opportunamente il modulo dell'applicazione a cui è diretta una richiesta per poterne caricare la configurazione ed i messaggi del Resource Bundle. Se questa fase viene saltata, si corre il rischio che non vengano visualizzati all'utente i messaggi corretti.

Un'altra motivazione è puramente concettuale, nel senso che, chiamare direttamente una pagina JSP viola il pattern MVC, poiché non si sfrutta il Controller. Per risolvere questa problema, è stata introdotta la classe *ForwardAction* che permette di eseguire il forward da una pagina all'altra, passando attraverso il Controller.

DispatchAction - LookupDispatchAction

In moltissimi casi, un'applicazione Web è caratterizzata da una serie di funzionalità che sono più o meno logicamente correlate. La soluzione di base sarebbe quella di definire più action separate, ciascuna delle quali svolga una

delle funzionalità, oppure definire un'unica action ed all'interno del metodo *execute()* gestire le diverse funzionalità sulla base dei parametri della request.

La migliore soluzione può essere realizzata utilizzando la classe *DispatchAction*, all'interno della quale è possibile avere più metodi, con la stessa signature del metodo *execute()*, che implementano le diverse funzionalità.

```
public class ClasseAction extends DispatchAction {

  public ActionForward metodo1(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response) throws Exception {

    ...
  }

  public ActionForward metodo2(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response) throws Exception {

    ...
  }
}
```

Per poter distinguere l'invocazione di un metodo da un altro, alla action viene assegnato un parametro specificato all'interno del file di configurazione mediante l'attributo parameter. Di volta in volta, tale parametro può assumere un valore diverso, tipicamente il nome di uno dei metodi della classe *DispatchAction*, in modo tale da utilizzare una delle funzionalità implementate.

```
<action input="pagina.jsp" name="nomeFormBean"
  parameter="dispatch" path="/nomeAction" scope="request"
  type="package.ClasseAction">
  <forward name="nomeForward1" path="pagina1.jsp"/>
  <forward name="nomeForwardN" path="paginaN.jsp"/>
</action>
```

In conclusione, ciascuna request destinata alla action, dovrà specificare il nome di quest'ultima seguito dal parametro con il suo relativo valore, per invocare uno dei metodi interni alla classe.

```
<html:form action="/nomeAction.do?dispatch=metodo1">
  ...
</html:form>
```

Un ulteriore miglioramento è stato introdotto dalla classe *LookupDispatchAction*, la quale prevede il metodo *getKeyMethodMap()*, che restituisce un oggetto della classe *Map*, facendo corrispondere ad una certa chiave del Resource Bundle, il nome di uno dei metodi della classe. In questo modo, la request non deve necessariamente assegnare al parametro suddetto il nome del metodo da invocare.

SwitchAction

La classe *SwithcAction* è stata introdotta dalla versione 1.1 del framework, in concomitanza con la nuova potenzialità di poter sviluppare le applicazioni di grandi dimensioni suddividendole in moduli separati. Tale classe, infatti, permette di passare da un modulo all'altro ed in particolare raggiungere una delle risorse del modulo scelto. Per fare questo, la action va invocata con due parametri:

- *prefix*: definisce il nome del modulo verso cui spostare il controllo;
- *page*: l'URI della risorsa a cui accedere, all'interno del modulo scelto.

2.2.1.4 ActionForward

Come visto in precedenza, il metodo *execute()* della classe *Action*, che gestisce una certa richiesta, fornisce come parametro di uscita un oggetto della classe *ActionForward*. Tale classe rappresenta un'astrazione logica di una risorsa Web, che può essere una pagina JSP, una Servlet oppure eventualmente un'altra action. Essa è costruita intorno a ciascuna risorsa, in modo da disaccoppiare l'applicazione dalla locazione fisica delle stessa, che viene specificata nel file di configurazione.

L'oggetto della classe *ActionForward* viene restituito dal metodo *execute()* della action, utilizzando il metodo *findForward()* della classe *ActionMapping*. Passando a tale metodo il nome con cui l'*ActionForward* è mappato

nel file *struts-config.xml*, viene determinata la risorsa verso la quale inoltrare il flusso dell'applicazione. Un oggetto *ActionForward* può essere creato anche direttamente all'interno del codice, mediante il suo costruttore, specificando l'URL verso il quale eseguire il forwarding.

2.2.2 Le classi di utilità

Nell'ambito della realizzazione di una Web application, ci sono molto spesso una serie di operazioni che vanno ripetute più volte. Tale situazione si verifica anche con il framework Struts, il quale raggruppa tutte queste funzionalità in opportune classi di utilità, in modo da poter essere condivise fra più componenti ed utilizzate da più applicazioni.

Tali classi sono raggruppate in package separati e le principali sono le seguenti:

- *classe RequestUtils*: mette a disposizione una serie di metodi per gestire ciascuna richiesta pervenuta alla Web application;
- *classe ResponseUtils*: ha un ruolo simile alla classe precedente, con la differenza che è orientata alla gestione delle response;
- *commons-package BeanUtils*: per la gestione dei JavaBeans definiti all'interno dell'applicazione, in particolare per popolare i formbean con i dati immessi dall'utente ed accedere alle relative proprietà.

2.2.3 Componenti del Model

Nell'ambito del pattern MVC, il Model è caratterizzato da tutte le classi che costituiscono la *business logic* ed implementano le funzionalità offerte dall'applicazione.

Tali componenti devono essere completamente indipendenti dal tipo di framework che viene adottato, in virtù del fatto che in una architettura a livelli, i livelli superiori devono essere legati a quelli inferiori, ma non vale il viceversa.

Nel momento in cui, all'interno delle classi della *business logic*, utilizziamo elementi del framework, si viola tale modello e si determina un accoppiamento tra due livelli che devono essere indipendenti.

All'interno del Model, si definiscono i cosiddetti *business-object (BO)*, i quali non rappresentano nient'altro che delle astrazioni software delle entità del mondo reale. Affinché una classe possa essere considerata un business-object, è necessario che abbia le seguenti caratteristiche:

- mantenga uno stato e definisca un comportamento;
- rappresenti una entità del dominio del problema;
- sia riutilizzabile.

Strutturando il Model in questo modo, esso diventa assolutamente indipendente dal framework che verrà adottato per realizzare la Web application. In alcuni casi, si potrebbe pensare di utilizzarlo anche per lo sviluppo di un'applicazione desktop e non orientata al Web.

In relazione al framework Struts, quest'ultimo non vincola lo sviluppatore nel dover realizzare il Model con una particolare tecnologia. I principali strumenti adottati per la realizzazione dei business-object, sono tipicamente gli Enterprise JavaBeans (EJB) oppure i più semplici JavaBeans (JB), fortemente affermati nella programmazione Java.

Mediante questi strumenti, è possibili sviluppare le classi del Model definendone sia i dati (lo stato dell'applicazione) che i metodi (il comportamento). Infine, per garantire la persistenza delle informazioni, è necessario fare in modo che tali oggetti siano associati ad una base di dati. La mappatura tra i due elementi, può essere realizzata in due modi principali:

- *JDBC (Java DataBase Connectivity)*: si realizzano ulteriori classi che sfruttano le funzionalità dei driver JDBC per accedere alle basi di dati, in questo modo, ciascun oggetto della *business logic* utilizza l'oggetto corrispondente JDBC per garantirsi la propria persistenza;

- *ORM Frameworks*: si utilizzano dei particolari frameworks, detti *ORM (Object to Relational Mapping)*, i quali eseguono la mappatura in maniera automatica, tra i più importanti è da ricordare *Hibernate*.

2.2.4 Componenti della View

In generale, la View rappresenta la visualizzazione del Model dell'applicazione, secondo una certa interfaccia utente. Ciò vuol dire che possono esistere più View differenti a dispetto di un unico Model.

Tipicamente, nell'ambito di Struts, le differenti Views vengono realizzate attraverso le pagine JSP, sfruttando le diverse librerie di tag che il framework mette a disposizione. Un ruolo fondamentale, nell'interazione con l'utente, è giocato dalla classe *ActionForm*, che permette di definire i formbean mediante i quali vengono raccolti i dati che l'utente immette attraverso i form dell'applicazione.

2.2.4.1 ActionForm

Generalmente, ogni Web application prevede l'interazione con l'utente e quindi l'immissione, da parte di quest'ultimo attraverso dei form costituiti da uno o più campi, di dati su cui verranno eseguite le elaborazioni. Una volta che i dati vengono trasmessi, l'applicazione deve farsi carico di acquisirli, validarli ed in caso di errore visualizzare un messaggio all'utente ed infine di elaborarli.

Per poter eseguire tutte queste operazioni nel modo più semplice possibile, Struts mette a disposizione la classe *ActionForm*, la quale ha come compito principale quello di raccogliere i dati dai form riempiti dall'utente e metterli a disposizione di una action, per eseguirne l'elaborazione. Ulteriore funzionalità fornita da questa classe è la validazione mediante la quale è possibile controllare che i dati immessi dall'utente siano corretti ed, in caso di esito negativo, segnalare gli errori a quest'ultimo. In generale, non bisogna dichiarare un formbean per ogni form HTML previsto dall'applicazione, perché lo stesso formbean può essere condiviso ed associato a più

action. Ovviamente, è necessario definirne anche l'ambito di visibilità , che può essere di due tipi:

- *request*: i dati del formbean sono memorizzati fino al completamento del ciclo request-response, dopodichè viene cancellato;
- *session*: i dati del formbean sono disponibili durante tutta la sessione utente.

La classe *ActionForm* è però una classe astratta, per cui è necessario implementare una propria classe che la estenda per poter definire un formbean necessario all'interno dell'applicazione. Tale classe avrà al suo interno una serie di proprietà che corrispondono ai campi del form ed i relativi metodi getter e setter, per accedere ad esse. Inoltre, è necessario eseguire l'overriding dei due seguenti metodi:

- *reset()*: per poter resettare il contenuto del formbean;
- *validate()*: per poter eseguire la validazione dei dati contenuti nel form e restituire un oggetto ActionErrors con eventuali messaggi di errore, se la validazione non ha avuto esito positivo.

```
public void reset(ActionMapping mapping,
  HttpServletRequest request) {

  ...
}

public ActionErrors validate(ActionMapping mapping,
  HttpServletRequest request) {

  ...
}
```

Ogni formbean realizzato va, inoltre, dichiarato all'interno del file di configurazine XML ed assegnata, sempre all'interno di quest'ultimo, alle action che potranno eseguire sui dati le corrispondenti elaborazioni.

```
<form-bean name="nomeFormBean" type="package.ClasseActionForm"/>
```

E' da sottolineare, che la signature del metodo execute() di una action, prevede tra i parametri di ingresso, proprio il formbean con i dati su cui andranno eseguite le operazioni della *business logic*.

2.2.4.2 ActionErrors

In generale, i dati che vengono immessi dall'utente all'interno di un form della Web application, sono sottoposti ad un'operazione di validazione. L'esito di quest'ultima non è sempre positivo ed, in caso di errori, è necessario visualizzare all'utente dei messaggi di errore, per segnalare che i dati immessi non sono corretti. Tali errori vengono memorizzati nel sistema, mediante la classe *ActionErrors*. Tipicamente, nel momento in cui è richiesta la validazione dei dati, viene invocato il metodo *validate()* del formbean, derivato da *ActionForm*. Se durante la validazione si verificano uno o più errori, viene creato un oggetto *ActionErrors*, il quale altro non è che una lista, all'interno della quale possono essere salvati tali errori con i relativi messaggi da visualizzare all'utente. La classe prevede il metodo *add()*, mediante il quale si può aggiungere un messaggio ed associarlo ad un elemento della View, in cui verrà eseguita la visualizzazione.

Le classi che permettono di creare dei messaggi di semplice segnalazione o di errore all'utente sono: *ActionMessage* e *ActionError*.

ActionMessage - ActionError

Il framework Struts prevede la classe *ActionError*, per creare dei messaggi di warning o di errore da visualizzare all'utente. Dalla versione 1.1, è stata aggiunta la classe *ActionMessage*, per un motivo puramente concettuale. Infatti, facendo uso soltanto della classe *ActionError*, anche un messaggio qualsiasi, non di errore, verrebbe interpretato come tale, dallo sviluppatore che legge o crea il codice. Per poter distinguere un errore, da un semplice messaggio da visualizzare nella View, è stata introdotta la classe *ActionMessage*, la quale altro non è che una generalizzazione della *ActionError*. Ovviamente, è stata aggiunta anche la classe *ActionMessages* che, alla pari

della classe *ActionErrors*, si fa carico di memorizzare più messaggi all'interno di una lista.

2.2.4.3 DynaActionForm

Uno dei problemi che sorge nell'utilizzo della classe *ActionForm* per la realizzazione dei formbean, è il numero elevato di classi da creare e da aggiungere al proprio progetto software. Per questi motivo, è stata introdotta la possibilità di realizzare dei formbean dinamici facendo uso della classe *DynaActionForm*, la quale altro non è che un'estensione della classe *ActionForm*.

Le proprietà di un formbean dinamico sono definite all'interno del file di configurazione *struts-config.xml*. Durante l'esecuzione della Web application, il framework crea un'istanza della classe e mette a disposizione dei metodi getter e setter per accedere a tali proprietà. La modifica oppure l'introduzione di uno o più proprietà, prevede l'intervento sul file di configurazione e tutto ciò garantisce grande potenza e flessibilità.

```
<form-bean name="nomeFormBeanDinamico"
  type="org.apache.struts.validator.DynaValidatorForm">
  <form-property name="porprieta1" type="ClasseProprieta1"/>
  <form-property name="proprietaN" type="ClasseProprietaN"/>
</form-bean>
```

Infine, la validazione di un formbean dinamico potrebbe essere eseguita realizzando una classe che estenda *DynaActionForm* ed esegua l'overriding del metodo *validate()*. In generale, per evitare lo sviluppo di classi aggiuntive, si fa uso del plug-in Validator, il quale mette a disposizione una serie di regole di validazione già pronte per l'uso.

2.2.4.4 Tag Libraries

Nell'ambito della View, oltre alle classi che permettono di gestire tutto ciò che riguarda l'interazione e lo scambio dei dati tra utente ed applicazione, rientrano anche le librerie di tag che Struts mette a disposizione per la realizzazione della *UI (User Interface)*. Queste ultime, non soltanto offrono la

possibilità di introdurre gli elementi grafici in una pagina JSP, ma ne permettono anche la costruzione dinamica, oltre alla possibilità di accedere alle proprietà dei beans previsti dall'applicazione.

La libreria Struts HTML mette a disposizione tutti gli elementi che permettono di realizzare i form e che definiscono quella che è la grafica dell'applicazione.

La libreria Struts Bean dispone di tutti i tag per poter visualizzare e modificare i valori delle proprietà dei bean previsti dall'applicazione. Ciascun tag ha sempre gli attributi che permettono di identificare il bean e la proprietà di interesse, separatamente l'uno dall'altro.

Infine, la libreria Struts Logic fornisce una serie di tag che permettono di definire dei costrutti logici e condizionali all'interno di una pagina JSP, per poter costruire il contenuto di quest'ultima in maniera dinamica e sulla base di particolari condizioni.

2.3 Flow Diagram di Struts

Il diagramma di flusso di Struts per l'effettuazione di una richiesta ha una struttura standard, tipica del pattern MVC. I passi che vengono eseguiti hanno un iter di questo tipo:

1. Il client invia una richiesta http tramite il proprio browser;

2. La richiesta viene ricevuta dalla servlet di controllo di Struts, ovvero l'*ActionServlet*, che è stata configurata nel file di configurazione dell'applicazione *web.xml* per accettare richieste di un certo tipo. L'*ActionServlet* provvede a popolare l'*ActionForm*, se presente, associato alla richiesta con i dati della request e l'*ActionMapping* con gli oggetti relativi alla *Action* associata alla richiesta. Tutti i dati di configurazione per Struts sono stati letti in fase di start-up dell'applicazione dal file *struts-config.xml*;

3. La *ActionServlet* delega l'elaborazione della richiesta alla *Action* associata al path della richiesta passandole in input la request http, l'*ActionForm* e l'*ActionMapping* precedentemente valorizzati;

4. La *Action* si interfaccia con lo strato di business che implementa la logica applicativa. Al termine dell'elaborazione restituisce alla *ActionServlet* un *ActionForward* contenente l'informazione del path della vista da fornire all'utente;

5. La *ActionServlet* esegue il forward alla pagina JSP specificata nell'*ActionForward*, tale pagina JSP può utilizzare le librerie di tag fornite da Struts per la visualizzazione delle informazioni.

Ovviamente il flusso di operazioni elencato non è completo ma fornisce una indicazione di base su come viene gestito il flusso elaborativo in un'applicazione sviluppata con Struts.

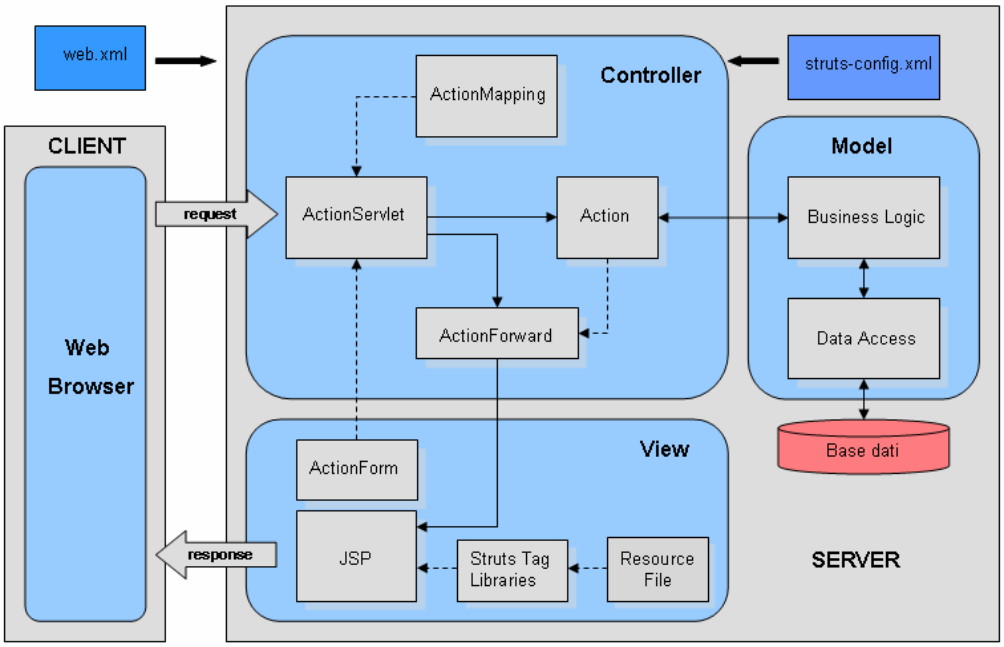

Figura 2.4: Flow Diagram di Struts

2.4 Gestione delle eccezioni

Prima delle versione 1.1, il framework Struts forniva un meccanismo di gestione delle eccezioni piuttosto semplice, che spingeva gli sviluppatori a realizzare soluzioni differenti per poter gestire le condizioni di errore nelle proprie applicazioni. Dalla versione 1.1, è stato introdotto un meccanismo non molto ampio ma abbastanza efficiente, che permette di gestire le eccezioni in due modalità:

- dichiarativa: le eccezioni, che possono essere sollevate nell'ambito dell'applicazione, sono dichiarate all'interno del file di configurazione e gestite in maniera automatica dal framework;
- programmatica: le eccezioni sono gestite direttamente nel codice delle action, nella modalità consueta del linguaggio Java.

I due tipi di approcci non sono mutuamente esclusivi, ma viceversa possono essere utilizzati contemporaneamente.

2.4.1 ModuleException

Il framework mette a disposizione una particolare classe, nota come *ModuleException*, per sollevare delle eccezioni che possano poi essere gestite mediante l'approccio dichiarativo. Infatti, all'interno del metodo *execute()* di una action, qual'ora si verificasse una condizione di errore, è possibile sollevare un'eccezione di questo tipo, che in maniera completamente automatica crea l'oggetto *ActionError* nel quale è possibile specificare il messaggio da visualizzare all'utente.

```
ModuleException me = new ModuleException("chiaveMsgErrore");
throw me;
```

Tipicamente, tale eccezione sarà dichiarata all'interno del file di configurazione, per cui sarà il *RequestProcessor* che si occuperà di catturarla e delegarne la gestione alla classe *ExceptionHandler*.

```
<global-exceptions>
  <exception bundle="ApplicationResources"
    key="chiaveMessaggioErrore"
    path="/paginaErrore.jsp"
    type="org.apache.struts.util.ModuleException"/>
</global-exceptions>
```

L'unico limite nell'utilizzo di questa classe è di determinare un accoppiamento tra l'applicazione e la modalità di gestione delle eccezioni di Struts, rendendo più complesso il procedimento di sostituzione del framework adottato, lasciando inalterato lo strato applicativo sottostante.

2.5 Internazionalizzazione (I18N)

Tipicamente, gli sviluppatori focalizzano la loro attenzione sulla realizzazione di un'applicazione per la risoluzione di un certo problema. Nella maggior parte dei casi, viene perso di vista un aspetto fondamentale che riguarda il pubblico a cui è diretto l'utilizzo dell'applicazione stessa. Soprattutto nel caso di Web application fruibili attraverso Internet, si pone il problema di permettere l'utilizzo di quest'ultima, a persone di paesi e lingue differenti.

L'Internazionalizzazione (I18N) è un processo che prevede per lo sviluppo di un'applicazione la capacità di poter supportare nel tempo più lingue e stati diversi, in modo tale che non ci sia bisogno di reingegnerizzare il sistema nel momento in cui sarà necessario fornire supporto per ulteriori di essi.

Le caratteristiche principali di un'applicazione che supporta l'Internazionalizzazione sono le seguenti:

- le lingue devono essere supportate senza la necessità di apportare modifiche al codice;
- i messaggi da visualizzare agli utenti sono contenuti in risorse esterne e completamente al di fuori del codice.

Il framework Struts mette a disposizione tali caratteristiche utilizzando

le principali classi Java che si occupano dell'internazionalizzazione: *Locale* e *ResourceBundle*.

La classe *Locale* contiene le informazioni relative appunto al *locale* associato all'utente che utilizza l'applicazione. Con il termine locale si intende una regione nell'ambito della quale sono condivisi costumi, culture e lingue. Attraverso tale classe, è possibile risalire alle informazioni relative alla valuta, alla formattazione delle date e delle ore ed a tante altre informazioni specifiche.

La classe *ResourceBundle* permette, invece, di definire la risorsa esterna all'interno della quale sono salvati i messaggi da visualizzare all'utente in una certa lingua. Attraverso l'utilizzo di questa classe, è possibile accedere ad essi direttamente da codice, mediante un meccanismo *key-message (chiave-messaggio)*, in base al quale, specificando una chiave, si ricava il messaggio corrispondente associato.

Generalmente, quando un utente accede ad una Web application realizzata con Struts, quest'ultimo memorizza all'interno della sessione utente, il locale definito dal browser che quest'ultimo sta adottando. Mediante questa informazione, è quindi possibile accedere al giusto Resource Bundle, da cui prelevare i messaggi da visualizzare all'utente nella sua lingua.

Struts prevede la definizione di uno o più file di proprietà *.properties*, aventi lo stesso nome ma un suffisso diverso, che specifica la lingua adottata nei messaggi contenuti nel file, attraverso le coppie key-message.

2.6 Estensioni con i PlugIn

Il framework Struts mette a disposizione un particolare meccanismo mediante il quale è possibile estenderne la struttura, ossia la definizione e l'uso del plug-in. Basti pensare che gli stessi Tiles e Validator sono dichiarati come tali all'interno del file di configurazione, in quanto non sono parte integrante dell'architettura di Struts. Ovviamente, lo sviluppatore ha la possibilità di realizzare una classe che implementi l'interfaccia *PlugIn* per poter disporre di

un proprio plug-in con cui estendere le funzionalità dell'applicazione sviluppata. L'uso di un plug-in può essere particolarmente conveniente quando è necessario effettuare una serie di elaborazioni una sola volta, nella fase di start-up dell'applicazione. Infatti, il framework effettua il caricamento dei plug-in configurati nella fase di inizializzazione della Web application, così come la corrispondente eliminazione nel caso di shutdown oppure riavvio del Web Container.

2.7 Il framework Validator

Per poter eseguire la validazione dei dati immessi dall'utente all'interno dei form, il framework Struts prevede che la classe *ActionForm*, utilizzata per definire un formbean, abbia il metodo *validate()* che viene eseguito subito dopo l'invio del form stesso. Un approccio di questo tipo è, ovviamente, programmatico e talvolta può presentare una serie di limitazioni.

In primo luogo, c'è da dire che la maggior parte dei form presenti in una Web application, hanno dei campi che possono contenere dei dati soggetti allo stesso tipo di validazione, per cui si è costretti a realizzare più metodi *validate()* praticamente uguali.

Nel momento in cui c'è la necessità di effettuare una modifica, quest'ultima andrà eseguita in tutti i metodi realizzati e comporterà la ricompilazione del codice. Per questi motivi, è stato implementato un particolare framework che permette di gestire la validazione dei dati in maniera dichiarativa. Dato il suo notevole utilizzo, da parte degli sviluppatori, si è deciso di considerarlo un progetto Jakarta ed introdurlo in ogni distribuzione di Struts.

Il framework Validator permette di trasferire la logica della validazione completamente fuori dalle classi *ActionForm*, configurando in maniera dichiarativa tutte le regole di validazione in un file XML. Esso fornisce una serie di funzioni standard per le validazioni più comuni ma è completamente estendibile, in quanto permette allo sviluppatore di realizzare e configurare dei metodi di validazione personalizzati.

La distribuzione del framework prevede una serie di classi ed in particolare due file XML, che ne permettono la configurazione in maniera semplice:

- *validation-rules.xml*;
- *validation.xml*.

Il file *validation-rules.xml* contiene una serie di funzioni che permettono di eseguire le validazioni tipiche. Ovviamente, a queste funzioni incluse nel framework ne possono essere aggiunte delle ulteriori, estendendo le classi principali del framework stesso.

Il file *validation.xml* permette di configurare tutti i form della Web application, i cui campi debbano essere soggetti a validazione e, per ogni form, specificare per ciascun campo la funzione di validazione da applicare. Il tag <*formset*> viene utilizzato per raggruppare tutti i form suddetti, ciascuno dei quali viene definito mediante il tag <*form*>. Inoltre, all'interno di quest'ultimo sono previsti uno o più tag <*field*> che permettono di specificare i campi che costituiscono il form in questione. Per ciascuno di essi è possibile specificarne il nome, mediante l'attributo *property* e la regola di validazione da applicare, mediante l'attributo *depends*.

```
<form name="nomeForm">
  <field depends="required" property="proprieta1"/>
  <field depends="required,minlength" property="proprietaN">
</form>
```

Generalmente, i form e le relative proprietà specificate all'interno di questo file, devono coincidere con le informazioni dei corrispondenti formbean che sono configurati all'interno del file *struts-config.xml* dell'applicazione.

L'ultima considerazione da fare è legata al fatto che il framework Validator è perfettamente integrato con Struts ed il suo utilizzo prevede che esso venga dichiarato all'interno del file di configurazione come se fosse un plug-in, specificando i percorsi dei file XML che lo caratterizzano.

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
  <set-property property="pathnames"
    value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"/>
</plug-in>
```

2.8 Il framework Tiles

Tiles è un framework mediante il quale è possibile realizzare il livello di presentazione di una Web application, separando il layout dai contenuti (*contents*). Mentre il layout definisce la struttura di una pagina, con la caratteristica che più pagine possono avere il medesimo layout, il contents definisce il contenuto di ciascuna di esse e può essere diverso da pagina a pagina oppure eventualmente uguale in alcuni casi. E' notevolmente importante poter separare l'uno dall'altro, in modo tale che apportando delle modifiche al primo non sussiste alcuna influenza sul secondo e viceversa.

Il framework Tiles permette di costruire le pagine assemblando i cosiddetti *tiles*, il cui significato è banalmente mattonelle evidenziando che sono dei componenti riutilizzabili che possono essere disposti in una struttura predefinita. Nella maggior parte dei casi, ogni tile non è nient'altro che una pagina JSP che a sua volta può essere composta da altri tiles. E' possibile distinguerne due tipologie:

- *tile Layout*: tile che definisce la struttura di una pagina, esso riceve da una pagina (tile non-Layout) il contenuto e lo dispone secondo tale struttura;
- *tile non-Layout*: tile che utilizza un layout (tile Layout) passando a quest'ultimo degli attributi che rappresentano il contenuto da disporre secondo una certa struttura.

Generalmente, nell'ambito di una Web application, il tile Layout è unico e definisce la struttura di tutte le pagine, mentre ciascuna pagina è definita attraverso un proprio tile non-Layout.

L'uso dei tiles può essere paragonato all'uso dei metodi in Java. Un metodo in Java è costituito da un body che ne definisce l'elaborazione e dai parametri di ingresso che specificano i dati su cui quest'ultima va effettuata. Un tile può essere considerato come un metodo, in quanto bisogna definirne la struttura e passargli dei parametri, detti attributi, che ne descrivano il contenuto. Ciascuno di essi è memorizzato nel context del tile stesso, attraverso la classe *ComponentContext* facente parte del framework. Infine, ogni attributo può essere una stringa oppure tipicamente una pagina JSP.

Nella maggior parte dei casi, il layout utilizzato per ciascuna pagina è proposto in figura.

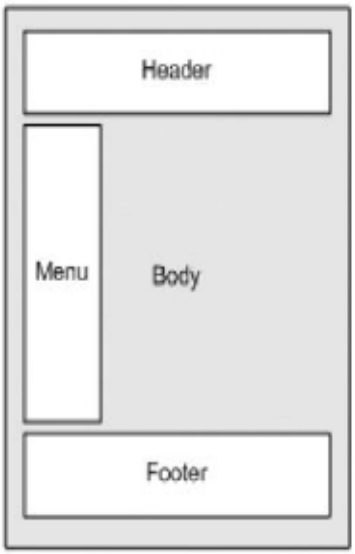

Figura 2.5: Layout Header/Footer/Menu/Body

Prevede fondamentalmente quattro parti: *header*, *footer*, *menu* e *body*.

Tipicamente, l'header ed il footer sono comuni a tutte le pagine, per cui il loro contenuto viene definito un'unica volta. Il vantaggio che ne scaturisce è dato dal fatto che se deve essere apportata una modifica ad uno di essi, quest'ultima va eseguita un'unica volta e non per tutte le pagine della Web application. Il menu, invece, può essere lo stesso nell'ambito di tutta l'applicazione e quindi avrà il medesimo vantaggio di header e footer oppure può

variare in relazione alla pagine visitate. Infine, il body è tipicamente diverso da pagina a pagina.

Per quanto riguarda la relazione con Struts, Tiles è perfettamente integrato con quest'ultimo, in quanto contenuto in ciascuna distribuzione, e viene dichiarato all'interno del file di configurazione in termini di plug-in.

```
<plug-in className="org.apache.struts.tiles.TilesPlugin">
  <set-property property="definitions-config"
    value="/WEB-INF/tiles-defs.xml"/>
  <set-property property="moduleAware" value="true"/>
</plug-in>
```

2.8.1 Definitions

I tiles possono essere descritti attraverso le cosiddette definitions, dichiarate in un file XML e che permettono:

- una dichiarazione centralizzata di una pagina;
- se si dispone di più tiles non-Layout con un contenuto uguale (es. le pagine hanno il medesimo header e footer), al posto di definire tale contenuto pagina per pagina, lo si definisce un'unica volta rendolo comune a tutte le pagine;
- supporto alla derivazione, descrivendo una nuova definition che ne estenda una preesistente, ereditandone le caratteristiche ed aggiungendone delle altre.

Generalmente, il file che contiene le definitions è noto come *tiles-defs.xml* e ciascuna di esse è dichiarata attraverso il tag <*definition*> che mediante i suoi attributi permette di specificare le seguenti informazioni:

- nome della definition per referenziarla all'interno di un tile che ne farà uso (*name*);
- percorso del tile Layout che descrive la struttura associata alla definition (*path*);

- definition da cui eseguire la derivazione (*extends*).

Per specificare gli attributi della definition, è possibile utilizzare uno o più tag <*put*>, ciascuna dei quali prevede le seguenti informazioni:

- nome dell'attributo;
- valore dell'attributo che può essere una stringa, una pagina JSP oppure un'altra definition se si prevede di utilizzare la composizione di più definitions.

```
<tiles-definitions>
  <definition name="page.template" path="/paginaLayout.jsp"/>
  <definition name="page.header" path="/header.jsp"/>
  <definition name="page.footer" path="/footer.jsp"/>
  <definition name="page.menu" path="/menu.jsp"/>
</tiles-definitions>
```

2.8.2 Costruzione di una pagina

La costruzione di una pagina JSP secondo un certo layout e con uno specifico contenuto è realizzata attraverso l'utilizzo di particolari tag della *Struts Tiles Library*. Il tag che permette di inserire un contenuto oppure utilizzare una definition all'interno di una pagina JSP è <*tiles:insert*>, il quale prevede le seguenti informazioni:

- *page*: percorso della pagina da inserire;
- *attribute*: nome dell'attributo;
- *definition*: nome della definition da adottare all'interno di un tile non-Layout.

Ovviamente, la definition utilizzata all'interno di ciascuna pagina avrà dei contenuti comuni fra le pagine ma queste ultime potrebbero anche avere alcuni contenuti differenti. Per inserire un contenuto nel layout, specificato da una definition, è possibile utilizzare il tag <*tiles:put*>, il quale descrive le seguenti informazioni:

- *name*: nome dell'attributo;
- *value*: valore dell'attributo che può essere una stringa oppure una pagina JSP.

```
<tiles:insert definition="page.template">
  <tiles:put name="title" value="Titolo"/>
  <tiles:put name="body"
    value="/BodyContext.jsp" type="page"/>
</tiles:insert>
```

Attraverso questo meccanismo, le pagine possono avere contenuti comuni che non devono ulteriormente specificare in quanto definiti all'interno della definition utilizzata. Nel caso in cui alcuni contenuti debbano essere differenti, basta utilizzare il tag <*tiles:put*> mediante il quale il contenuto specificato va a sostituire quello di default.

2.9 Sicurezza con SSL

La sicurezza di un'applicazione realizzata con il framework Struts rappresenta un aspetto completamente indipendente da quest'ultimo e può essere gestita mediante le funzionalità offerte dal Web Container. Parlare di sicurezza attraverso la rete significa soprattutto garantire la trasmissione delle informazioni e dei dati in modalità crittografata, in modo tale da essere inaccessibili dall'esterno.

Il meccanismo principale messo a disposizione si basa sul protocollo *SSL (Secure Socket Layer)*, sulla base del quale è stata creata in passato una nuova versione del protocollo HTTP, ossia l'HTTPS. Mediante quest'ultimo, infatti, le informazioni sono trasmesse dal client al server e viceversa in modalità crittografata, anche sulla base dell'utilizzo di un regolare certificato digitale.

Gli elementi principali che permettono un approccio di questo tipo sono fondamentalmente due:

- l'acquisizione di un certificato digitale;

- uso del protocollo HTTPS, ossia HTTP basato su SSL.

Attraverso un certificato digitale, rilasciato dalla *CA (Certification Authority)*, è sempre possibile essere a conoscenza dell'identità dell'interlocutore e decidere se accettare o meno di stabilire una comunicazione con qeust'ultimo. Uno strumento di questo tipo risulta notevolmente utile, in virtù del fatto che ci si potrebbe ritrovare a scambiare informazioni con un'entità sulla rete che non è quella da noi attesa, ma che ne abbia preso il posto in maniera fraudolenta. Chiunque sia in possesso di un certificato digitale è stato riconosciuto da terze parti e quindi si ha la certezza che i dati trasmessi verranno acquisiti dal destinatario corretto. Una delle funzionalità che è possibile sfruttare mediante i certificati digitali è la crittografia, mediante la quale è possibile cifrare i dati trasmessi e renderli illeggibili durante il loro invio. Soltanto il destinatario, riconosciuto sulla base di un certificato digitale, avrà la possibilità di decrittografarli attraverso l'utilizzo di una chiave.

La comunicazione avviene attraverso l'uso del protocollo HTTPS, ossia il tipico HTTP basato su SSL, che garantisce una cifratura con chiave a 128 bit.

Per poter fare in modo che le pagine di un'applicazione implementata con Struts, utilizzino tale protocollo, è necessario in primo luogo abilitarlo all'interno del Web Container e successivamente modificare il Deployment Descriptor *web.xml* dell'applicazione stessa, andando a specificare le pagine la cui trasmissione deve essere effettuata su protocollo HTTPS. Inoltre sono necessarie ulteriori classi che implementano la sicurezza, le quali forniscono un'estensione di Struts per il protocollo SSL. Tali classi sono contenute all'interno della libreria *sslext.jar*, la quale deve essere posta tra le librerie dell'applicazione in *WEB-INF/lib*.

2.10 Logging con log4j

E' sempre più importante, nelle applicazioni odierne, tenere traccia di tutto ciò che succede nell'applicazione, sia in automatico sia su richiesta dell'utente. Tutto questo, nel mondo dell'informatica, prende il nome di logging. Il logging si basa sulla generazione di messaggi di log che possono aiutare a individuare i difetti presenti nelle proprie applicazioni e a svolgere altre funzionalità importanti. Ad esempio, la sicurezza e il controllo possono dipendere dal logging per raccogliere informazioni per gli amministratori di sistema riguardo a cosa stanno facendo gli utenti autorizzati e sopratutto quelli non autorizzati. Con informazioni in tempo reale circa potenziali attacchi alla sicurezza dell'applicazione, il logging può dare un margine molto utile agli amministratori di sistema e permettere reazioni più rapide.

Struts fornisce il supporto per poter utilizzare diverse implementazioni di logging, ma una delle più popolari implementazioni di logging in uso è *log4j*.

Anche log4j, come Struts, è un progetto Open Source che fa parte dell'insieme di progetti Jakarta. Essenzialmente è una raccolta di classi ed interfacce Java, che offrono funzioni di logging per svariati tipi di destinazione. Tali classi e interfacce sono contenute all'interno della libreria *log4j.jar*, la quale deve essere posta tra le librerie dell'applicazione in *WEB-INF/lib*.

In log4j la classe principale è la classe *Logger*, che svolge la maggior parte delle funzionalità.

Con log4j si possono mandare messaggi di log verso varie destinazioni, indicate con il termine *appender*. Inoltre esso definisce cinque livelli di priorità per i messaggi di log. I livelli permettono di impostare un limite per un logger particolare scartando tutti i messaggi di log che non lo raggiungono. I cinque livelli sono: DEBUG, INFO, WARN, ERROR e FATAL.

Infine, le proprietà di configurazione di log4j possono essere inizializzate in diversi modi, due di questi sono:

- inizializzazione tramite un file di proprietà;
- inizializzazione tramite un file XML.

Il primo modo riguarda la creazione di un file di proprietà *log4j.properties* che contenga i necessari elementi di configurazione per le nostre necessità di logging. Tale file di proprietà deve essere posto nella directory *WEB-INF/classes*, in modo tale che log4j possa trovarlo e utilizzarlo per l'ambiente di logging della Web application. Infine, il formato degli elementi definiti all'interno del file di proprietà deve essere di tipo *key-value (chiave-valore).*

Nel secondo approccio il problema viene risolto con la creazione di un file XML. Al solito, si porrà il file XML nella directory *WEB-INF/classes.*

2.11 Configurazione dell'applicazione

Il framework Struts utilizza uno o più file di configurazione per poter creare e caricare in fase di start-up dell'applicazione, tutte le risorse ed i componenti necessari a quest'ultima. Tali file permettono di specificare in maniera dichiarativa il comportamento degli elementi dell'applicazione, evitando che tali specifiche siano definite all'interno del codice. Questo permette agli sviluppatori di realizzare delle estensioni e di utilizzarle nella Web application, garantendo che il framework sia capace di sfruttarle in maniera dinamica. I file di configurazione si basano sul linguaggio XML e possono essere anche più di uno, nel momento in cui l'applicazione è realizzata con moduli indipendenti. Ciascuno di essi ha le proprie informazioni di configurazione, inclusi i Resource Bundles ed è completamente indipendente dagli altri. Tutto ciò favorisce lo sviluppo di un'applicazione di grandi dimensioni, anche da team di lavoro distinti.

Con la versione 1.1 del framework Struts, è stato introdotto il package *config*, all'interno del quale ci sono una serie di classi che permettono di memorizzare le informazioni di configurazione, nel momento in cui queste ultime vengono lette dal file XML. In questo modo, si ha a disposizione una visione di queste informazioni in termini di classi, residenti nella memoria e quindi accessibili dallo sviluppatore direttamente da codice.

Ciascuna classe del package contiene le informazioni che riguardano una

parte specifica del file di configurazione. Dopo che è stato eseguito il parsing e la validazione del file di configurazione, il framework istanzia ed alloca in memoria, gli oggetti contenenti le informazioni ad essi pertinenti.

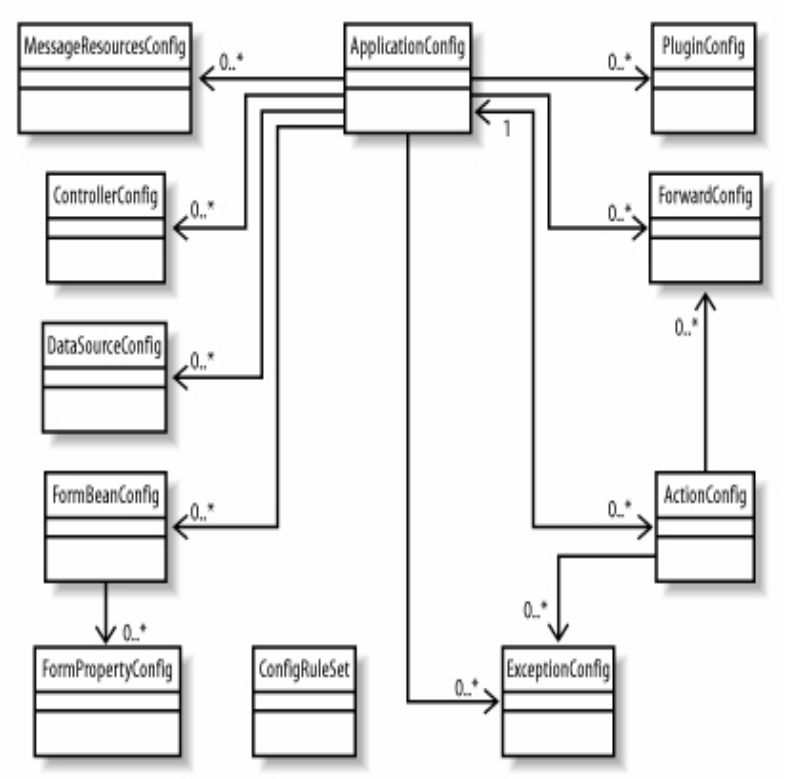

Figura 2.6: Class Diagram Configurazione

La classe centrale del package è l'*ApplicationConfig*, la quale è legata a tutte le altre classi contenenti le informazioni di configurazione. Attraverso l'istanza di tale classe, è possibile accedere a tutte le altre e quindi alle informazioni in esse memorizzate. Se l'applicazione è sviluppata in moduli, ciascun modulo ha a sua disposizione una corrispondente istanza di tale classe.

2.11.1 DataSource

Generalmente, una Web application prevede l'utilizzo di un database all'interno del quale sono memorizzate le informazioni in maniera permanente.

L'utilizzo di una base di dati, oppure più generalmente di un datasource, può essere specificato all'interno del file di configurazione. Per questo scopo, è messo a disposizione il tag <*data-source*>, che va utilizzato più volte in relazione al numero di database di cui dispone l'applicazione, distinguendo un datasource dall'altro attraverso la specifica di una chiave (*key*). Inoltre, è possibile definire una serie di proprietà caratteristiche del datasource, utilizzando al suo interno uno o più tag <*set-property*>. Le informazioni tipicamente configurate sono le seguenti:

- una descrizione del datasource;
- la classe che implementa il driver per l'accesso alla base di dati. In generale, sono utilizzati i driver JDBC, ma in molti casi si può fare anche uso di driver specifici per il connection-pooling;
- l'username e la password per accedere alla base di dati;
- l'URL in cui si trova il database.

Tutte queste informazioni sono memorizzate, in fase di start-up dell'applicazione, nella classe *DataSourceConfig*, mediante la quale è possibile accedere alle proprietà del datasource ed eventualmente modificarle, direttamente da codice.

```
<data-source key="database">
  <set-property property="driverClassName"
    value="com.mysql.jdbc.Driver"/>
  <set-property property="url"
    value="jdbc:mysql://localhost/database"/>
  <set-property property="username" value="admin"/>
  <set-property property="password" value="pass"/>
</data-source>
```

2.11.2 FormBean

Ciascun formbean realizzato come estensione della *ActionForm* ed utilizzato all'interno dell'applicazione, deve essere specificato nel file di configurazione mediante l'uso del tag <*form-bean*>.

Quest'ultimo permette di specificare le seguenti informazioni :

- *name*: il nome del formbean mediante il quale si farà riferimento ad esso nel resto del file; ad esempio, quando si definisce una action che utilizza i dati di un certo formbean, va specificato il nome di quest'ultimo;
- *type*: la classe che implementa il formbean; tipicamente, questa può essere la classe *ActionForm* oppure *DynaActionForm* o eventualmente una delle classi che derivano da esse; in alcuni casi, è invece una classe realizzata dallo sviluppatore, che comunque estende una delle classi suddette.

```
<form-bean name="nomeFormBean" type="package.ClasseActionForm"/>
```

E' ovvio che, nel caso in cui la classe che implementa il formbean sia stata realizzata dallo sviluppatore, le proprietà del formbean stesso sono esattamente le proprietà della classe, che non devono essere ulteriormente specificate nel file di configurazione. Nel caso in cui, si preferisce adottare ad esempio un formbean dinamico, è necessario specificare le proprietà del formbean all'interno del file *struts-config.xml*, non essendoci una classe realizzata ad hoc.

Per questo scopo, il tag <*form-bean*> può contenere uno o più tag <*form-property*>, ciascuno dei quali permette di definire una proprietà del formbean, specificandone le seguenti informazioni:

- *name*: il nome della proprietà;
- *type*: la classe (oppure il tipo di dato primitivo) della proprietà;
- *initial*: l'eventuale valore iniziale, quando il formbean è vuoto.

```
<form-bean name="nomeFormBeanDinamico"
  type="org.apache.struts.validator.DynaValidatorForm">
  <form-property name="porprieta1" type="ClasseProprieta1"/>
  <form-property name="proprietaN" type="ClasseProprietaN"/>
</form-bean>
```

Tutte queste informazioni sono memorizzate all'interno delle classi *FormBeanConfig* e *FormPropertyConfig*, che contengono rispettivamente tutto ciò che riguarda in generale il formbean e le relative proprietà.

2.11.3 Global Exceptions

Il framework Struts mette a disposizione la possibilità di gestione delle eccezioni che possono essere sollevate all'interno delle action. Ovviamente, è possibile definire, per ciascuna action, le eventuali eccezioni che essa può sollevare, ma in moltissimi casi, esistono delle eccezioni che possono essere sollevate da più action diverse. In questi casi, la soluzione migliore non è certamente quella che prevede di specificare la stessa eccezione per ogni action, ma bensì di definire una eccezione come globale. In questo modo, quando una action solleva un'eccezione, l'*ExceptionHandler* valuta in primo luogo se essa è specifica della action che l'ha sollevata oppure è di tipo globale e va gestita in un certo qual modo.

Ciascuna eccezione globale viene specificata nel file di configurazione, all'interno del tag <*global-exceptions*>, facendo uso del tag <*exception*>, il quale permette di definire le seguenti informazioni:

- *key*: la chiave relativa al Resource Bundle e corrispondente al messaggio da visualizzare, quando l'eccezione viene sollevata;
- *path*: il percorso della pagina alla quale l'utente verrà redirezionato;
- *handler*: la classe handler che si occuperà di gestire l'eccezione; per default, essa è la classe *ExceptionHandler*, ma ovviamente lo sviluppatore può realizzarne una propria estendendo quest'ultima;
- *type*: la classe che rappresenta il tipo di eccezione.

La classe che memorizza tutte queste informazioni è la *ExceptionConfig*.

```
<global-exceptions>
  <exception bundle="ApplicationResources"
    key="chiaveMessaggioErrore"
    path="/paginaErrore.jsp"
    type="org.apache.struts.util.ModuleException"/>
</global-exceptions>
```

2.11.4 Global Forwards

Quando una action, all'interno del metodo *execute()*, termina le operazioni relative al proprio scopo, essa restituisce un oggetto *ActionForward* con il quale si definisce la risorsa verso la quale proseguire la navigazione. Per ciascuna action, all'interno del file di configurazione, vengono specificati i *forward* ad essa associati. In alcuni casi, può essere utile fare uso dei forward globali, i quali non sono specifici di una action, ma possono essere referenziati ad una qualsiasi action all'interno dell'applicazione.

Ciascun forward globale viene specificato nel file di configurazione, all'interno del tag <*global-forwards*>, facendo uso del tag <*forward*>, il quale permette di definire le seguenti informazioni:

- *name*: il nome del forward per referenziarlo all'interno di una Action;
- *path*: il percorso della risorsa fisica a cui fa riferimento il forward logico.

Queste informazioni sono memorizzate nella classe *ForwardConfig*.

```
<global-forwards>
  <forward name="nomeForward" path="pagina.jsp"/>
</global-forwards>
```

2.11.5 Action Mapping

Il contenuto principale del file di configurazione di Struts è il mapping delle action, attraverso il quale è possibile specificare tutte le action utilizzate all'interno del framework, associarle ad un percorso logico per la loro invocazione e specificarne i forward per proseguire la navigazione da ciascuna di esse.

Per definire ciascuna action, viene utilizzato il tag <*action*>, mediante il quale sono descritte le seguenti informazioni:

- *path*: il percorso (URI) della action per poter essere invocata da un qualsiasi punto dell'applicazione;
- *type*: la classe che implementa la action, tipicamente realizzata dallo sviluppatore, estendendo la classe di base *Action* oppure una delle sue derivate;
- *name*: il nome del formbean, i cui dati saranno oggetto dell'elaborazione della action;
- *scope*: la visibilità del formbean, distinguendo fra request e session;
- *input*: la pagina alla quale ritornare il controllo, qualora si verificassero errori di validazione dei dati nel formbean;
- *parameter*: un eventuale parametro che permette di specificare quale metodo della action eseguire nel caso in cui si tratti di una *DispatchAction*.

Inoltre, attraverso l'attributo *validate*, è possibile specificare se, prima di invocare la action, debba essere eseguita o meno la validazione dei dati contenuti nel formbean associato.

Infine, al termine dell'esecuzione del metodo execute(), la action restituisce l'oggetto *ActionForward* che specifica la risorsa verso la quale proseguire la navigazione. Attraverso uno o più tag <*forward*>, inclusi nel tag

<*action*>, è possibile specificare tutti i forward che può restituire la action stessa. Le informazioni principali sono tipicamente le seguenti:

- *name*: il nome del forward per referenziarlo all'interno di una action;
- *path*: il percorso della risorsa fisica a cui fa riferimento il forward logico.

Inoltre, è possibile specificare anche le eccezioni che possono essere sollevate da ciascuna action, facendo uso del tag <*exception*> così come vengono definite le eccezioni globali.

Tutte le informazioni descritte vengono memorizzate nella classe *ActionConfig*, in stretta relazione con le classi *ForwardConfig* ed *ExceptionConfig*.

```
<action input="pagina.jsp"
  name="nomeFormBean" parameter="dispatch"
  path="/nomeAction" scope="request"
  type="package.ClasseAction">
  <forward name="nomeForward1" path="pagina1.jsp"/>
  <forward name="nomeForwardN" path="paginaN.jsp"/>
</action>
```

2.11.6 Controller

Il framework, nell'ambito della gestione di tutte le richieste che arrivano alla Web application, utilizza le versioni di default per le classi *ActionServlet* e *RequestProcessor*, che realizzano il Controller. Nel momento in cui lo sviluppatore decide di realizzare un Controller personalizzato, non fa altro che estendere la classe *RequestProcessor* e definire al suo interno le elaborazioni da eseguire. Nel file di configurazione, va specificato quale deve essere il Controller da adottare, facendo uso del tag <*controller*>, mediante il quale, tra le altre informazioni, va specificata in particolare la classe che implementa il Controller, la quale può essere quella di default *RequestProcessor* oppure una classe che derivi da quest'ultima (*processorClass*).

Le informazioni riguardanti il Controller sono memorizzate nella classe *ControllerConfig*.

```
<controller processorClass="controller.CustomRequestProcessor"/>
```

2.11.7 Message resources

Per garantire l'internazionalizzazione, il framework Struts mette a disposizione la possibilità di realizzare una serie di file, all'interno di ciascuno dei quali salvare i messaggi da visualizzare all'utente, in una lingua specifica. Tali file sono anche noti come Resource Bundles o Message Resources e vanno specificati nel file di configurazione. In realtà, non è necessario definire l'elenco di tutti questi file, ma unicamente il file associato al linguaggio di default, poiché tutti gli altri avranno esattamente lo stesso nome, con in più un suffisso che ne distingue la lingua utilizzata. Per questo scopo, viene adottato il tag <*message-resources*>, nel quale è possibile specificare il nome del Resource Bundle, mediante l'attributo parameter.

Tali informazioni sono memorizzate nella classe *MessageResourcesConfig*.

```
<message-resources parameter="ApplicationResources"/>
```

2.11.8 Plug-In

Dalla versione 1.1 del framework, è stato introdotta la possibilità di utilizzare dei plug-in, mediante i quali estendere le potenzialità di quest'ultimo. I due esempi tipici sono il Validator Plug-In, per la validazione dei dati, ed il framework Tiles, per la definizione del layout dell'applicazione. Lo sviluppatore ha comunque la possibilità di realizzare una classe che estenda l'interfaccia *PlugIn* del framework, per implementare un proprio plug-in che venga eseguito in fase di start-up dell'applicazione. E' ovvio che ciascun plug-in utilizzato va specificato nel file di configurazione, facendo uso del tag <*plug-in*>, nel quale, tra le altre informazioni, va segnalata principalmente la classe che estende l'interfaccia *PlugIn* ed implementa il plug-in (*className*).

Tutte le informazioni relative ad un plug-in sono memorizzate nella classe *PlugInConfig*.

```
<plug-in className="package.ClassePlugIn"/>
```

Capitolo 3

Analisi e Progettazione

3.1 Introduzione

E' stata realizzata un'applicazione Web per il commercio elettronico, dal nome NetShop, che consente l'utilizzo e la gestione di un centro commerciale virtuale.

NetShop ingloba una serie di negozi dando da una lato la possibilità ai clienti di poter fare degli acquisti e da un altro ai gestori di poter creare e gestire il proprio negozio. Inoltre è presente un amministratore, per l'amministrazione di tutti i clienti, i gestori ed i loro negozi.

Le fasi di sviluppo hanno seguito il tipico ciclo di vita del software secondo il *Modello a cascata (Waterfall Model)*, che prevede le seguenti fasi:

- *Studio di Fattibilità*;
- *Analisi e Specifica dei Requisiti*;
- *Progettazione*;
- *Implementazione*;
- *Testing*;
- *Messa in Esercizio* e *Manutenzione*.

Lo studio di fattibilità, in questo caso, è stato immediatamente superato, in quanto non c'erano da fare considerazioni riguardo le possibili soluzioni da adottare, le risorse finanziarie ed umane a disposizione, tenendo conto che sono stati fissati a priori gli strumenti da utilizzare.

Nella fase di analisi e specifica dei requisiti e di progettazione sono stati realizzati i principali diagrammi *UML(Unified Modelling Language)* previsti per lo sviluppo di un'applicazione.

L'analisi e la specifica dei requisiti ha previsto la realizzazione dei seguenti diagrammi:

- *Use Case Diagrams;*
- *Modello Concettuale.*

La fase di progettazione è stata divisa in due parti:

- *Progettazione della Base Dati*;
- *Progettazione dell'Applicazione.*

Nella fase di progettazione della base dati, è stato sviluppato il seguente diagramma:

- *Modello Logico*;

mentre nella fase di progettazione dell'applicazione Web, sono stati realizzati i seguenti diagrammi:

- *Class Diagram*;
- *Activity Diagrams*;
- *Sequence Diagrams.*

Lo sviluppo ha poi previsto la fase di implemetazione, realizzando l'applicazione con il *framework Struts*.

La fase di testing ha permesso la correzione di errori e problemi rilevati durante il funzionamento del software.

Infine, non sono state previste delle vere e proprie fasi di messa in esercizio e manutenzione.

Lo strumento software di tipo *CASE (Computer Aided Software Engineering)*, che è stato utilizzato per la realizzazione di tutti i diagrammi è il noto *Sybase PowerDesigner*.

3.2 Requisiti

Si vuole realizzare una applicazione Web per il commercio elettronico, dal nome NetShop, che permetta un'organizzazione ottimale di un centro commerciale virtuale.

Gli utilizzatori del centro commerciale possono assumere i seguenti ruoli:

- *Utente*: possiede un carrello virtuale dove pone gli aricoli da acquistare, ma non può effettuarne l'ordine;
- *Cliente*: può effettuare l'ordine degli articoli presenti nel carrello;
- *Gestore*: gestisce il proprio negozio virtuale;
- *Amministratore*: si occupa dell'amministrazione del centro commerciale.

Di seguito vengono descritti i requisti funzionali e i requisiti sui dati necessari per lo sviluppo dell'applicazione.

3.2.1 Requisiti funzionali

Ciascuno utilizzatore di NetShop possiede determinate funzionalità in base al proprio ruolo.

Funzionalità dell'Utente

Per l'*Utente* vi è la possibilità di:

- Visualizzare le categorie degli articoli, che verranno visualizzate in ogni area di NetShop;
- Visualizzare i negozi in rotazione, che verranno visualizzati nell'area iniziale di NetShop;
- Visualizzare i negozi con articoli in una categoria;
- Visualizzare gli articoli in una categoria;
- Visualizzare i dati e i reparti di un negozio, che verranno visualizzati in ogni area del negozio;
- Visualizzare gli articoli in vetrina per un negozio, che verranno visualizzati nell'area iniziale del negozio;
- Visualizzare gli articoli presenti in un reparto di un negozio;
- Visualizzare tutti gli articoli di un negozio;
- Visualizzare gli articoli di un negozio in una categoria;
- Visualizzare il dettaglio di un articolo;
- Visualizzare le informazioni di un negozio, che visualizzano i dati del negozio, delle sue modalità di consegna e del suo gestore;
- Cercare i negozi per nome o con ricerca avanzata, per arrivare facilmente al negozio desiderato;
- Cercare gli articoli per nome e categoria o con ricerca avanzata, per poter trovare subito gli articoli desiderati all'interno del centro commerciale;

- Cercare gli articoli in un negozio per nome o con ricerca avanzata, per poter trovare subito gli articoli all'interno del negozio che si sta visitando;
- Gestire un carrello della spesa, per poter mantenere temporaneamente gli articoli che si vorrebbero acquistare;
- Registrarsi, per poter diventare cliente o gestore e quindi godere di ulteriori funzionalità; nella registrazione come gestore viene creato il negozio ma non sarà visibile agli altri utenti fino a quando non verrà stipulato un contratto con NetShop; dopo la registrazione il cliente o il gestore e l'amministratore ricevono un'email della avvenuta registrazione;
- Effettuare il Login, per poter riaccedere a NetShop come cliente, gestore o amministratore;
- Recuperare la password per il login, nel caso in cui venga dimenticata, rispondendo ad una domanda segreta inserita durante la registrazione.

Funzionalità del Cliente

Per il *Cliente* vi è la possibilità di:

- Effettuare un ordine, quindi acquistare gli articoli posti nel carrello per un negozio; dopo l'effettuazione dell'ordine viene inviata un'email sia al cliente che al gestore del negozio;
- Gestire i suoi dati, per visualizzare i suoi dati ed eventualmente modificarli;
- Gestire gli ordini effettuati, ovvero visualizzare tutti gli ordini effettuati, il loro stato ed eliminare gli ordini non ancora in preparazione;
- Effetturare il Logout, per uscire dalla ruolo attuale e ritornare ad essere solo un utente;

inoltre il *Cliente* possiede tutte le funzionalità previste per l'*Utente*.

Funzionalità dell Gestore

Per il *Gestore* vi è la possibilità di:

- Gestire i dati del Negozio, per visualizzare e modificare i dati del suo negozio;
- Gestire il catalogo degli articoli, ovvero la possibilità di poter inserire, modificare ed eliminare gli articoli del negozio e poterli organizzare in reparti;
- Gestire gli ordini ricevuti, per visualizzare gli ordini ricevuti, il loro dettaglio e modificarne lo stato; quando l'ordine viene spedito viene invia un'email al cliente che lo avvisa dell'avvenuta spedizione dell'ordine;
- Gestire le modalità di consegna, classificate in modalità di spedizione e di pagamento, con la possibilità di visualizzarle, modificarle, aggiungerne delle nuove o eliminarne;

inoltre il *Gestore* possiede tutte le funzionalità previste per il *Cliente*.

Funzionalità dell'Amministratore

Per l'*Amministratore* vi è la possibilità di:

- Amministrare i clienti, per visualizzare tutti i clienti, concederli o meno l'accesso al sistema ed eventualmente eliminarli del tutto;
- Amministrare i gestori, per visualizzare tutti i gestori, concederli o meno l'accesso al sistema ed eventualmente eliminarli con il loro negozio;
- Amministrare i negozi, per visualizzare tutti i negozi, nasconderli o renderli visibili;

inoltre l'*Amministratore* possiede tutte le funzionalità previste per il *Cliente*.

3.2.2 Requisiti sui dati

Il *Cliente*, il *Gestore* e l'*Amministratore* possiedono le stesse proprietà, che sono:

- l'username e la password, per l'accesso al sistema;
- il nome, il cognome e il codice fiscale, che rappresentano i dati anagrafici;
- l'indirizzo, la città, la provincia e il cap, necessari per la spedizione degli articoli;
- l'email e il numero di telefono, per un eventuale contatto.

Il *Gestore* può gestire un solo negozio ed un negozio può essere gestito da un solo gestore. I negozi sono caratterizzati dalle seguenti proprietà:

- il nome del negozio e la sua descrizione in generale;
- il logo, la partiva iva del negozio e l'iva sugli articoli;
- il fax, che rappresenta il numero di fax per contattare un eventuale negozio reale.

Ogni negozio contiene più reparti. I reparti contengono più articoli, i quali vengono classificati in base alla loro categoria merceologica e possono essere posti in vetrina.

Gli articoli sono caratterizzati dalle seguenti proprietà:

- il nome dell'articolo e la sua descrizione;
- l'immagine dell'articolo;
- il prezzo dell'articolo.

Gli articoli possono essere acquistati attraverso un ordine.

Gli ordini vengono effettuati dai clienti, in base ad una modalità di spedizione e una modalità di pagamento e sono caratterizzati dalle seguenti proprietà:

- la data, che rappresenta quando è stato effettuato l'ordine;
- lo stato, che rappresenta lo stato in cui si trova l'ordine, che può essere: aperto, in preparazione, spedito o chiuso.

Ogni negozio propone, per quanto riguarda gli ordini, le sue modalità di consegna distinte in modalità di spedizione e pagamento che sono caratterizzate dalle seguenti proprietà:

- la descrizione, che descrive la modalità con cui avviene la spedizione o il pagamento;
- il costo, che rappresenta le spese necessarie per effettuare la modalità.

3.3 Analisi e Specifica dei Requisiti

3.3.1 Use Case Diagrams

I diagrammi dei casi d'uso (Use Case Diagrams) costituiscono uno strumento utile per catturare il comportamento esterno del sistema da sviluppare, senza dover specificare come tale comportamento debba essere realizzato; il sistema è visto come una scatola nera (black-box). Essi forniscono una descrizione dei modi in cui il sistema potrà essere utilizzato, ossia ciò che l'utente può fare su di esso e come quest'ultimo risponde alle sollecitazioni. La realizzazione dei diagrammi ha previsto un approccio top-down, partendo dallo scenario di carattere generale, in cui le modalità d'uso logicamente correlate sono accorpate, fino ad una decomposizione in cui sono descritti i casi di utilizzo non ulteriormente scomponibili.

3.3.1.1 Generale

Facendo riferimento ai requisiti assegnati, è stato definito un Use Case Diagram Generale, che si pone al livello di astrazione più elevato del sistema, mettendo in evidenza tutte quelle che sono le modalità di utilizzo dello stesso da parte dell'utente, del cliente, del gestore e dell'amministratore. Questi ultimi rappresentano gli attori che possono accedere alle funzionalità offerte dal sistema, di cui alcune sono condivise, ossia accessibili da entrambi, mentre altre sono prerogativa esclusiva di uno dei ruoli.

A partire da questo diagramma, è possibile effettuare una decomposizione scendendo ad un livello di astrazione minore, in cui si prendono in considerazione nel dettaglio le modalità di utilizzo del sistema.

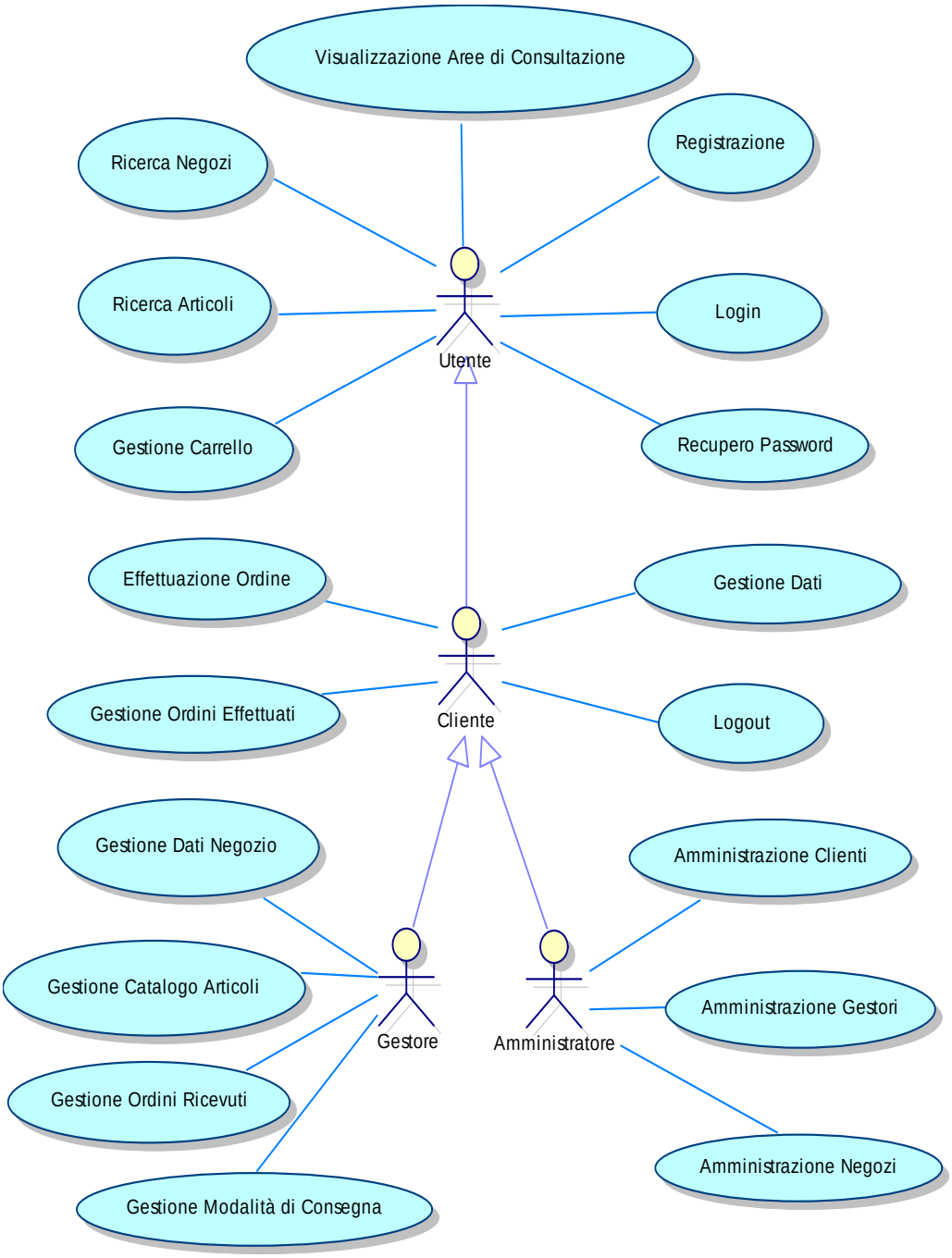

Figura 3.1: Use Case Diagram Generale

3.3.1.2 Visualizzazione Aree di Consultazione

Comprende tutti i casi d'uso relativi alla visualizzazione delle informazioni contenute nelle aree di consultazione. Le aree di consultazione possono essere classificate in aree di consultazione di NetShop e aree di consultazione del negozio.

Per quanto riguarda le aree di consultazione di NetShop esse possono visualizzare le categorie degli articoli, i negozi in rotazione, i negozi che contengono gli articoli in una determinata categoria e gli articoli in una determinata categoria.

Mentre le aree di consultazione di un negozio possono visualizzare i dati del negozio e i suoi reparti, gli articoli in vetrina, gli articoli in un determinato reparto, gli articoli del negozio, gli articoli del negozio in una determinata categoria e le informazioni del negozio.

Infine sia in NetShop che nel negozio esiste un area di consultazione che consente di visualizzare il dettaglio di un articolo.

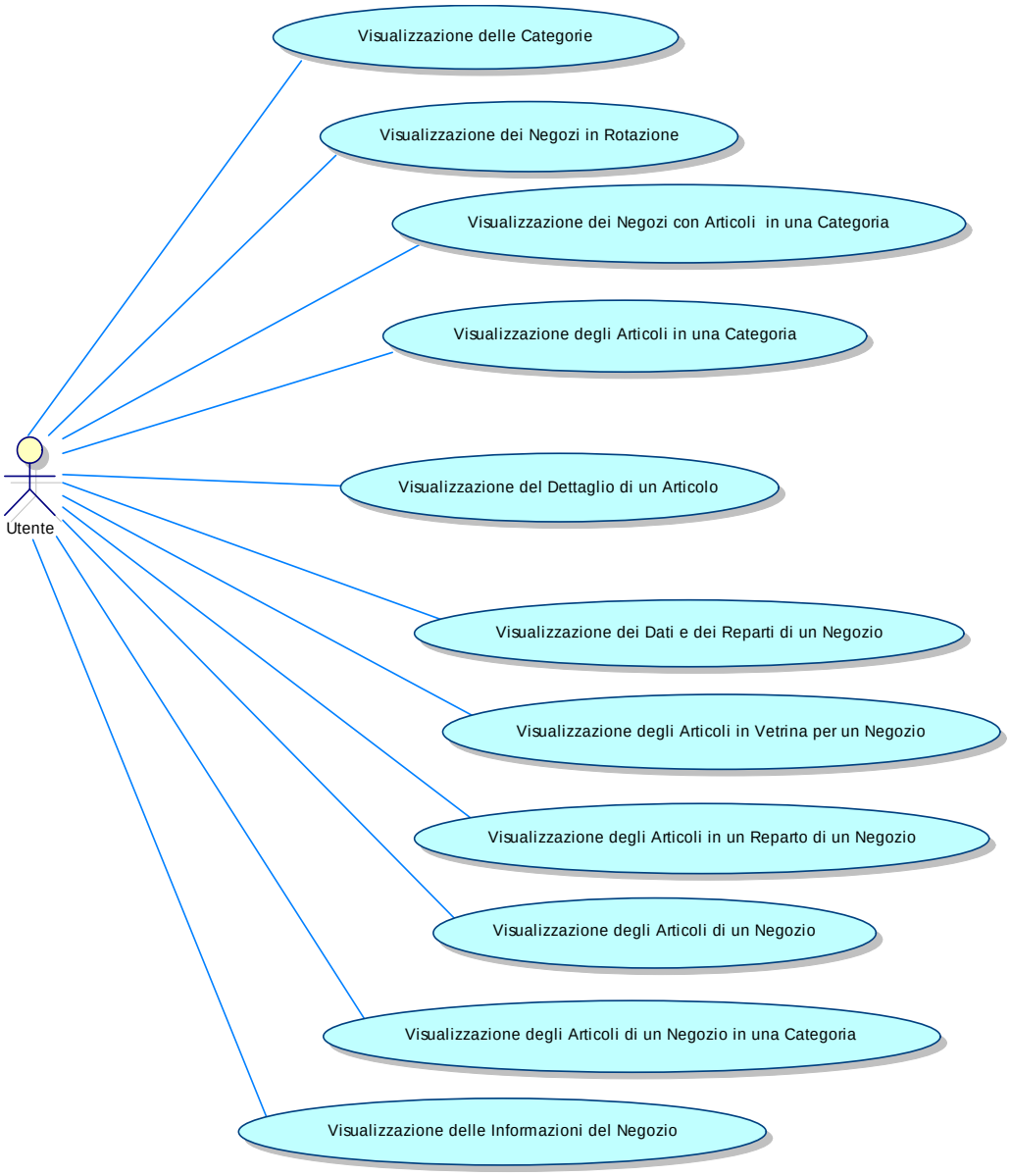

Figura 3.2: Use Case Diagram Visualizzazione Aree di Consultazione

3.3.1.3 Ricerca Negozi

Comprende i casi d'uso per la ricerca di negozi. La ricerca dei negozi può essere effettuata per nome o con ricerca avanzata, fornendo per quest'ultima, oltre al nome, anche la descrizione. Dopo che si è conclusa la ricerca, verranno visualizzati i i negozi trovati.

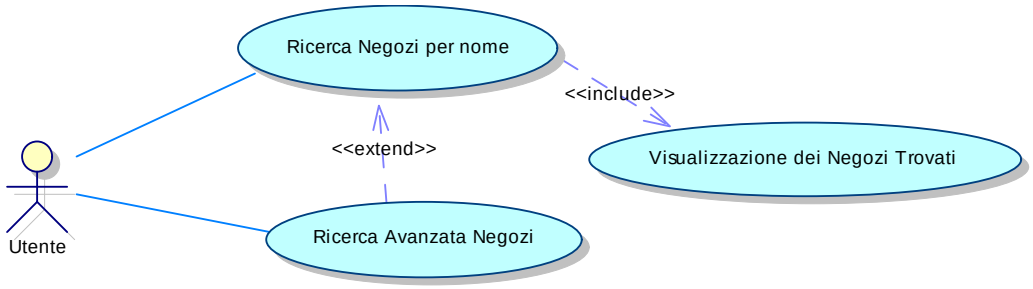

Figura 3.3: Use Case Diagram Ricerca Negozi

3.3.1.4 Ricerca Articoli

Comprende i casi d'uso per la ricerca degli articoli in NetShop o all'interno di un negozio. La ricerca degli articoli all'interno di NetShop può essere effettuata per nome e categoria degli articoli, mentre in un negozio solo per nome degli articoli. In entrambi i casi è possibile effettuare una ricerca avanzata, fornendo parametri più specifici. Dopo che si è conclusa la ricerca, verranno visualizzati gli articoli trovati.

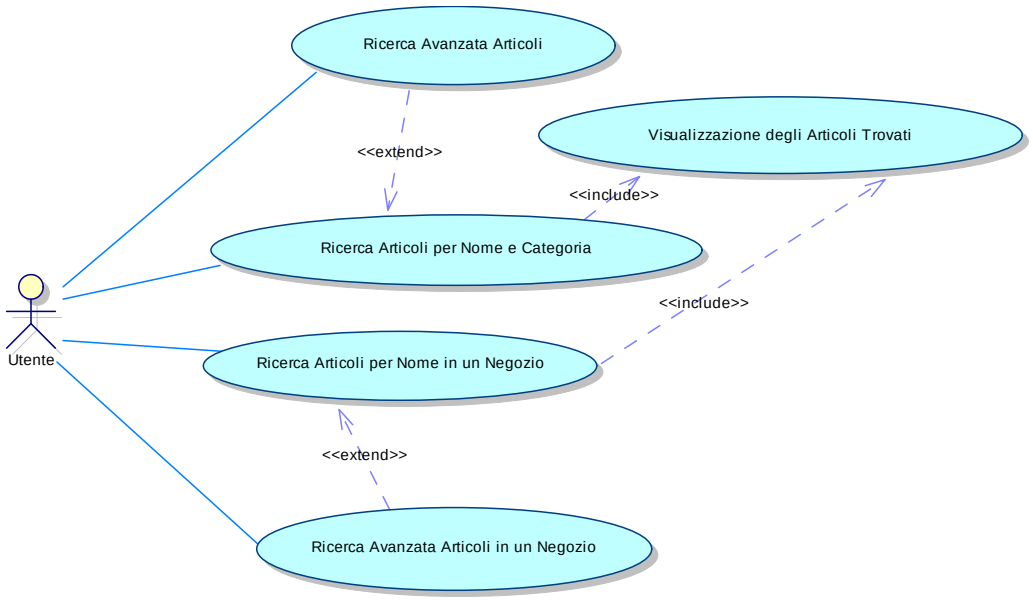

Figura 3.4: Use Case Diagram Ricerca Articoli

3.3.1.5 Gestione Carrello

Comprende i casi d'uso per la gestione di un carrello virtuale. L'*Utente* può inserire un articolo nel carrello, visualizzare tutti gli articoli presenti, aggiornare il carrello, ovvero aggiornare le quantità per alcuni articoli ed eventualmente eliminarli ed infine può svuotare il carrello.

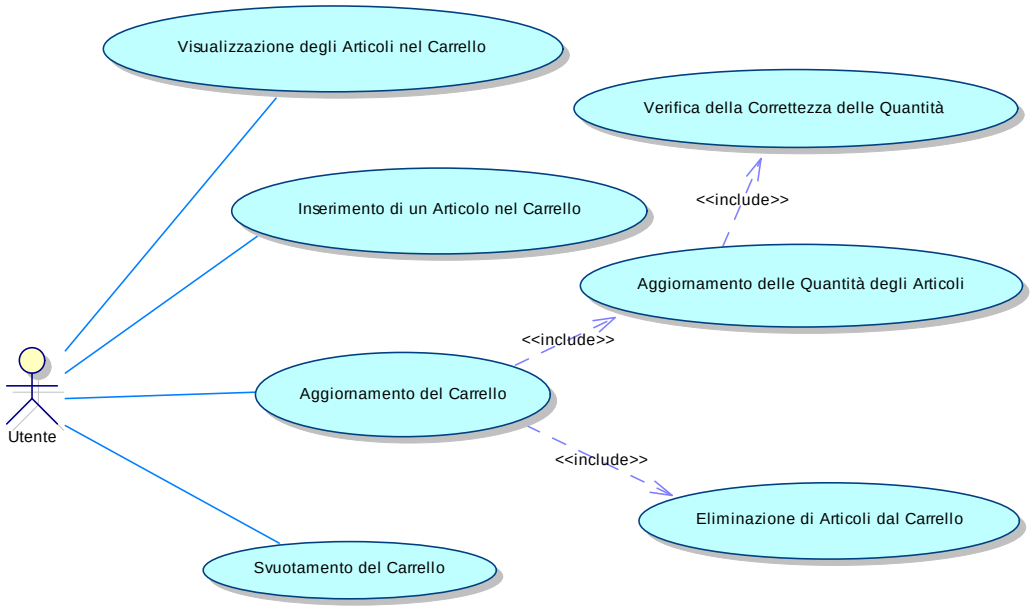

Figura 3.5: Use Case Diagram Gestione Carrello

3.3.1.6 Registrazione

Comprende i casi d'uso necessari per poter effettuare la registrazione a NetShop come cliente o come gestore. La registrazione come cliente inserisce il nuovo cliente, effettua il login in automatico ed invia un email al cliente e all'amministratore per l'avvenuta registrazione. La registrazione come gestore inserisce il nuovo gestore, il suo negozio con una modalità di spedizione, di pagamento e un reparto iniziale, effettua il login in automatico ed invia un email al gestore e all'amministratore per l'avvenuta registrazione.

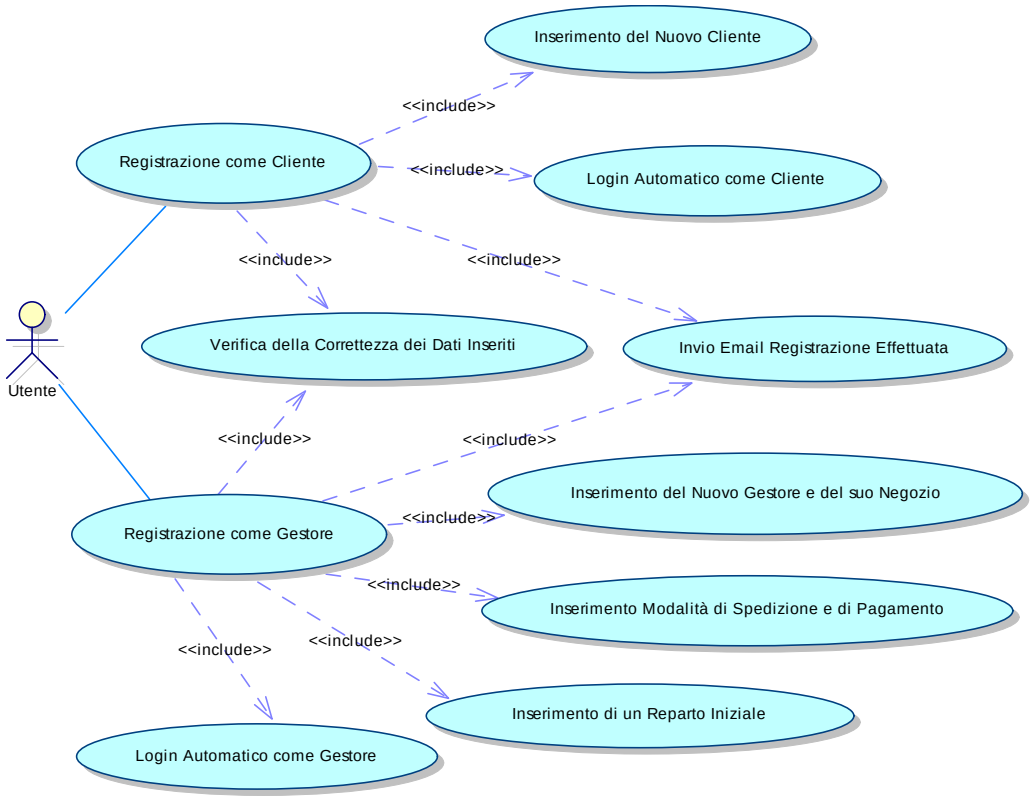

Figura 3.6: Use Case Diagram Registrazione

3.3.1.7 Login e Recupero Password

Questi due casi d'uso risultano essere atomici e non ulteriormente scomponibili. Il login consente all'*Utente* di poter accedere al sistema come cliente o come gestore. Il recupero password consente all'*Utente* di poter recuperare la password dimenticata, necessaria per effettuare il login.

3.3.1.8 Effettuazione Ordine e Logout

Questi due casi d'uso risultano essere atomici e non ulteriormente scomponibili. L'effettuazione dell'ordine consente al *Cliente* di effettuare un ordine presso un negozio. Il logout consente di poter chiudere la sessione di lavoro aperta in precedenza con la procedura di login.

3.3.1.9 Gestione Dati

Comprende i casi d'uso per la gestione dei dati. Il *Cliente* può visualizzare e modificare i suoi dati personali oppure può visualizzare o modificare i suoi dati di autenticazione.

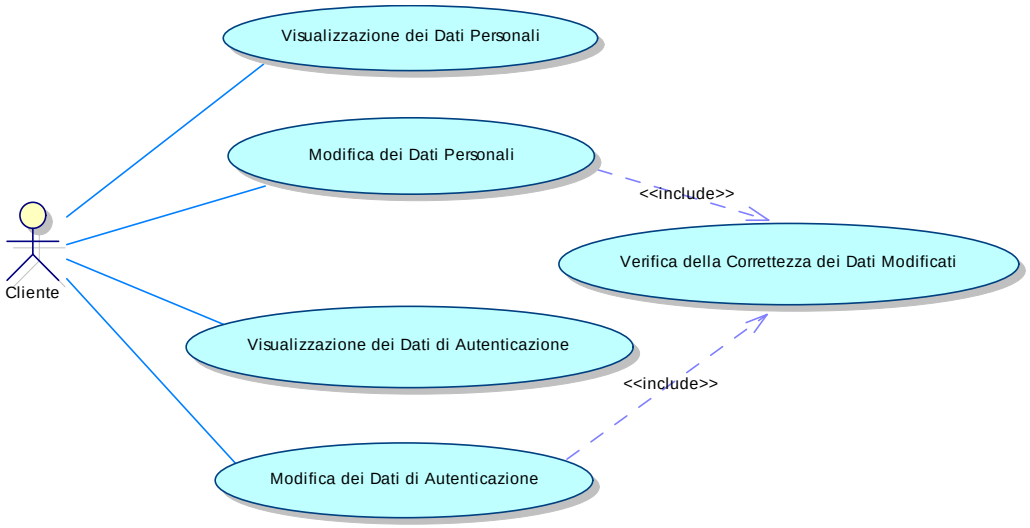

Figura 3.7: Use Case Diagram Gestione Dati

3.3.1.10 Gestione Ordini Effettuati

Comprende i casi d'uso per la gestione degli ordini effettuati. Il *Cliente* può visualizzare gli ordini effettuati, visualizzarne il dettaglio ed eliminarne uno non ancora in preparazione.

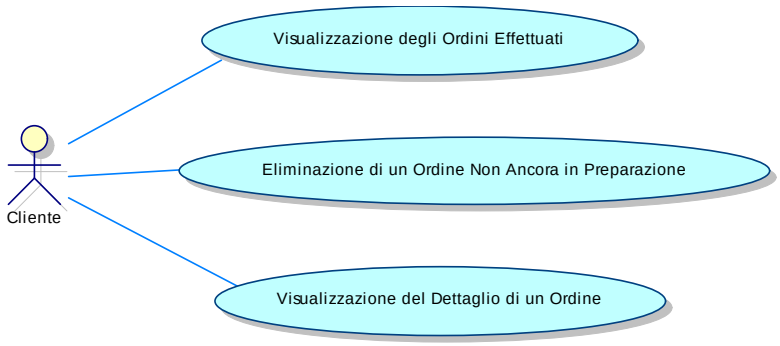

Figura 3.8: Use Case Diagram Gestione Ordini Effettuati

3.3.1.11 Gestione Dati Negozio

Comprende i casi d'uso per la gestione dei dati del negozio. Il *Gestore* può visualizzare e modificare i dati del suo negozio.

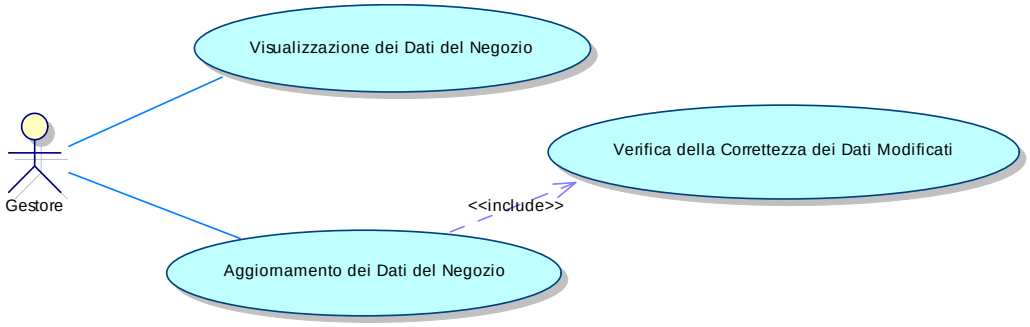

Figura 3.9: Use Case Diagram Gestione Dati Negozio

3.3.1.12 Gestione Catalogo Articoli

Comprende i casi d'uso per la gestione del proprio catalogo degli articoli. Il *Gestore* può effettuare la gestione dei reparti e degli articoli del suo negozio. Per quanto riguarda la gestione dei reparti è possibile visualizzarli tutti, inserirne un nuovo, aggiornarne il nome ed eliminarne uno che non contenga articoli. Mentre per la gestione degli articoli è possibile inserire un nuovo articolo, aggiornarne i dati ed eliminarne uno.

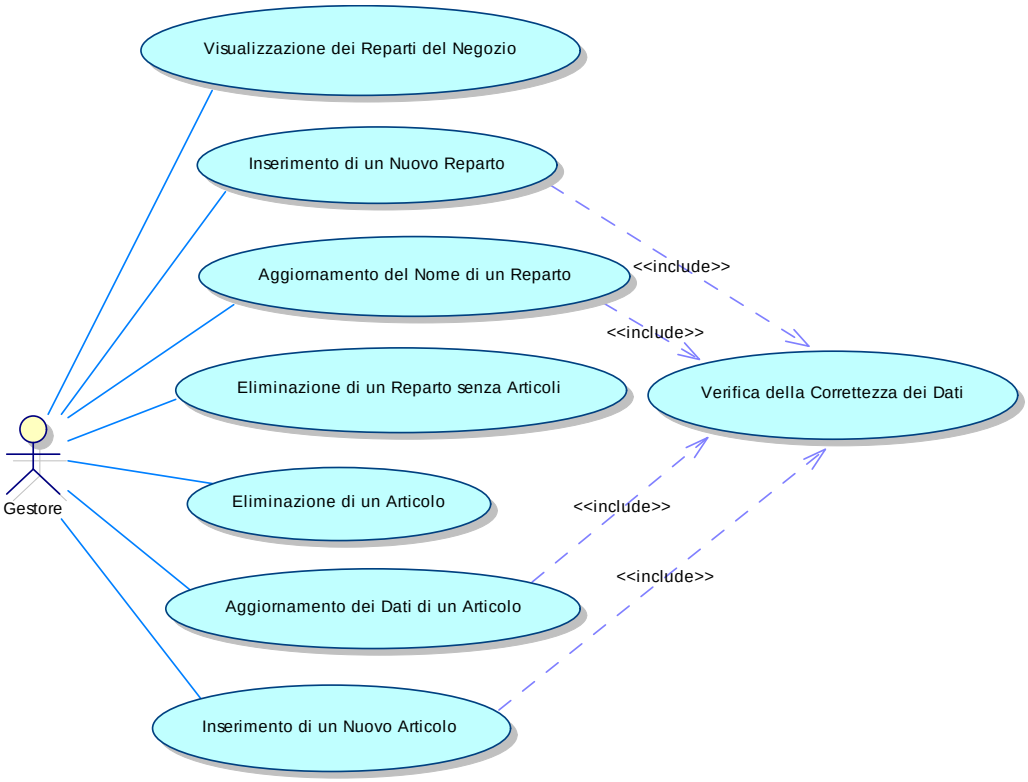

Figura 3.10: Use Case Diagram Gestione Catalogo Articoli

3.3.1.13 Gestione Ordini Ricevuti

Contiene i casi d'uso per la gestione degli ordini ricevuti. Il *Gestore* può visualizzare gli ordini ricevuti, visualizzare il dettaglio di un'ordine ricevuto, aggiornarne lo stato e visualizzarne le informazioni del cliente che lo ha effettuato.

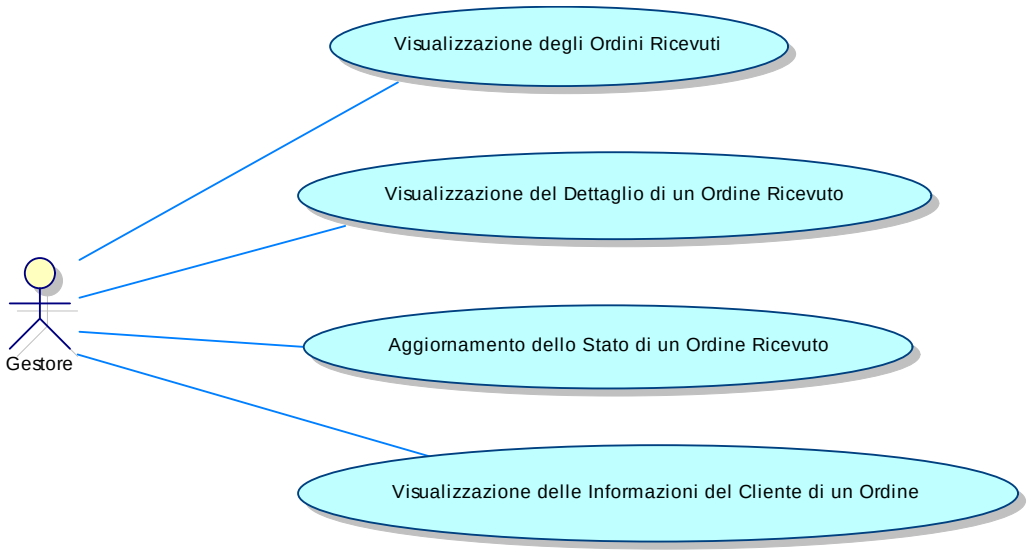

Figura 3.11: Use Case Diagram Gestione Ordini Ricevuti

3.3.1.14 Gestione Modalità di Consegna

Contiene i casi d'uso per la gestione delle modalità di consegna. Il *Gestore* può visualizzare le modalità di spedizione o di pagamento, inserirne, aggiornarne o eliminarne una.

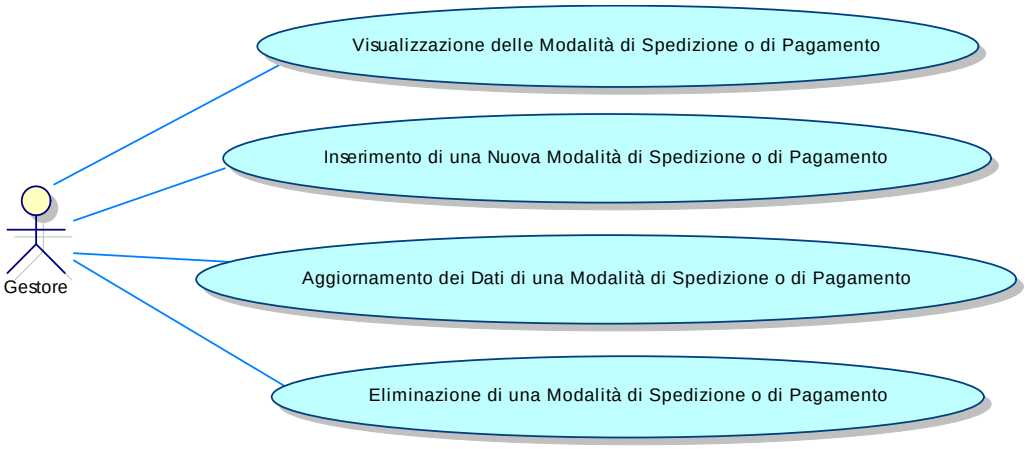

Figura 3.12: Use Case Diagram Gestione Modalità di Consegna

3.3.1.15 Amministrazione Clienti

Contiene i casi d'uso per l'amministrazione di tutti i clienti di NetShop. L'*Amministratore* può visualizzare tutti i clienti presenti, visualizzare i dati di un cliente, bloccarlo o sbloccarlo per negargli o concedergli la possibilità di accedere a NetShop o eliminarlo definitivamente insieme agli ordini che ha effettuato.

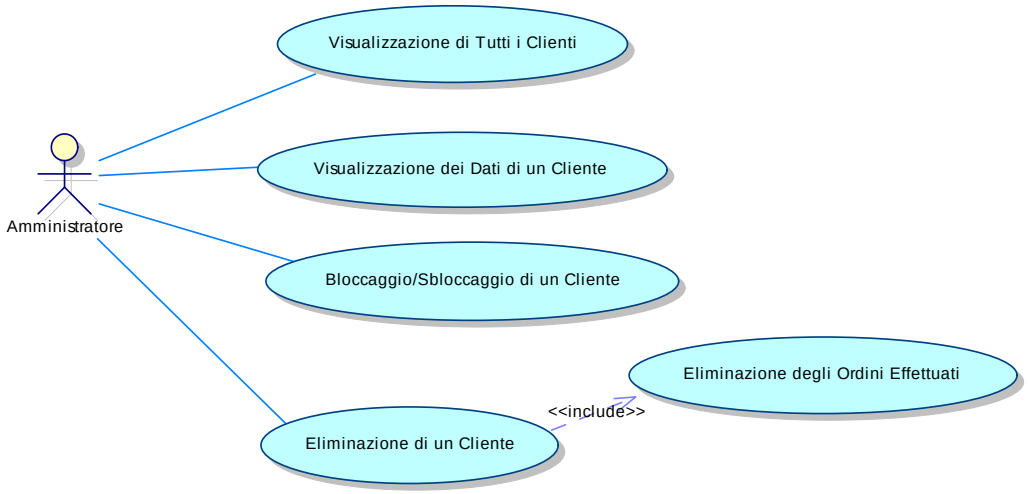

Figura 3.13: Use Case Diagram Amministrazione Clienti

3.3.1.16 Amministrazione Gestori

Contiene i casi d'uso per l'amministrazione di tutti i gestori di NetShop. L'*Amministratore* può visualizzare tutti i gestori presenti, visualizzare i dati di un gestore, bloccarlo o sbloccarlo per negargli o concedergli la possibilità di accedere a NetShop o eliminarlo definitivamente insieme al suo negozio e a tutti gli elementi ad esso associati.

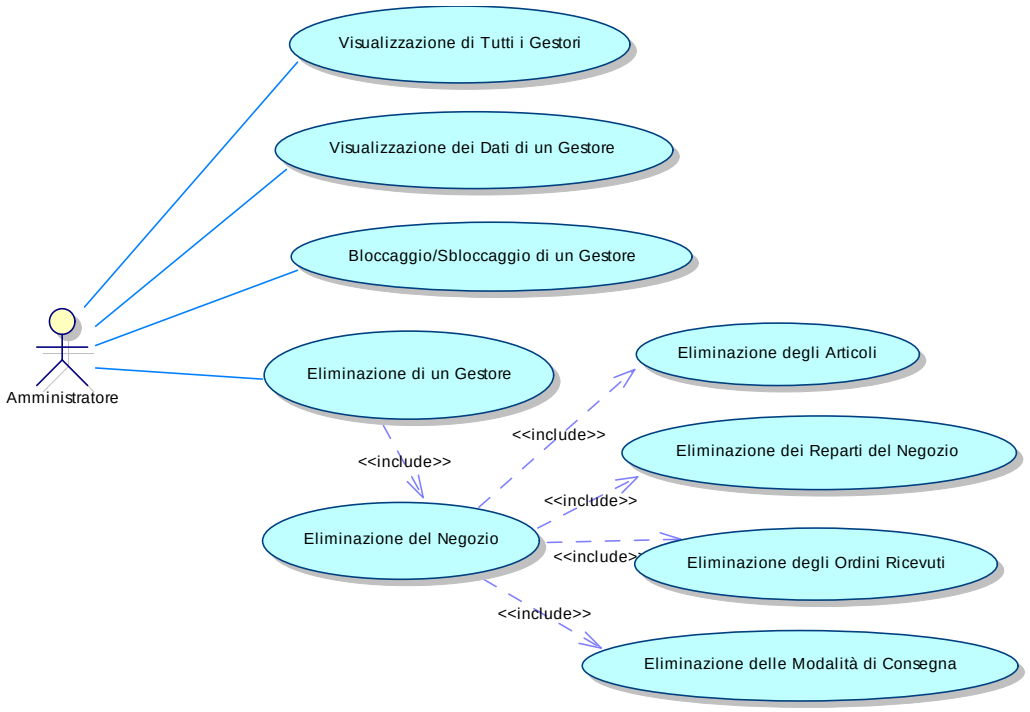

Figura 3.14: Use Case Diagram Amministrazione Gestori

3.3.1.17 Amministrazione Negozi

Contiene i casi d'uso per l'amministrazione di tutti i negozi presenti in NetShop. L'*Amministratore* può visualizzare tutti i negozi presenti, visualizzare i dati di un negozio, nasconderlo o renderlo visibile all'interno di NetShop.

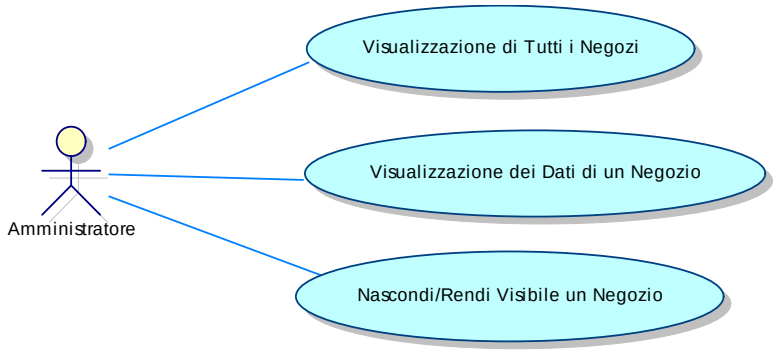

Figura 3.15: Use Case Diagram Amministrazione Negozi

3.3.2 Modello Concettuale

Il modello concettuale permette di descrivere tutte quelle che sono le entità del modello reale e le relazioni tra di esse. Per ciascuna entità o relazione è possibile definirne gli attributi, ponendo particolare attenzione ai cosiddetti attributi primari delle entità.

Tale modello è stato realizzato a partire dai requisiti che sono stati forniti. Sono state individuate le entità del modello e per esse i loro attributi, poi sono state ricavate le relazioni che legano le entità tra di loro cercando di evitare ridondanze sia negli attributi che nelle relazioni.

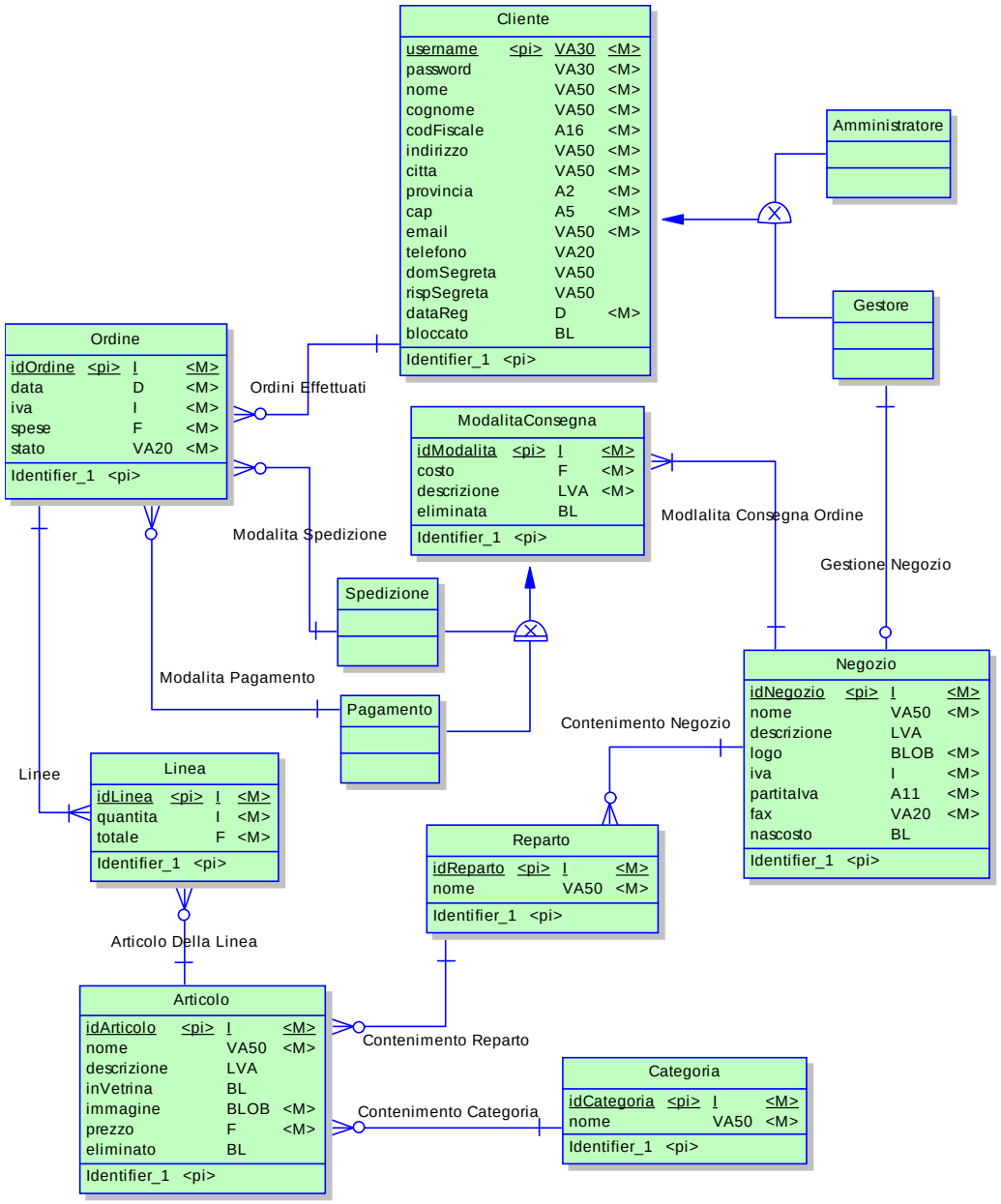

Figura 3.16: Modello Concettuale

3.3.2.1 Dizionario dei dati

Il dizionario dei dati descrive le entità, le relazioni ed i loro attributi.

Di seguito vengono descritte in breve le entità, alcuni dei loro attributi degni di nota e le relazioni esistenti tra le entità.

Cliente

Contiene le informazioni del cliente. Gli attributi da evidenziare sono:

- *username*: rappresenta la primary key;
- *password*: rappresenta la parola segreta per l'accesso al sistema;
- *domSegreta* e *rispSegreta*: rappresentano la domanda segreta e la sua risposta, che vengono utilizzate per il recupero della password;
- *dataReg*: rappresenta la data di registrazione;
- *bloccato*: rappresenta lo stato del cliente, che lo rende o non lo rende accessibile al sistema.

Gestore e Amministratore

Rappresentano due specializzazioni dell'entità *Cliente* in una gerarchia di tipo parziale ed esclusiva. Esse non contengono attributi propri ma rappresentano solo un particolare tipo di cliente.

Negozio

Contiene le informazioni del negozio. Gli attributi da evidenziare sono:

- *idNegozio*: rappresenta la primary key;
- *nascosto*: rappresenta lo stato del negozio, che lo rende o non lo rende visibile da parte degli utenti.

Articolo

Contiene le informazioni dell'articolo. Gli attributi da evidenziare sono:

- *idArticolo*: rappresenta la primary key;
- *inVetrina*: ci dice se l'articolo è posto in vetrina;

- *eliminato*: ci dice se l'articolo è stato eliminato; tale attributo è necessario, in quanto in caso di eliminazione, se ci sono altre entità legate all'articolo, esso non può essere eliminato realmente dalla base dati.

Categoria

Rappresenta le informazioni della categoria di un articolo. L'unico attributo da evidenziare è *idCategoria*, che rappresenta la primary key.

Reparto

Rappresenta le informazioni del reparto di un articolo. L'unico attributo da evidenziare è *idReparto*, che rappresenta la primary key.

Ordine

Contiene le informazioni di un ordine. Gli attributi da evidenziare sono:

- *idOrdine*: rappresenta la primary key;
- *iva*: rappresenta il valore dell'iva sugli articoli che sono contenuti nell'ordine; questo valore viene mantenuto perchè il negozio potrebbe cambiare in futuro tale valore;
- *spese*: rappresenta il costo delle spese di spedizione e di pagamento; questo valore viene mantenuto perchè i costi delle modalità di spedizione e di pagamento potrebbero in un futuro cambiare.

Linea

Rappresenta una linea d'ordine. Vengono descritti tutti i suoi attributi, che sono:

- *idLinea*: rappresenta la primary key;
- *quantita*: rappresenta la quantità dell'articolo a cui fa riferimento la linea;

- *totale*: rappresenta il totale della linea ovvero il prodotto tra il prezzo dell'articolo e la quantità; questo valore viene mantenuto perchè il prezzo dell'articolo potrebbe in futuro cambiare.

ModalitaConsegna

Rappresenta una modalità con la quale viene consegnato un ordine. Gli attributi da evidenziare sono:

- *idModalita*: rappresenta la primary key;
- *eliminata*: ci dice se la modalità di consegna è stata eliminata; tale attributo è necessario, in quanto in caso di eliminazione, se ci sono altre entità legate alla modalità di consegna, essa non può essere eliminata realmente dalla base dati.

Spedizione e Consegna

Rappresentano due specializzazione di *ModalitaConsegna* in una gerarchia di tipo totale ed esclusiva. Esse non contengono attributi propri ma rappresentano solo una particolare modalità di consegna.

Relazioni tra le entità

Le relazioni che legano le entità del modello concettuale che è stato definito sono le seguenti:

- *Gestione Negozio*: lega *Gestore* e *Negozio* in una relazione uno-a-uno; un gestore può gestire un solo negozio e un negozio può essere gestito da un solo gestore;
- *Contenimento Categoria*: lega *Categoria* e *Articolo* in una relazione uno-a-molti; ad una categoria possono appartenere più articoli mentre un articolo può appartenere ad una solo categoria;

- *Contenimento Reparto*: lega *Reparto* e *Articolo* in una relazione uno-a-molti; un reparto può contenere più articoli mentre un articolo può appartenere ad un solo reparto;
- *Contenimento Negozio*: lega *Negozio* e *Reparto* in una relazione uno-a-molti; un negozio può contenere più reparti mentre un reparto può appartenere ad un solo negozio;
- *Modalità Consegna Ordine*: lega *Negozio* e *ModalitaConsegna* in una relazione uno-a-molti; un negozio può contenere da due a più modalità di consegna mentre una modalità di consegna può appartenere ad un solo negozio;
- *Ordini Effettuati*: lega *Cliente* e *Ordine* in una relazione uno-a-molti; un cliente può effettuare più ordini mentre un ordine può appartenere ad un solo cliente;
- *Linee*: lega *Ordine* e *Linea* in una relazione uno-a-molti; un ordine può contenere da una o più linee mentre una linea può appartenere ad un solo ordine;
- *Articolo della Linea*: lega *Articolo* e *Linea* in una relazione uno-a-molti; un articolo può essere contenuto in più linee mentre una linea contiene un solo articolo;
- *Modalità di Spedizione*: lega *Spedizione* a *Ordine* in una relazione uno-a-molti; un ordine contiene un unica modalità di spedizione mentre un modalità di spedizione può essere utilizzata da più ordini;
- *Modalità di Pagamento*: lega *Pagamento* a *Ordine* in una relazione uno-a-molti; un ordine contiene un unica modalità di pagamento mentre un modalità di pagamento può essere utilizzata da più ordini.

3.4 Progettazione della Base Dati

La fase di progettazione della base dati ha previsto la realizzazione del Modello Logico, attraverso un'opera di ristrutturazione ed una successiva opera di traduzione del Modello Concettuale. Dal Modello Logico realizzato sarà poi possibile implementare la base dati.

3.4.1 Dal Modello Concettuale al Modello Logico

3.4.1.1 Ristrutturazione

Prima di eseguire l'opera di traduzione dal Modello Concettuale al Modello Logico, bisogna effettuare la ristrutturazione; essa consta delle seguenti operazioni:

- Analisi di minimalità;
- Analisi di leggibilità;
- Analisi di normalità.

Avendo progettato il Modello Concettuale con PowerDesigner, si ottiene che, senza ulteriori operazioni, le tre analisi danno risultato positivo e quindi si può proseguire con la traduzione da un modello all'altro; nello specifico il Modello Concettuale non presenta ridondanze, o sono già state tradotte perché non è possibile averle nel software di CASE utilizzato, ed è già normalizzato.

3.4.1.2 Traduzione

L'opera di traduzione viene effettuata automaticamente dal software usato, però è sempre meglio controllare ciò che viene generato; in quanto potrebbero esserci incongruenze con ciò che ci si aspettava di avere, un pò per qualche errore che può capitare durante la progettazione del Modello Concettuale e un pò per errori di utilizzo del programma, che esegue solo i nostri comandi, corretti o sbagliati che siano.

3.4.2 Modello Logico

Dopo la fase di ristrutturazione e di traduzione, il modello logico risultante è costituito da una serie di tabelle, con le relative colonne, che prendono il posto delle entità e delle relazioni, con i relativi attributi, presenti nel modello concettuale. In particolare il modello logico risulta avere le seguenti caratteristiche:

- le entità figlie delle due relazioni di specializzazioni, relative alle entità padre *Cliente* e *ModalitaConsegna*, sono state accorpate nelle relative entità padre e per poter distinguere le occorrenza dell'una e dell'altra è stato introdotto l'attributo *tipo;*
- la relazione uno-a-uno *Gestione Negozio* è stata trasformata in entrambi i lati in un attributo *idNomeEntità* dell'entità del lato opposto;
- tutte le relazioni uno-a-molti sono state trasformate in un attributo *idNomeEntità* dell'entità dal lato uno e poste nell'entità dal lato molti;
- infine, tutte le relazione hanno vincoli (*costraints*) di tipo *restrict* sugli eventi *onDelete* e *onUpdate*; in tal modo la base dati è più sicura e la gestione degli eventi in caso di eliminazioni o di modifiche viene effettuata a livello applicativo.

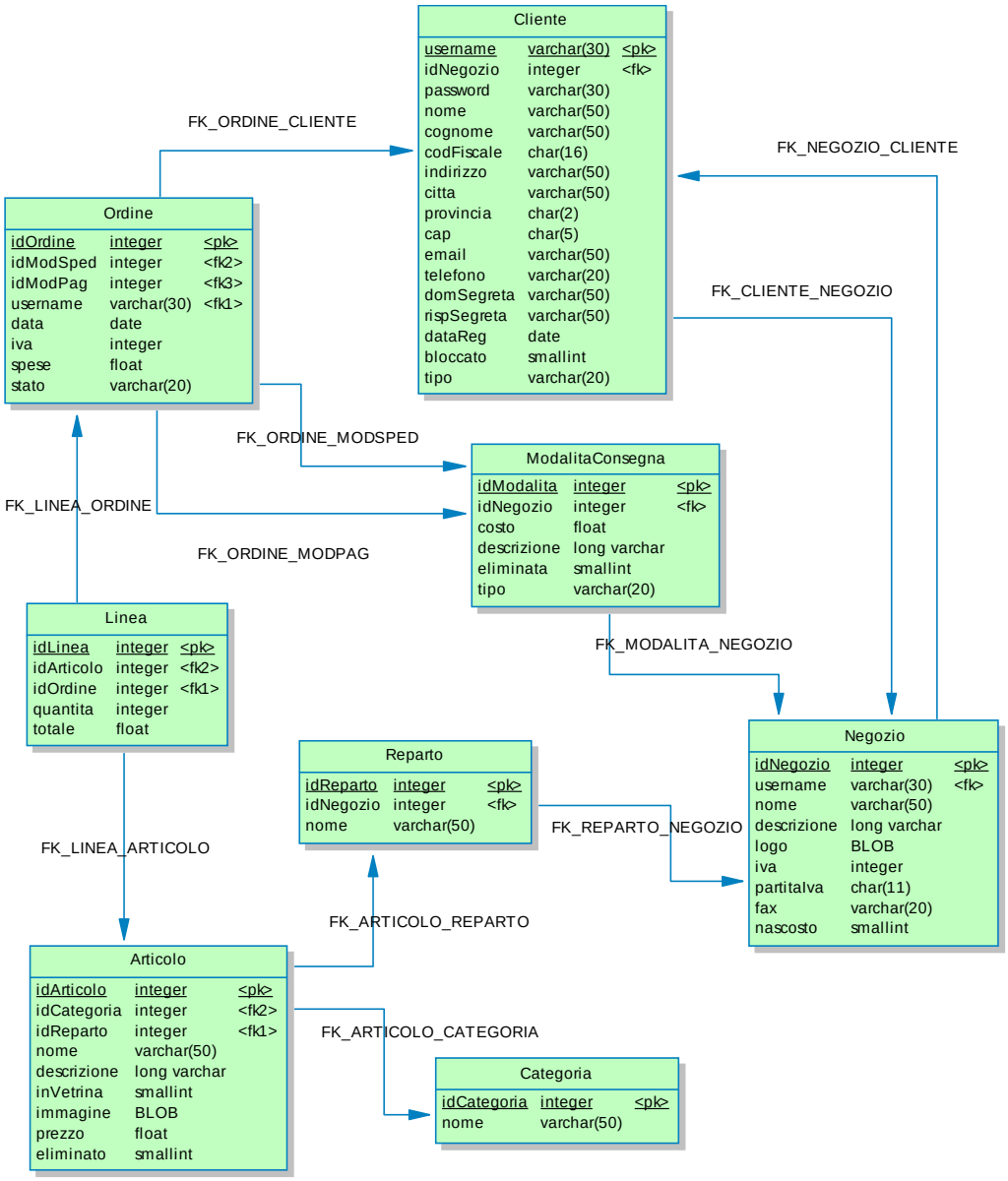

Figura 3.17: Modello Logico

3.5 Progettazione dell'Applicazione

La fase di progettazione dell'applicazione ha previsto la realizzazione di alcuni dei principali diagrammi UML: Class Diagram, Activity Diagrams e Sequence Diagrams. Essi costituiscono una astrazione del sistema e sono indipendenti da quelli che possono essere gli strumenti utilizzati nella fase di implementazione dell'applicazione.

3.5.1 Class Diagram

Mediante il diagramma delle classi (Class Diagram) è possibile definire tutte quelle che sono le entità caratteristiche del sistema software e le relazioni che ci sono tra di esse. Ciascuna entità è ovviamente modellabile attraverso il concetto di classe, che ne definisce le caratteristiche (i dati) ed il comportamento (le operazioni). Attraverso questo diagramma è possibile modellare la vista statica del sistema, tenendo anche conto di quelle che sono le funzionalità offerte da quest'ultimo e rese disponibili attraverso le entità che lo compongono.

Tale diagramma è stato costruito a partite dal Modello Concettuale e sulla base dei requisiti forniti per l'applicazione.

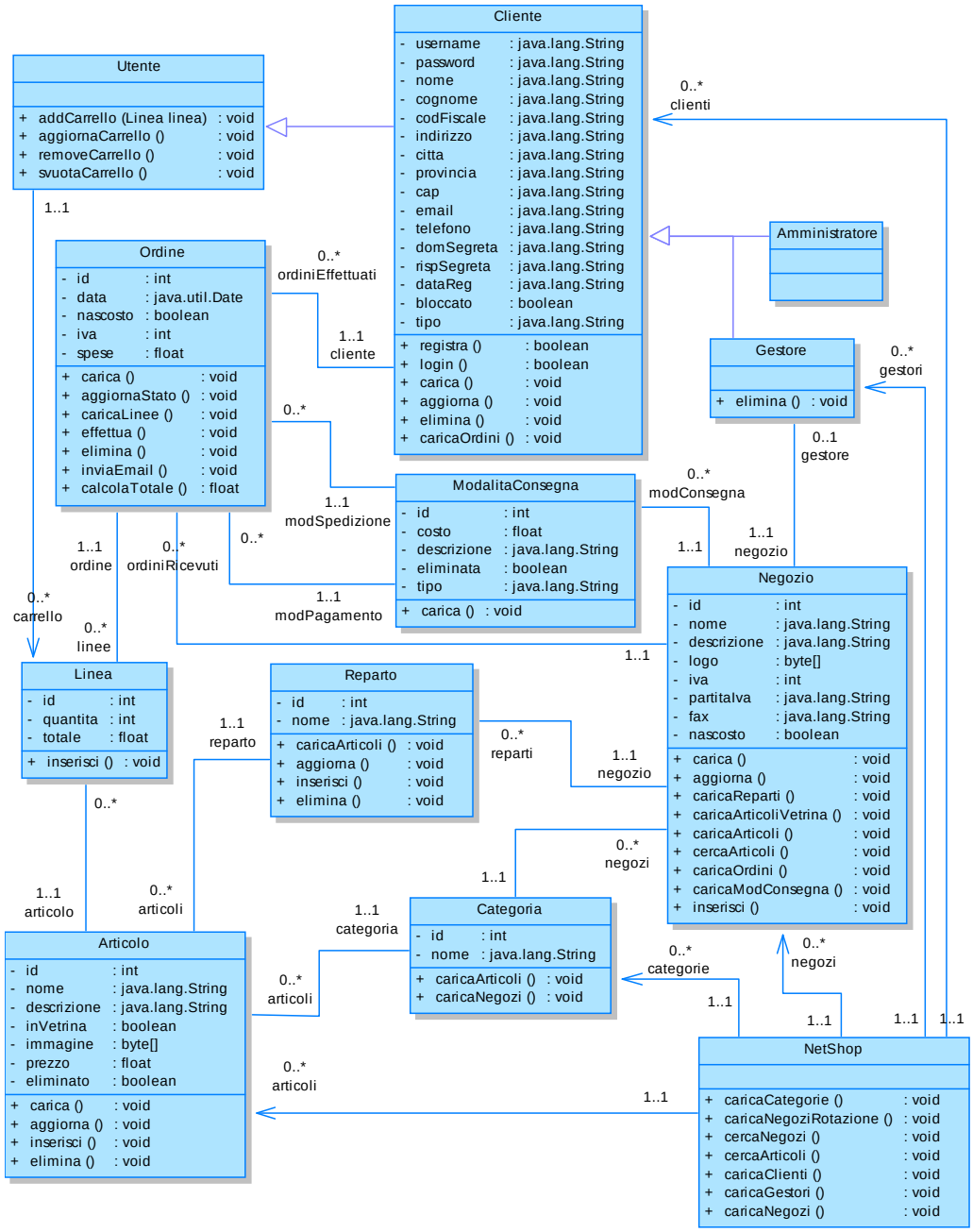

Figura 3.18: Class Diagram

3.5.2 Activity Diagrams - Sequence Diagrams

Facendo riferimento a quelle che sono le funzionalità offerte dal sistema e le modalità di utilizzo di quest'ultimo, attraverso gli Activity Diagrams è

possibile modellare le azioni necessarie per compiere un'attività ed il flusso di controllo tra di esse.

Invece, attraverso i Sequence Diagrams è possibile descrivere la vista dinamica del sistema, enfatizzando le interazioni dovute allo scambio di messaggi tra gli oggetti e il loro ordinamento temporale.

Nel seguito vengono presentati gli Activity Diagrams e dei Sequence Diagrams per alcuni casi d'uso.

3.5.2.1 Visualizzazione delle Categorie

Visualizza le categorie degli articoli. L'*Utente* non richiede direttamente la visualizzazione delle categorie, ma esse vengono visualizzate da parte del sistema ogni volta che viene richiesta la visualizzazione di un'area di NetShop. Il sistema crea un oggetto *NetShop*, legge dalla base dati le categorie, le carica nell'oggetto *NetShop* e ne visualizza il nome nell'interfaccia. E' da notare che l'oggetto *NetShop*, che viene creato con questa funzionalità, viene riutilizzato nell'esecuzione di altre funzionalità.

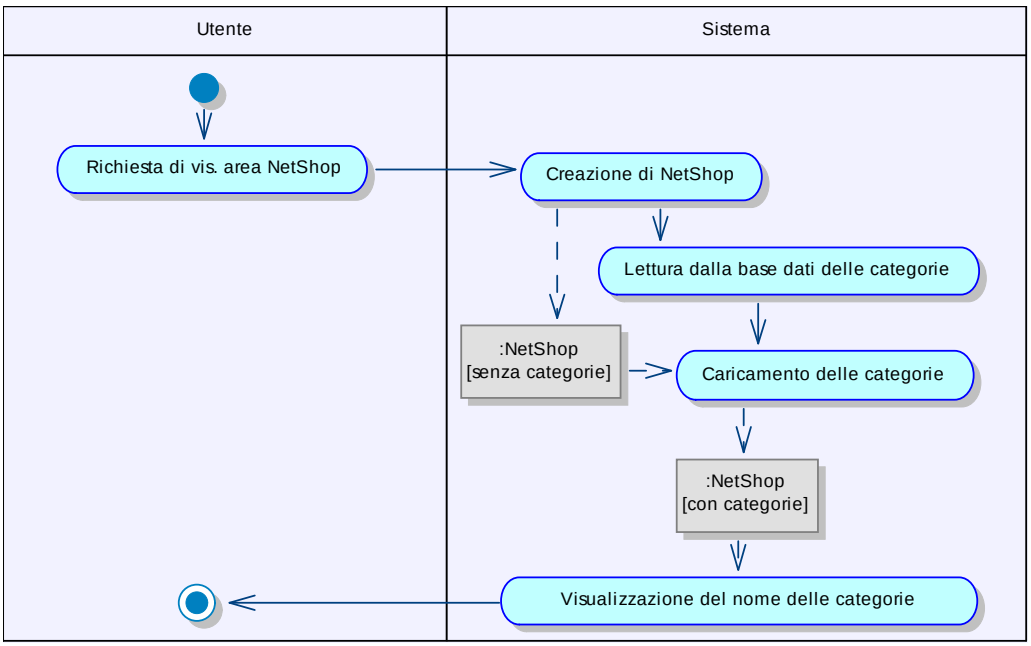

Figura 3.19: Activity Diagram Visualizzazione delle Categorie

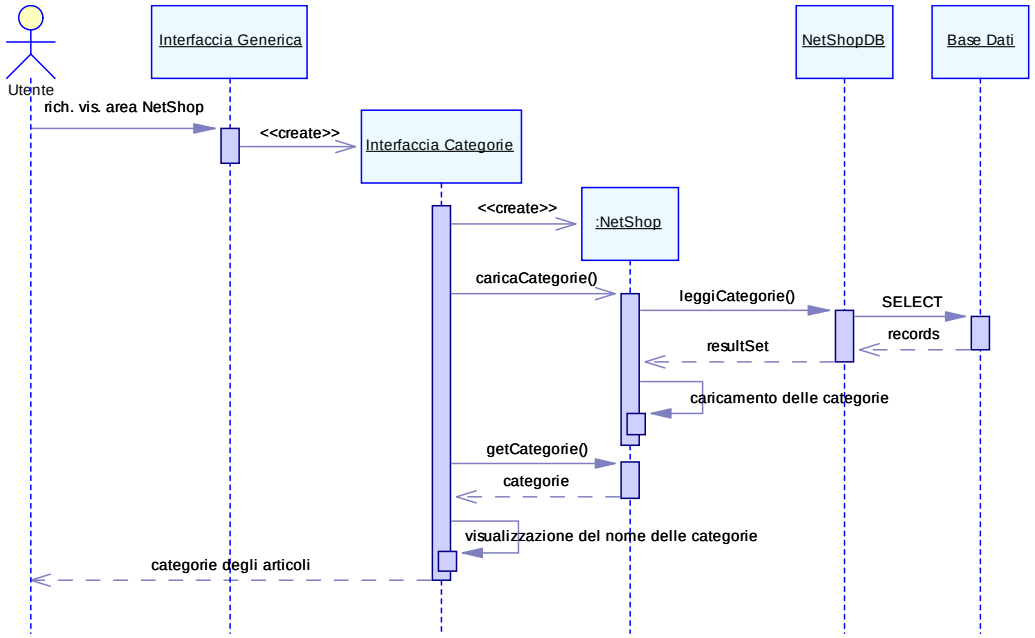

Figura 3.20: Sequence Diagram Visualizzazione delle Categorie

3.5.2.2 Visualizz. dei Dati e dei Reparti di un Negozio

Visualizza i dati e i reparti di un negozio. L'*Utente* non richiede direttamente la visualizzazione di tale area, ma essa viene visualizzata da parte del sistema ogni volta che viene richiesta la visualizzazione di un'area del negozio. Il sistema crea un oggetto *Negozio*, legge dalla base dati i dati del negozio e li carica nell'oggetto *Negozio*, poi legge dalla base dati i reparti del negozio e carica anch'essi nell'oggetto *Negozio*, infine visualizza i dati del negozio e dei reparti all'interno dell'interfaccia. E' da notare che l'oggetto *Negozio*, che viene creato con questa funzionalità, viene riutilizzato nell'esecuzione di altre funzionalità.

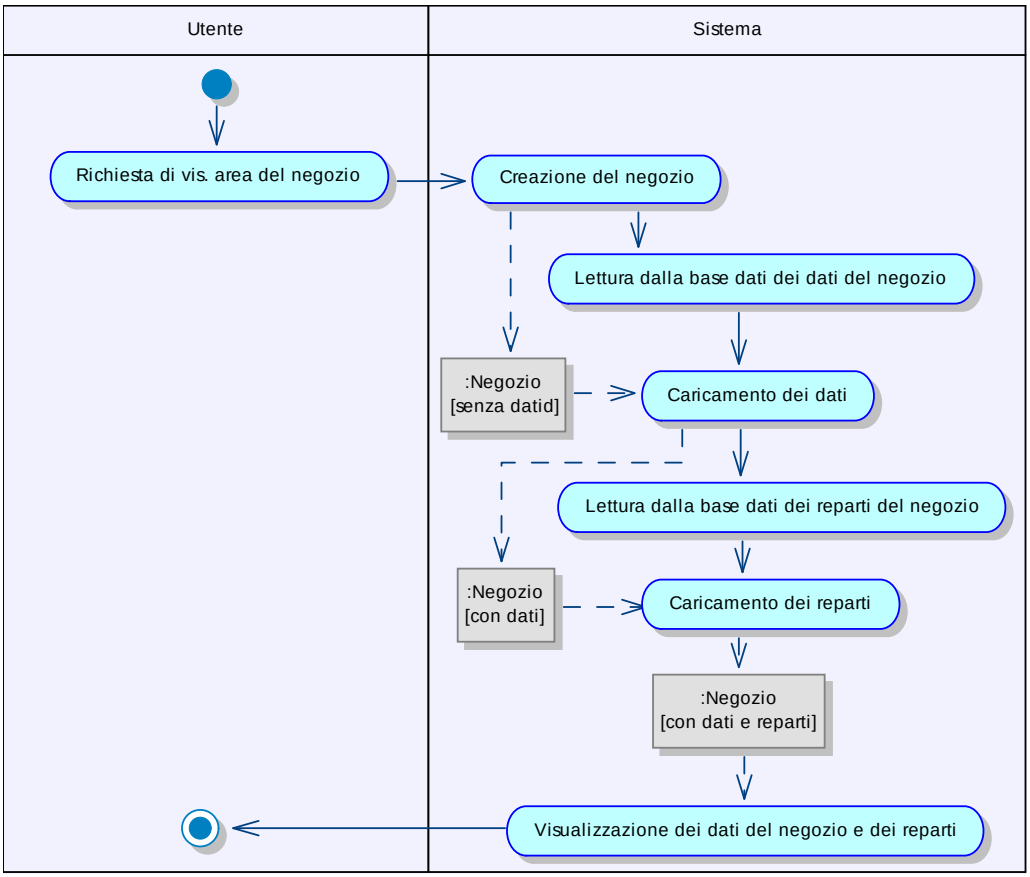

Figura 3.21: Activity Diagram Visualizz. dei Dati del Negozio e dei Reparti

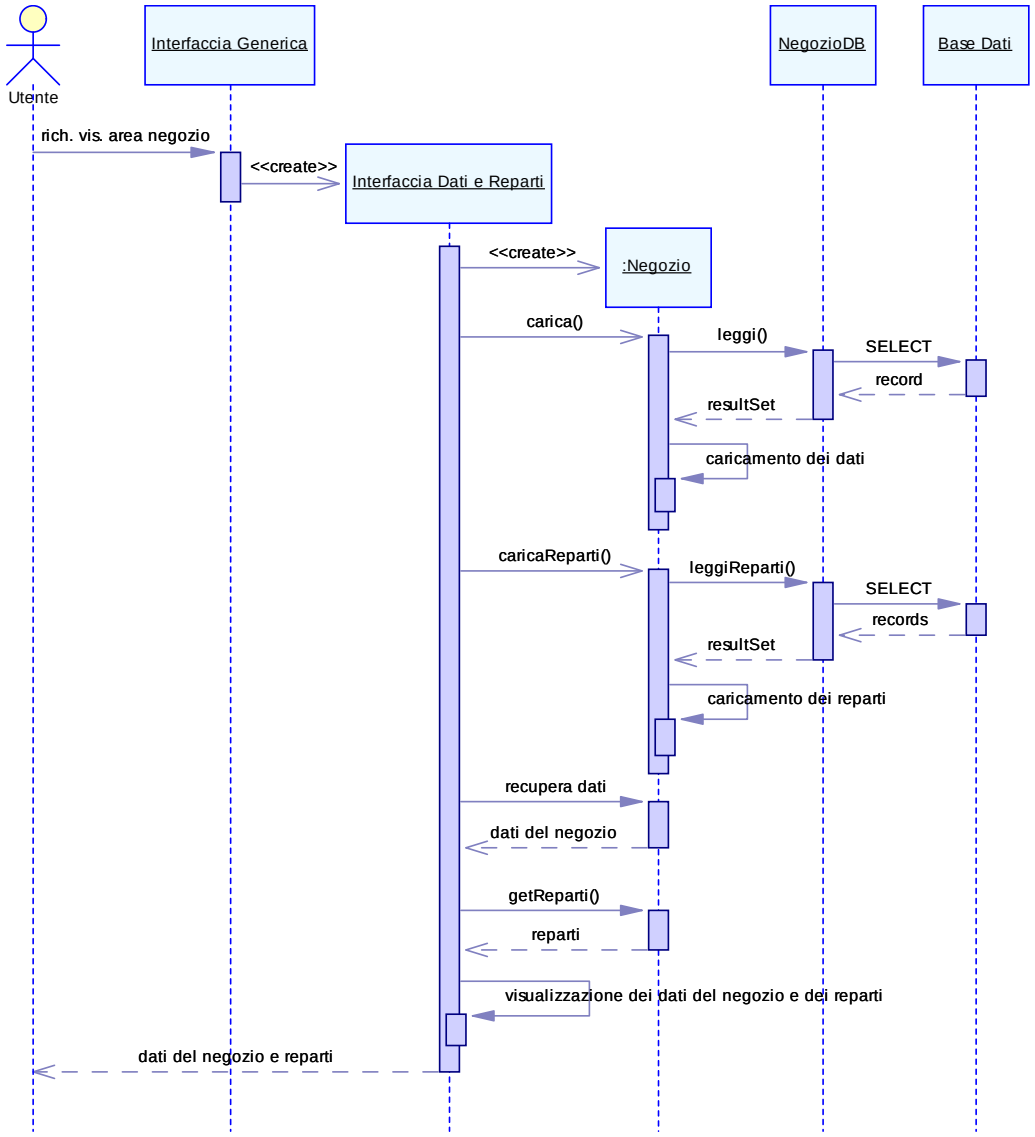

Figura 3.22: Sequence Diagram Visualizz. dei Dati del Negozio e dei Reparti

3.5.2.3 Visualizzazione del Dettaglio di un Articolo

Visualizza tutti i dati di un articolo. Il sistema crea un oggetto *Articolo*, legge dalla base dati i dati dell'articolo, li carica nell'oggetto *Articolo* e infine ne visualizza i dati nell'interfaccia.

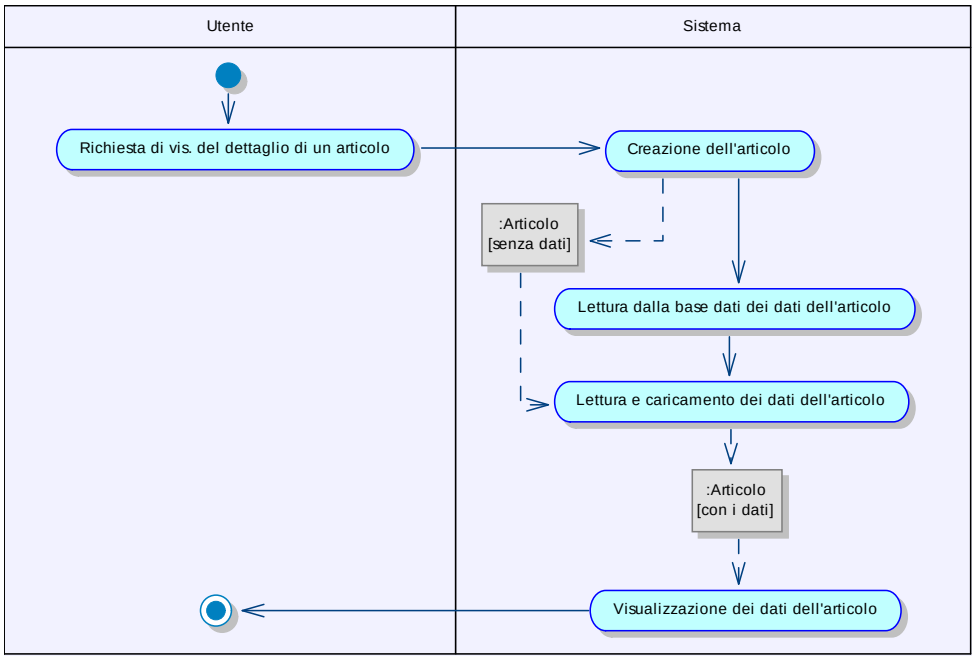

Figura 3.23: Activity Diagram Visualizzazione del Dettaglio di un Articolo

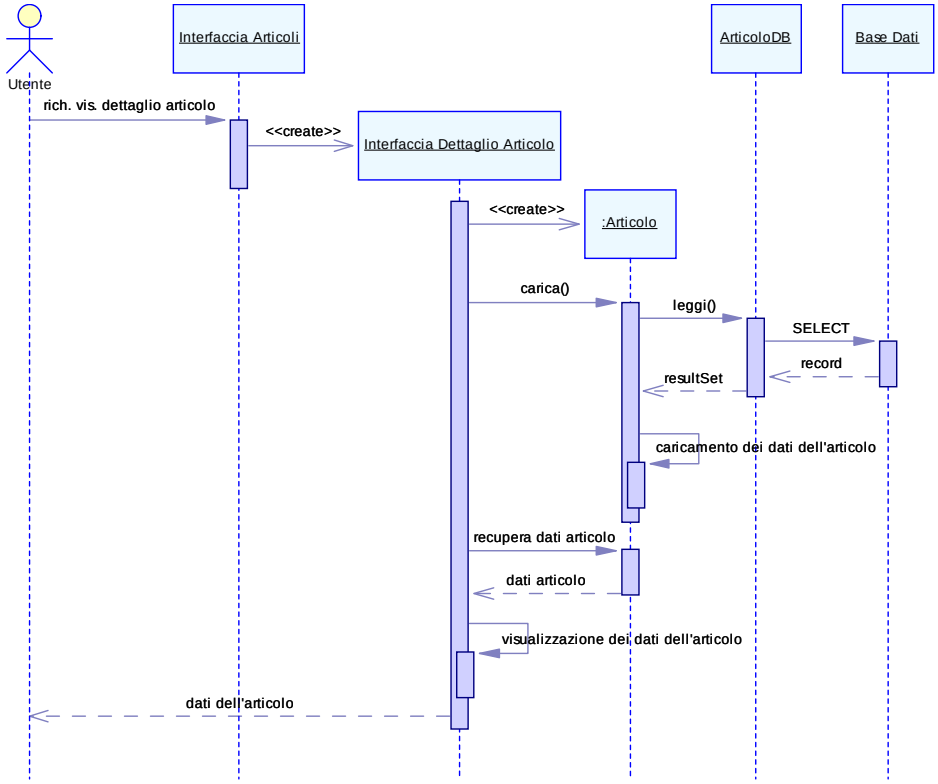

Figura 3.24: Sequence Diagram Visualizzazione del Dettaglio di un Articolo

3.5.2.4 Ricerca Negozi per Nome

Ricerca i negozi per nome. Per poter effettuare la ricerca l'*Utente* inserisce il nome dei negozi da cercare e richiede al sistema di effettuare la ricerca. Il sistema ricerca nella base dati i negozi, li carica nell'oggetto *NetShop* e ne visualizza i dati nell'interfaccia.

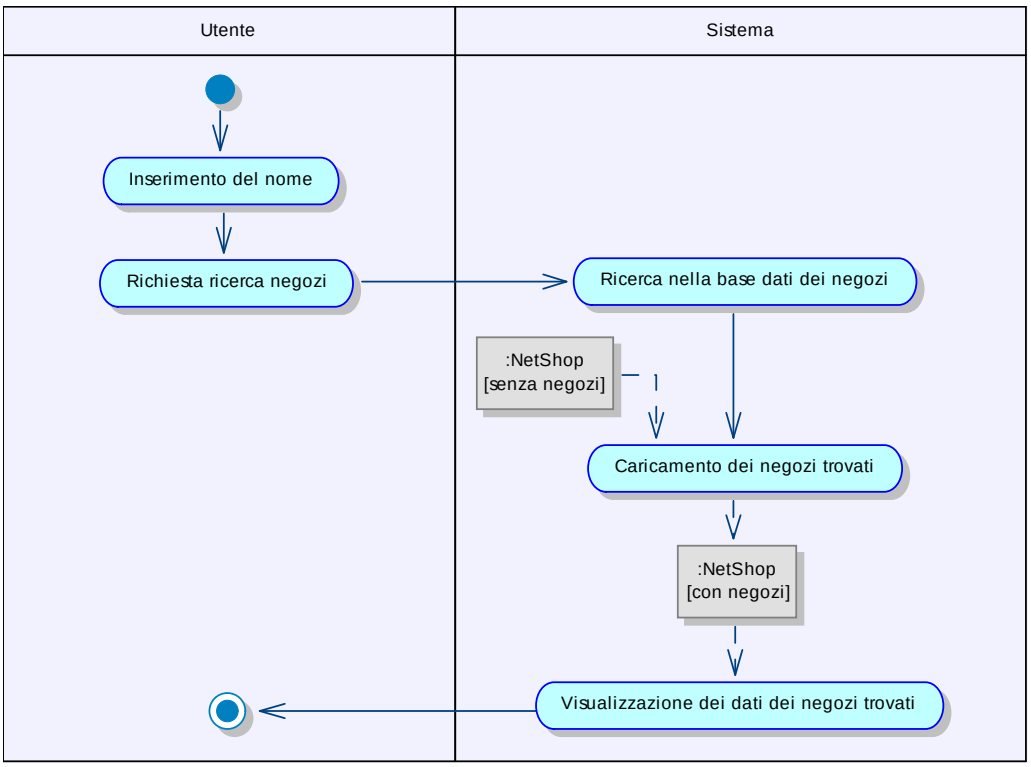

Figura 3.25: Activity Diagram Ricerca Negozi per Nome

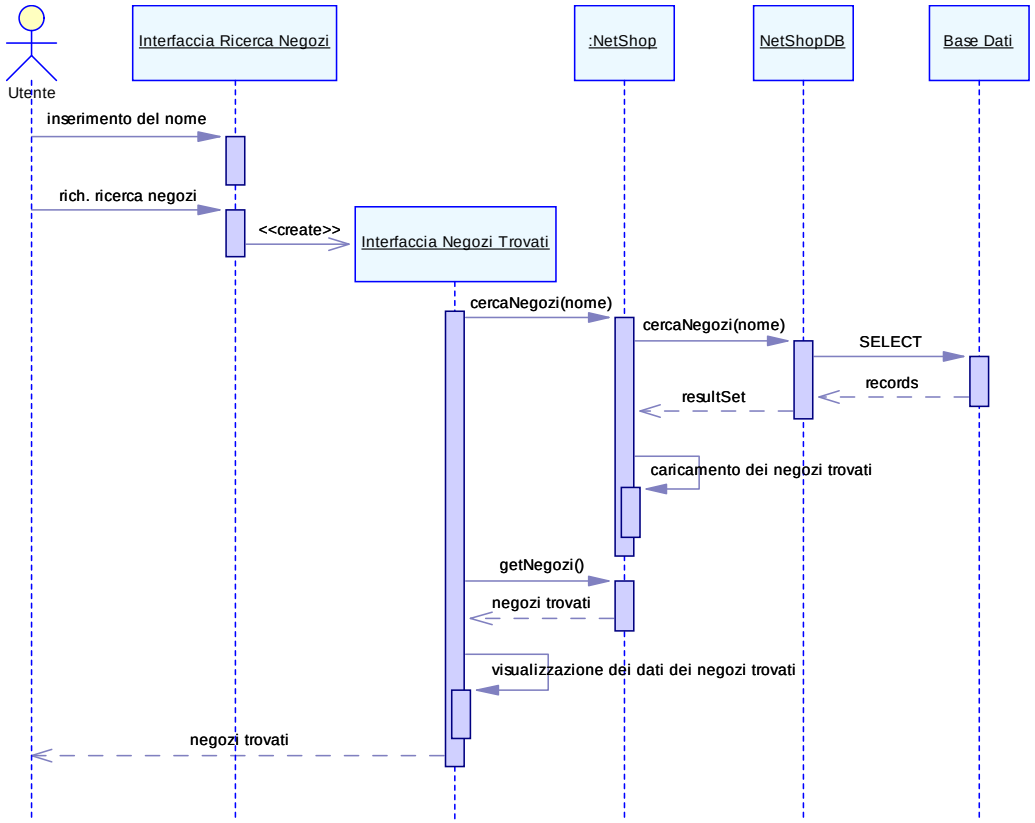

Figura 3.26: Sequence Diagram Ricerca Negozi per Nome

3.5.2.5 Ricerca Articoli per Nome e Categoria

Ricerca gli articoli all'interno di NetShop in base ad un nome specificato e ad una categoria scelta. Per poter effettuare la ricerca l'*Utente* inserisce il nome degli articoli da cercare, sceglie la categoria e richiede al sistema di effettuare la ricerca. Se non viene scelta nessuna categoria viene effettuata una ricerca per tutte le categorie. Il sistema ricerca nella base dati gli articoli, li carica nell'oggetto *NetShop* e ne visualizza i dati nell'interfaccia.

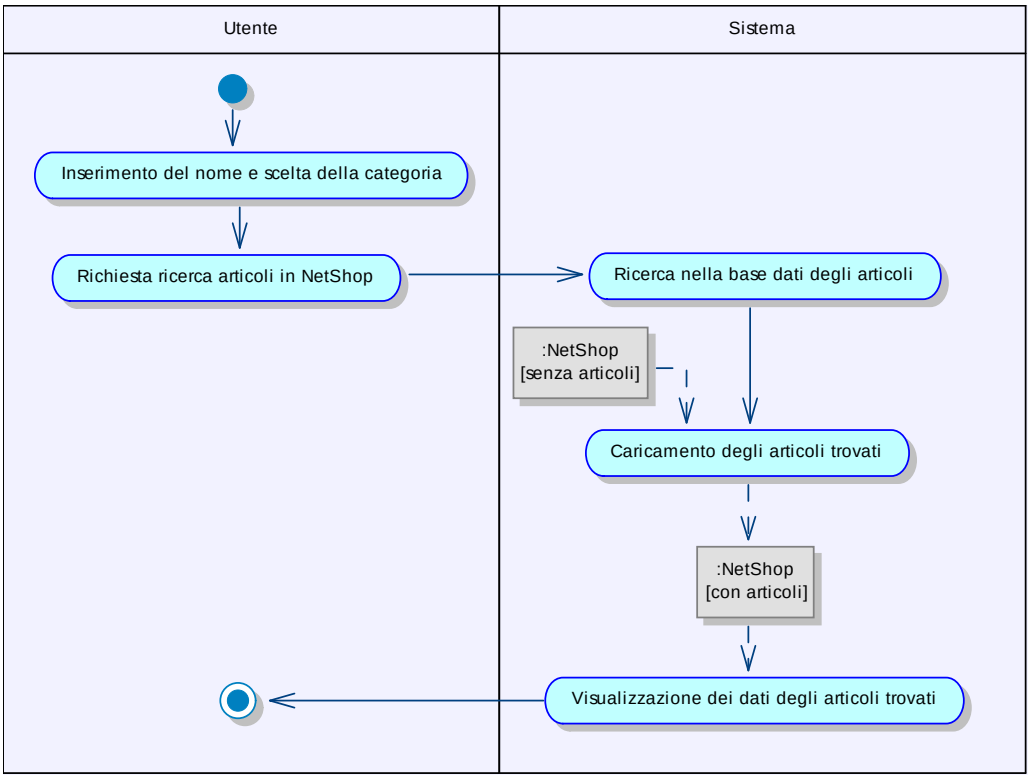

Figura 3.27: Activity Diagram Ricerca Articoli per Nome e Categoria in NetShop

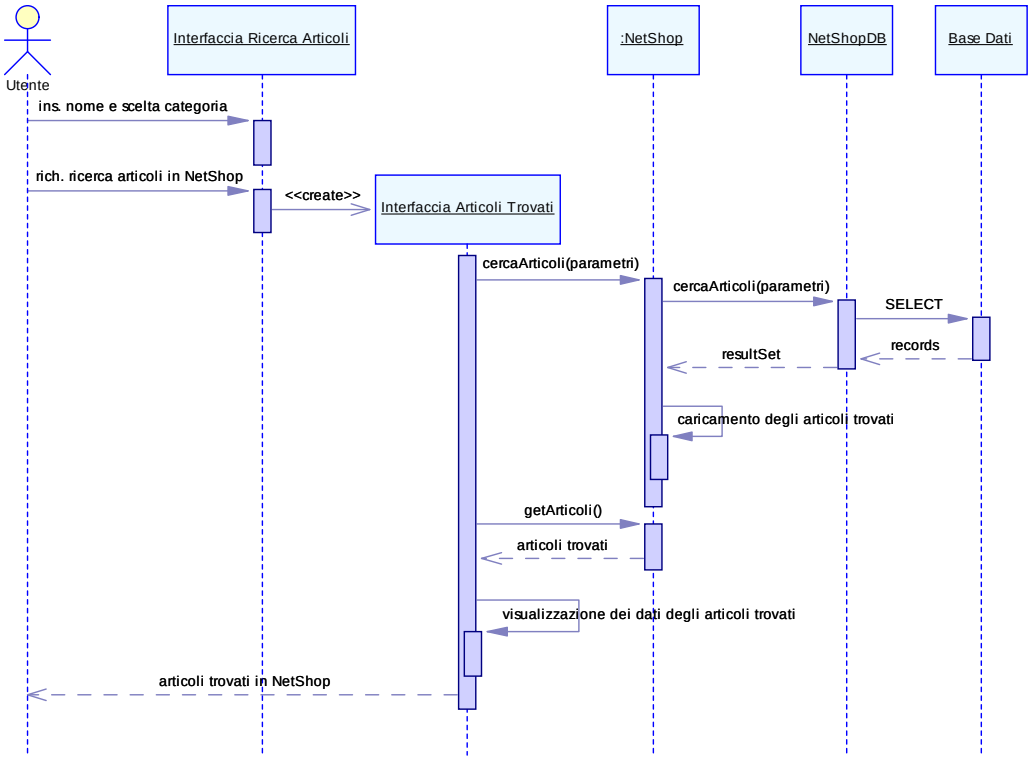

Figura 3.28: Sequence Diagram Ricerca Articoli per Nome e Categoria in NetShop

3.5.2.6 Inserimento di un Articolo nel Carrello

Permette di poter inserire un articolo nel carrello. Il sistema crea un oggetto *Articolo*, legge dalla base dati i dati dell'articolo e li carica nell'oggetto *Articolo*. Successivamente, il sistema crea un oggetto *Linea* a partire dall'articolo creato e verifica se l'articolo è già presente in qualche altra linea del carrello, se l'articolo è già presente in qualche altra linea aggiorna la quantità di questa ultima altrimenti inserisce la nuova linea all'interno del carrello.

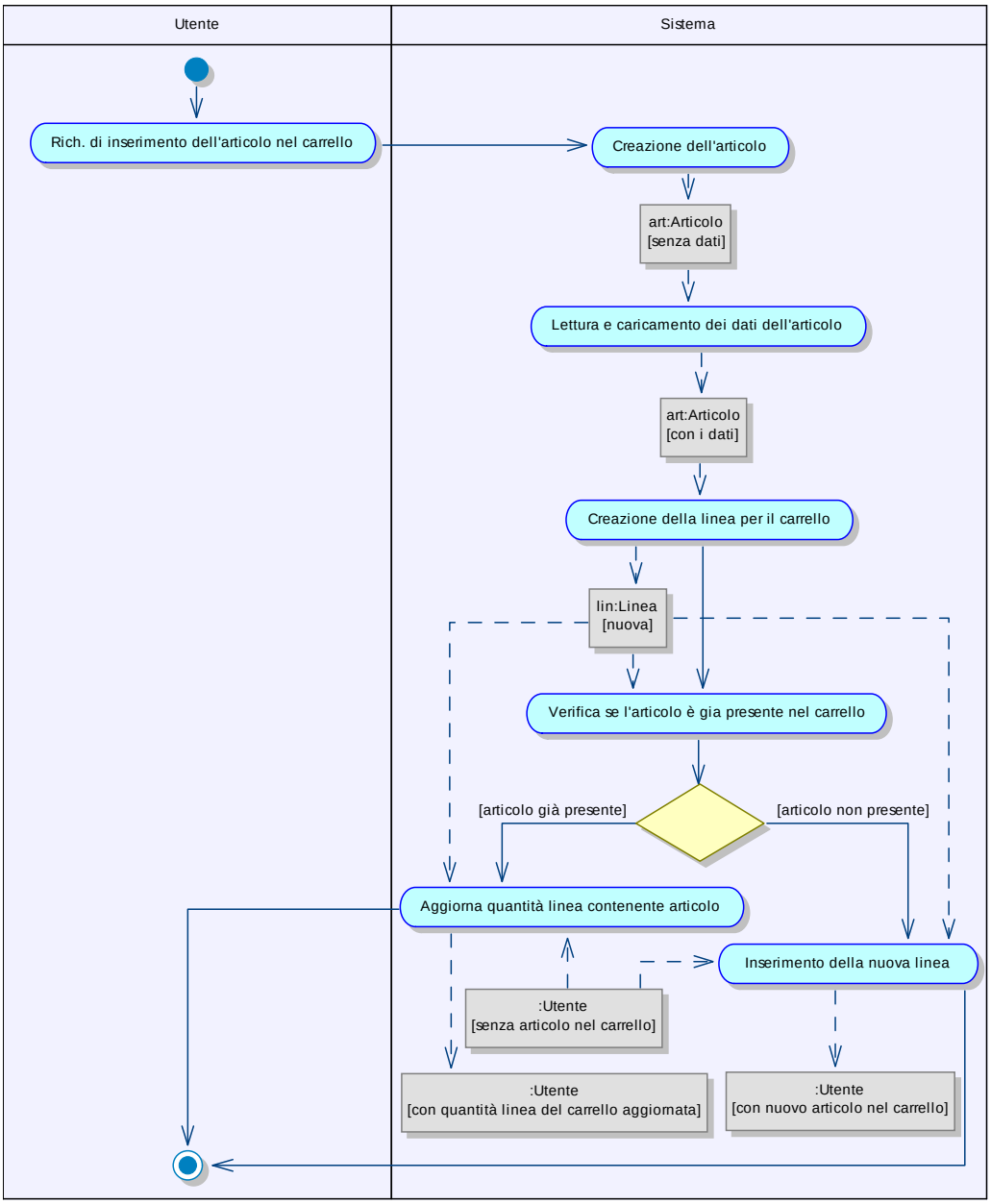

Figura 3.29: Activity Diagram Inserimento di un Articolo nel Carrello

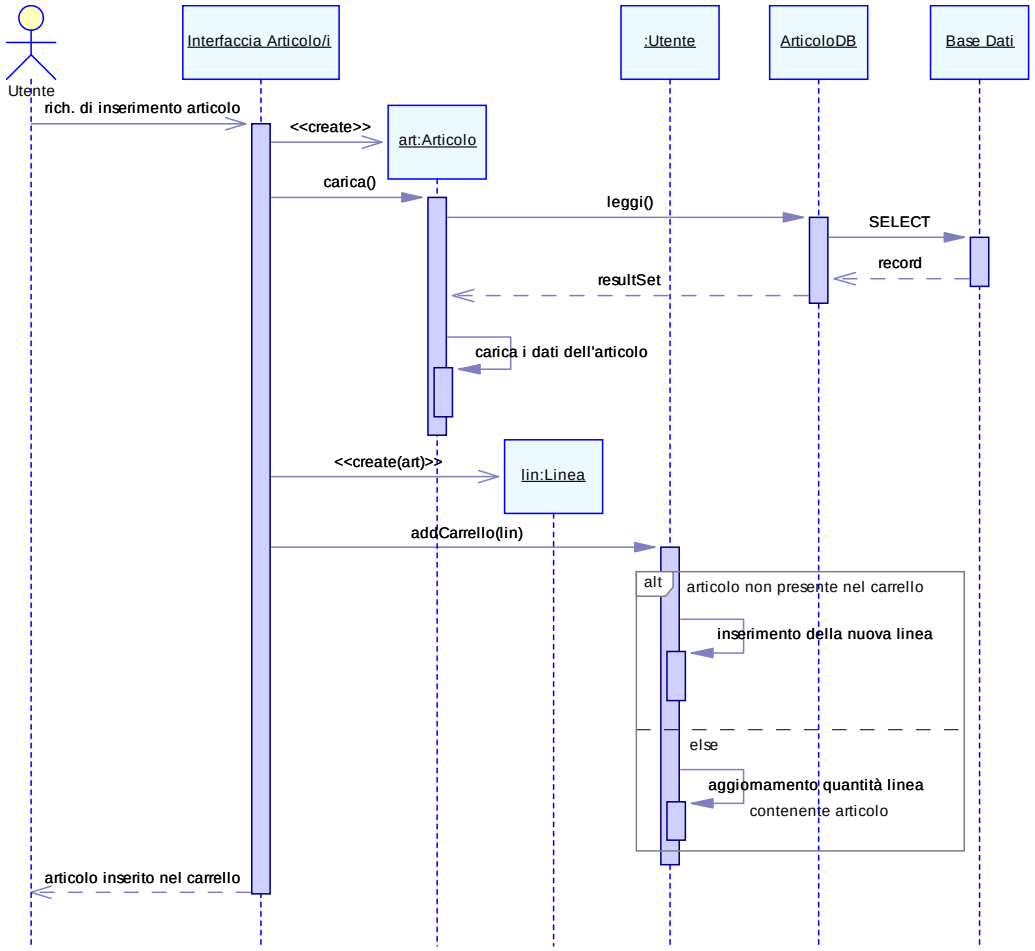

Figura 3.30: Sequence Diagram Inserimento di un Articolo nel Carrello

3.5.2.7 Registrazione come Cliente

Permette all'*Utente* di potersi registrare come cliente. L'*Utente* inserisce i dati del cliente e richiede l'effettuazione della registrazione. Il sistema verifica la correttezza dei dati inseriti, se i dati non sono corretti visualizza un messaggio di errore altrimenti continua con la registrazione. Il sistema crea un nuovo oggetto *Cliente* e verifica che l'username non sia già utilizzata, se l'username è già utilizzata visualizza un messaggio di errore altrimenti continua con la registrazione. Il sistema inserisce nella base dati il nuovo cliente, recupera il carrello dall'utente, autorizza il cliente per l'accesso al sistema, lo inserisce nella sessione ed infine invia un'email per la registrazione effettuata al cliente e all'amministratore.

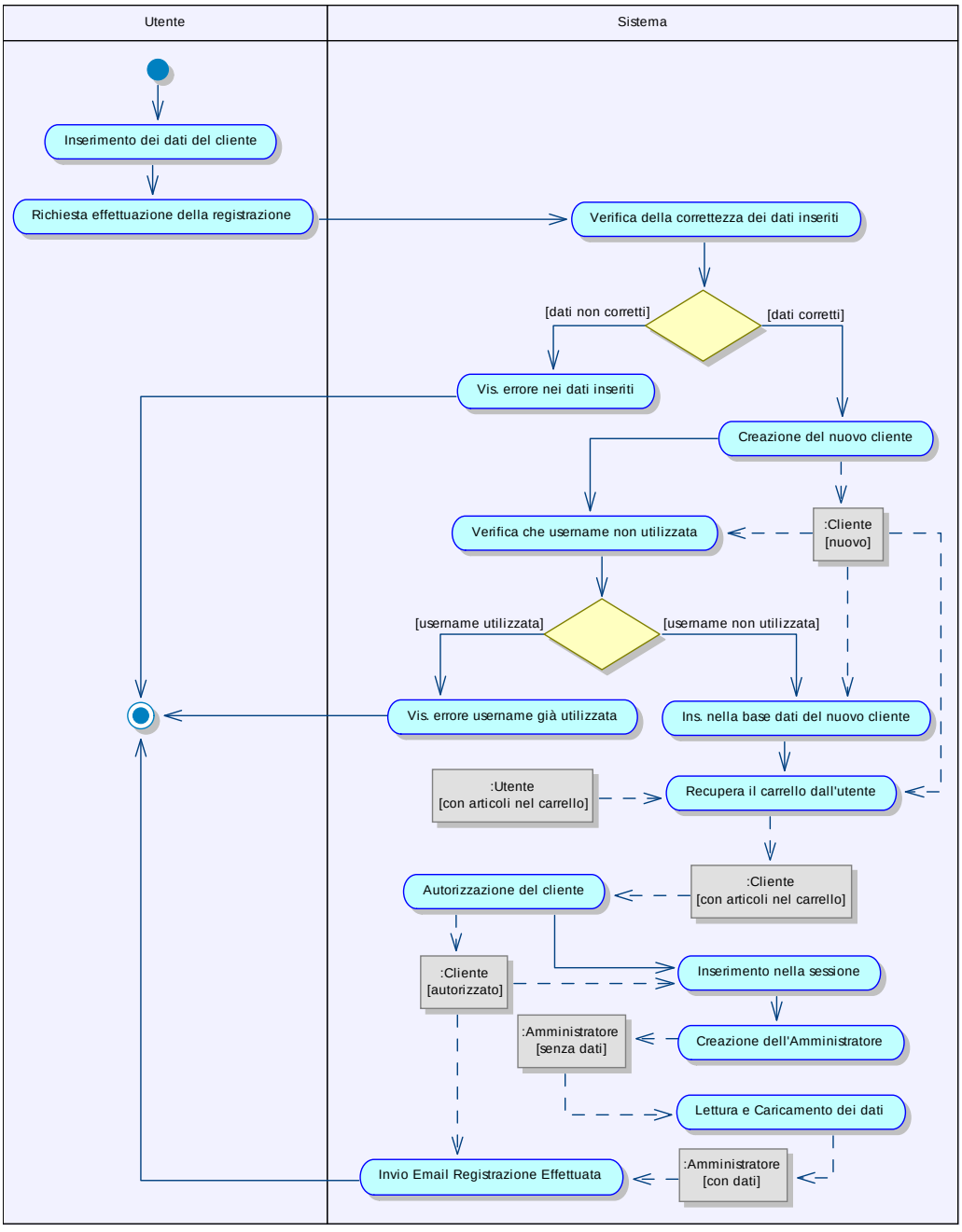

Figura 3.31: Activity Diagram Registrazione come Cliente

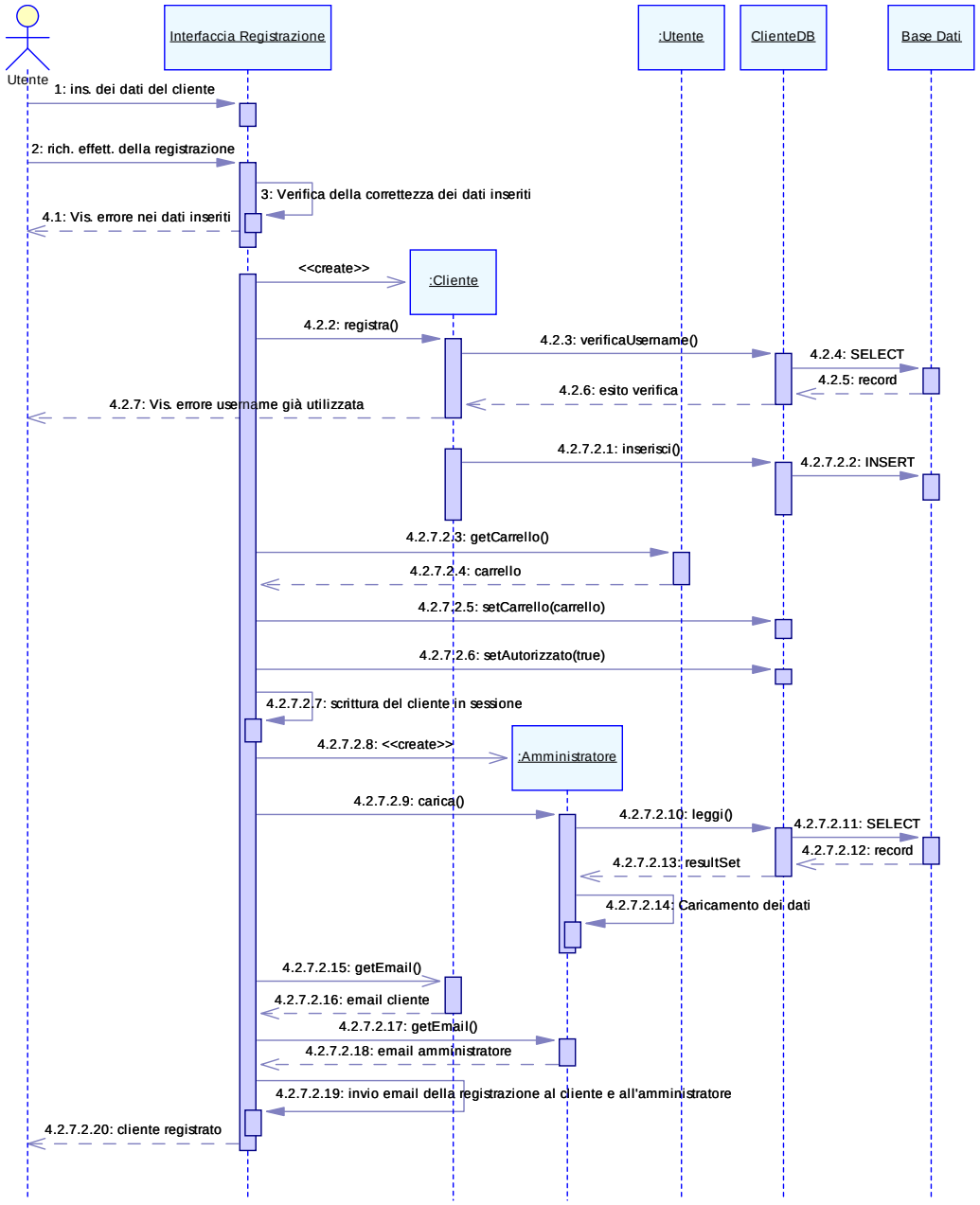

Figura 3.32: Sequence Diagram Registrazione come Cliente

3.5.2.8 Login

Permette di poter effettuare il login al sistema come cliente, gestore o amministratore. L'*Utente* inserisce l'username e la password e richiede l'effettuazione del login. Il sistema verifica la correttezza dei dati inseriti, nel caso in cui i dati non sono corretti visualizza un messaggio di errore altrimenti continua con il login. Il sistema crea un oggetto *Cliente* con i dati del login e ne effettua la lettura e il caricamento dei dati, nel caso in cui il cliente non sia presente o sia attualmente bloccato visualizza un messaggio di errore altrimenti continua con il login. Il sistema recupera il carrello dall'utente e autorizza il nuovo cliente all'accesso al sistema. Resta solo da capire se si tratta di un cliente normale, di un gestore o di un amministratore, per fare ciò il sistema verifica il tipo del cliente; se si tratta di un cliente inserisce nella sessione l'oggetto *Cliente*; se si tratta di un amministratore crea un oggetto *Amministratore* a partire dai dati del cliente e lo si inserisce nella sessione; se invece si tratta di un gestore crea un oggetto *Gestore* a partire dai dati del cliente, legge e carica i dati del negozio gestito, rimuove dal carrello gli articoli del negozio gestito e infine effettua l'inserimento nella sessione.

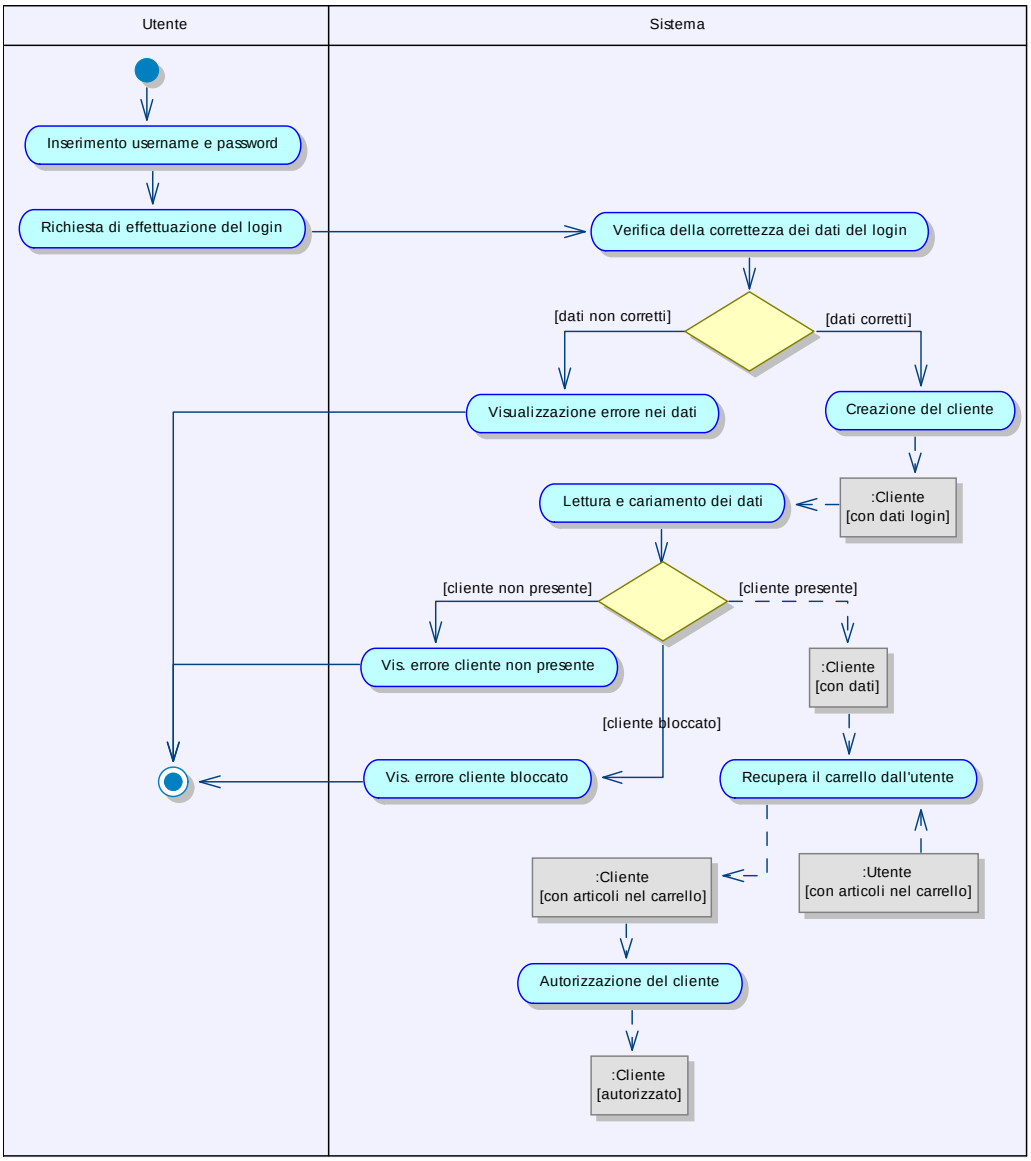

Figura 3.33: Activity Diagram Login (1a parte)

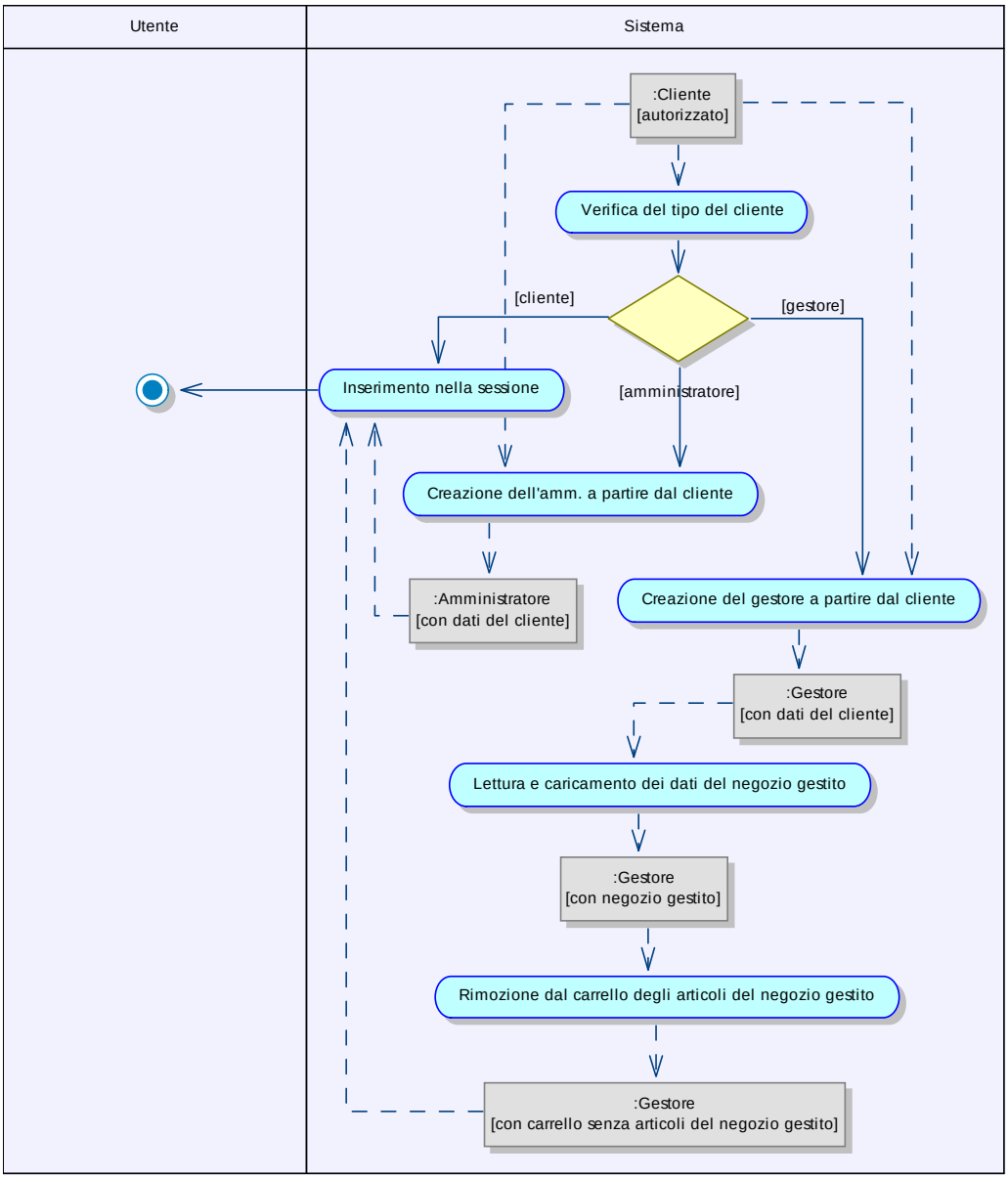

Figura 3.34: Activity Diagram Login (2a parte)

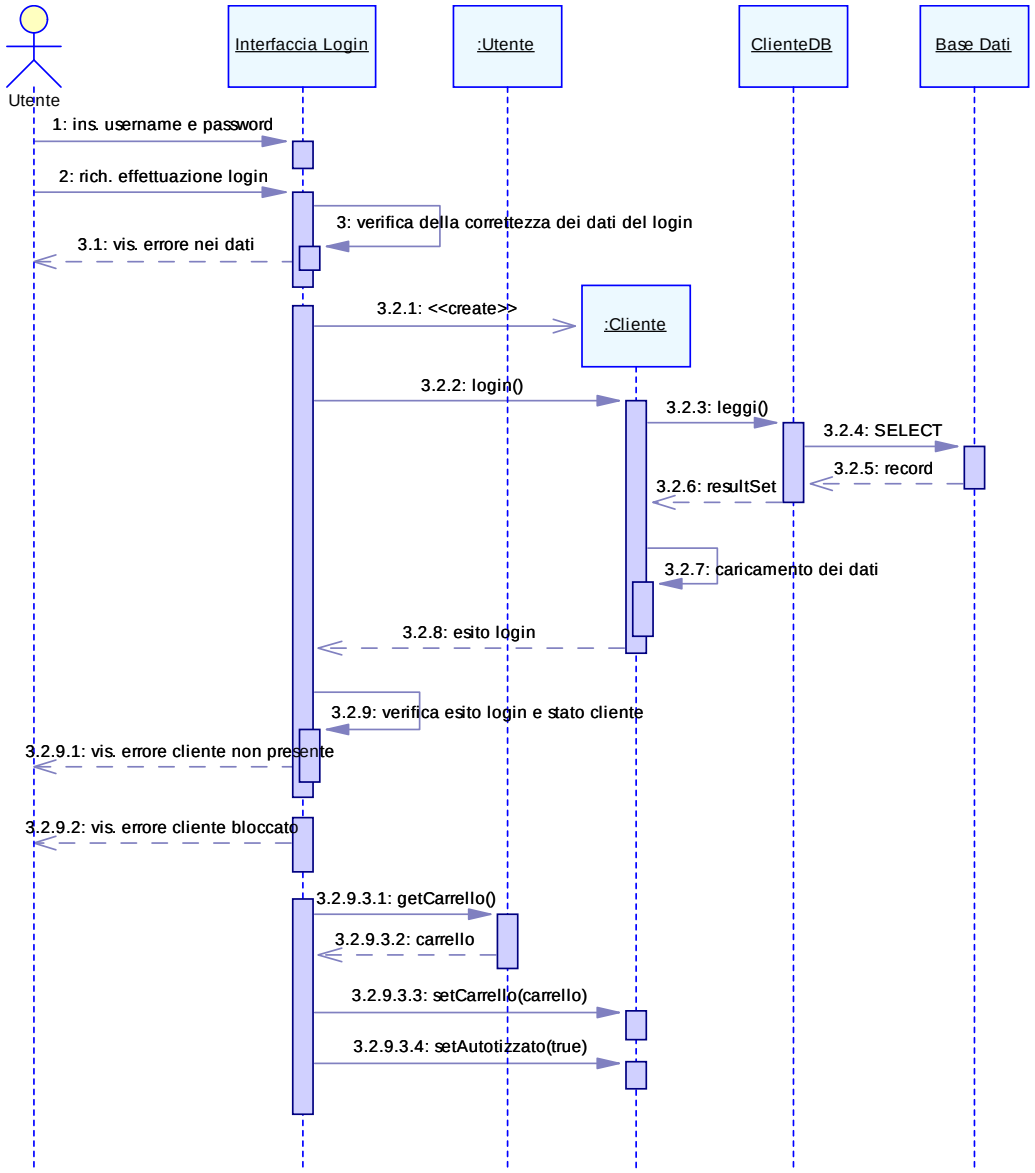

Figura 3.35: Sequence Diagram Login (1a parte)

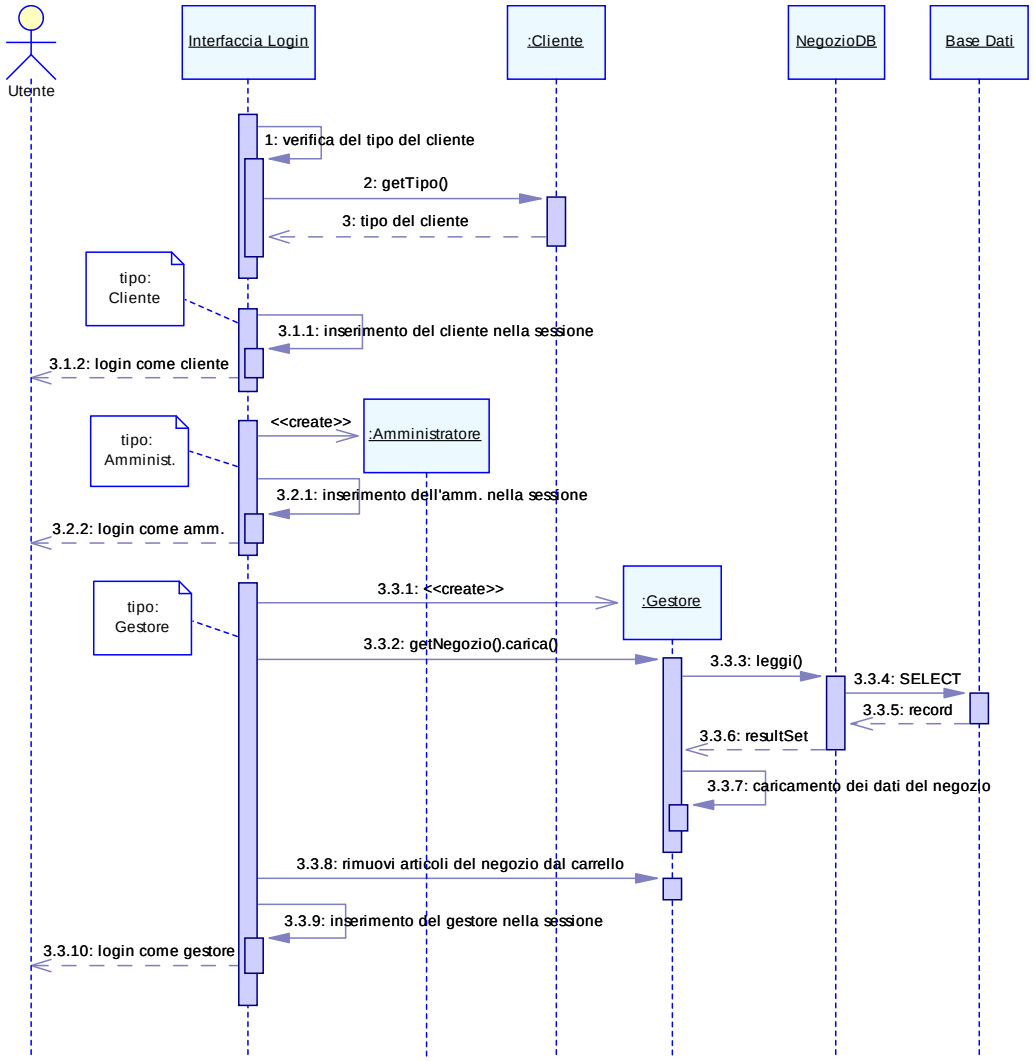

Figura 3.36: Sequence Diagram Login (2a parte)

3.5.2.9 Effettuazione Ordine

Permette al cliente di poter effettuare un ordine degli articoli del carrello verso un determinato negozio. Dopo che il *Cliente* ha richiesto l'effettuazione di un'ordine, il sistema crea un oggetto *Negozio*, legge e carica i dati del negozio, del suo gestore e delle sue modalità di consegna e infine visualizza le modalità di spedizione e di pagamento, offerte dal negozio. A partire dalle modalità di spedizione e pagamento visualizzate, il *Cliente* sceglie una modalità di pagamento e di spedizione e continua con l'effettuazione dell'ordine. A questo punto, il sistema crea un oggetto *Ordine*, ne inserisce le linee, a partire dagli articoli presenti nel carrello del cliente e visualizza il riepilogo dell'ordine. Dopo la visualizzazione del riepilogo dell'ordine, il *Cliente* può scegliere o meno di concludere l'ordine. Nel caso in cui il cliente scelga di concludere l'ordine, il sistema effettua l'ordine, andandolo ad inserire nella base dati, poi invia un email sia al cliente che al gestore del negozio e infine rimuove gli articoli acquistati dal carrello del cliente.

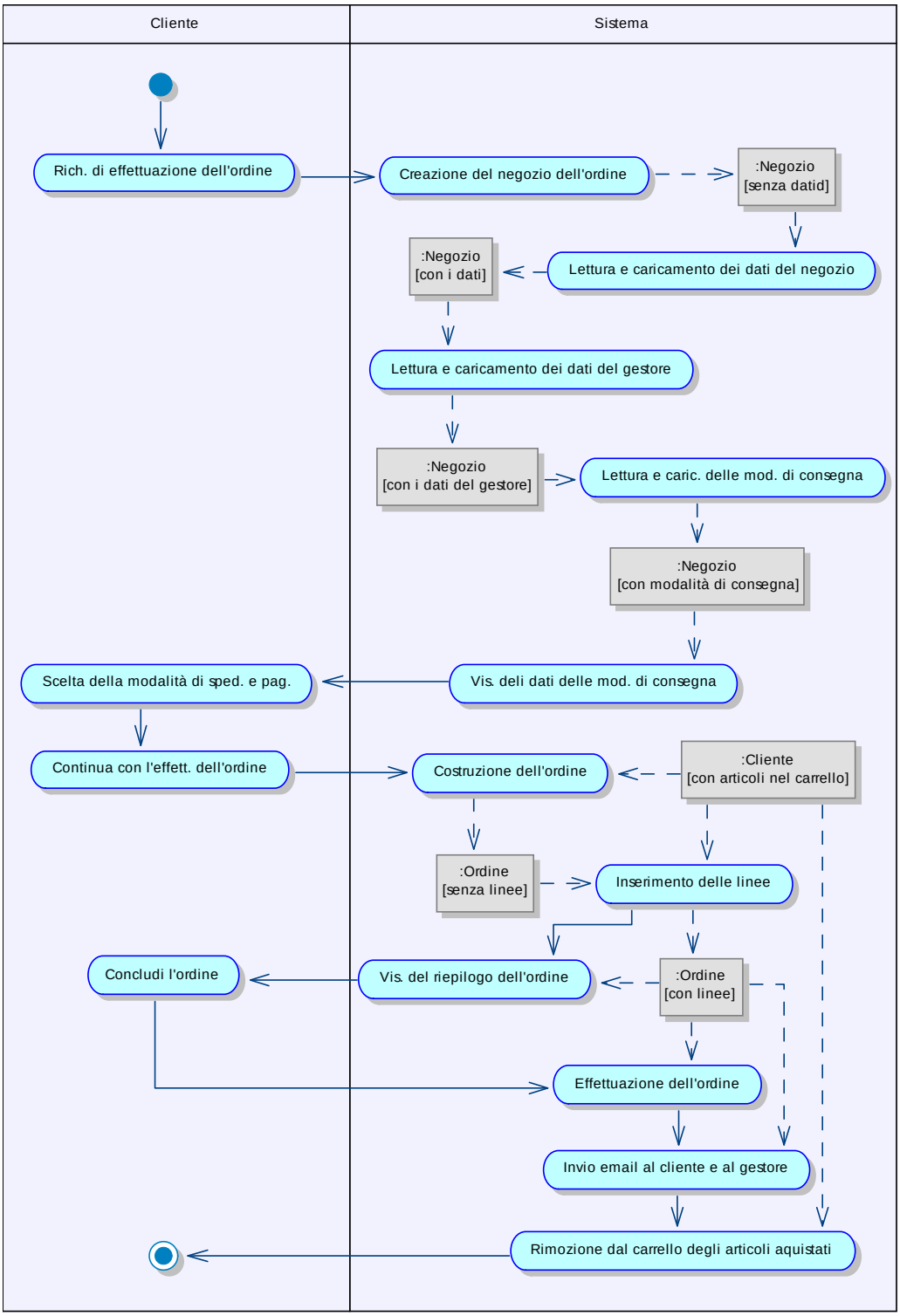

Figura 3.37: Activity Diagram Effettuazione Ordine

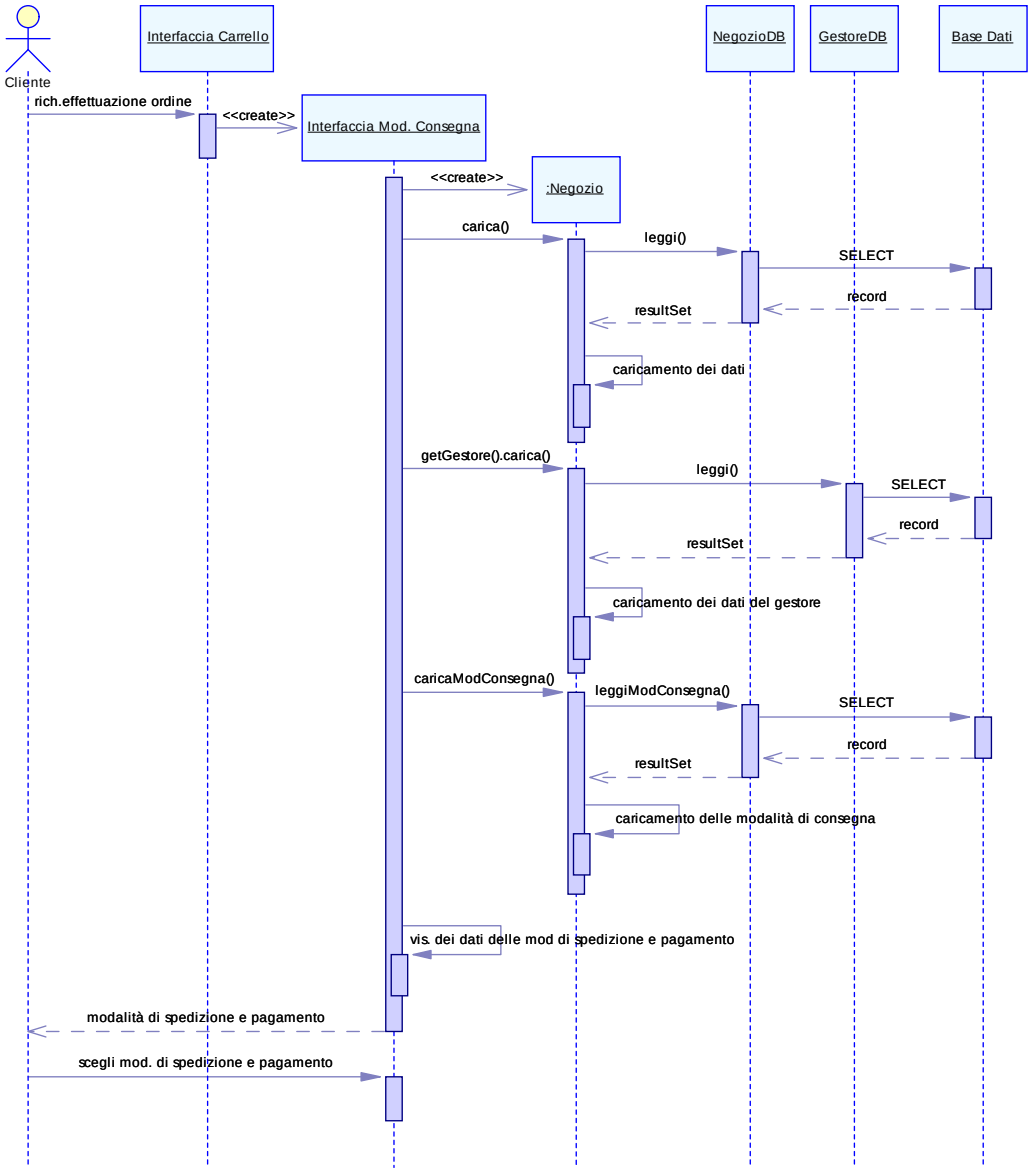

Figura 3.38: Sequence Diagram Effettuazione Ordine (1a parte)

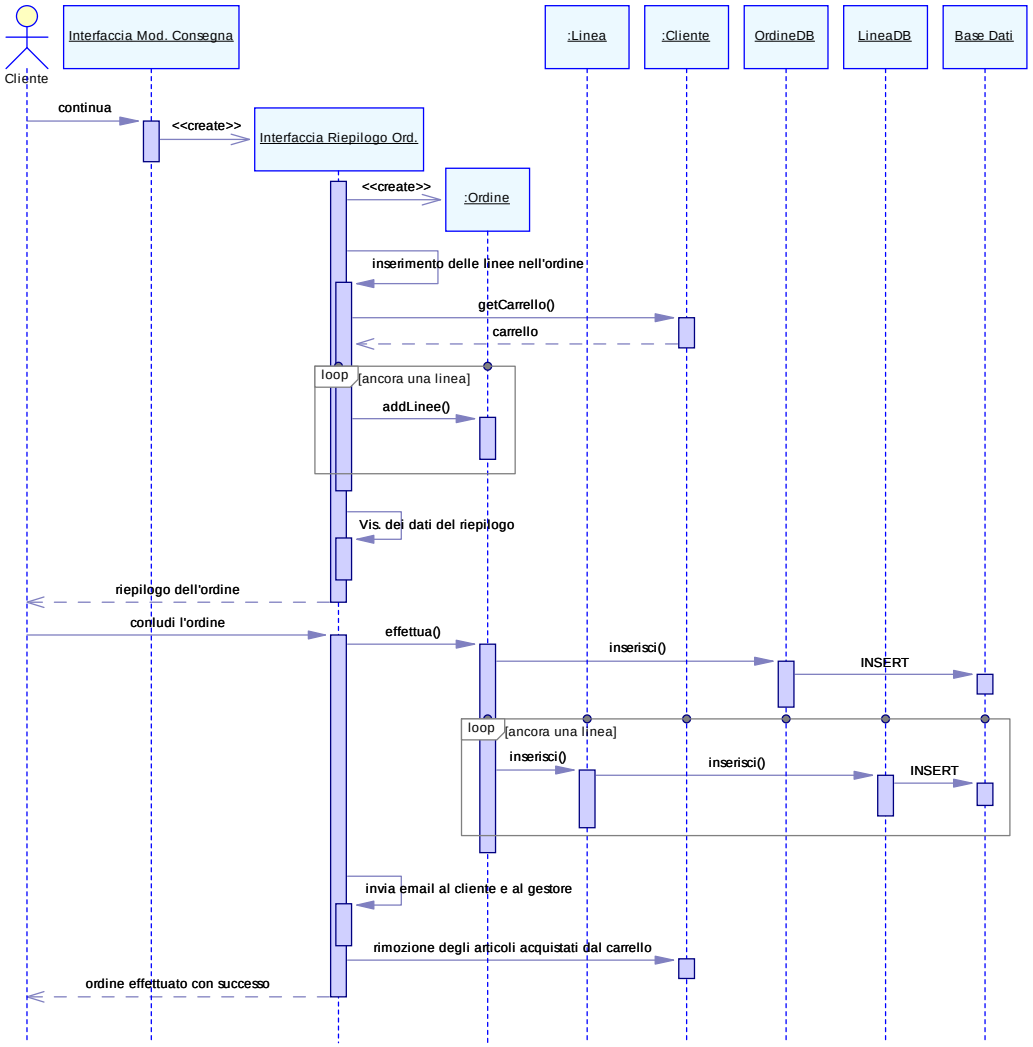

Figura 3.39: Sequence Diagram Effettuazione Ordine (2a parte)

3.5.2.10 Inserimento di un Nuovo Articolo

Permette al *Gestore* di poter inserire un nuovo articolo nel suo catalogo articoli. Il *Gestore* inserisce i dati dell'articolo ed effettua la richiesta di inserimento del nuovo articolo. Il sistema verifica la correttezza dei dati inseriti, nel caso in cui i dati non sono corretti visualizza un messaggio di errore altrimenti continua con l'inserimento. Il sistema crea un oggetto *Articolo*, a partire dai dati inseriti e inserisce il nuovo articolo all'interno della base dati.

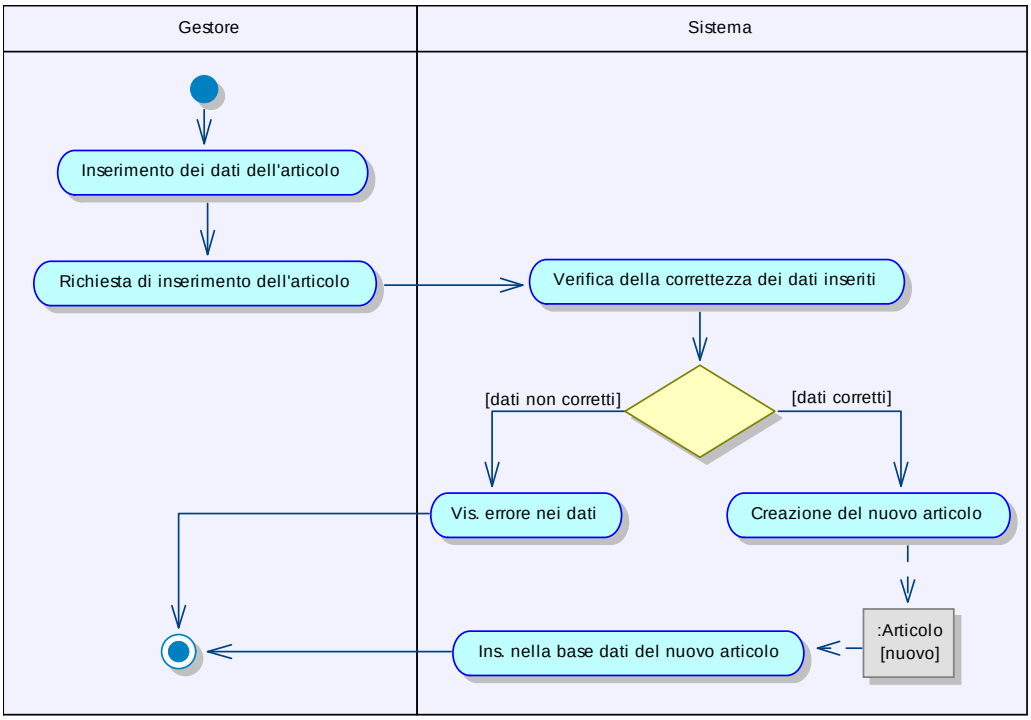

Figura 3.40: Activity Diagram Inserimento di un Nuovo Articolo

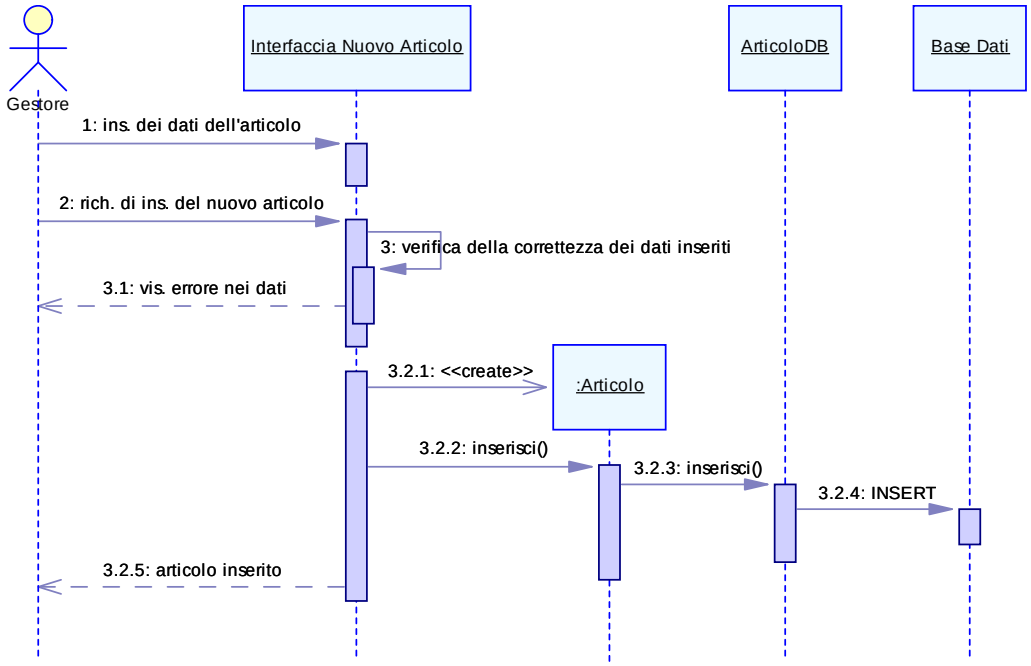

Figura 3.41: Sequence Diagram Inserimento di un Nuovo Articolo

3.5.2.11 Visualizzazione degli Ordini Ricevuti

Permette al *Gestore* di poter visualizzare gli ordini ricevuti. Dopo che Il *Gestore* ha richiesto la visualizzazione degli ordini ricevuti, il sistema legge dalla base dati gli ordini ricevuti, li carica all'interno del negozio del gestore e infine ne visualizza i dati nell'interfaccia.

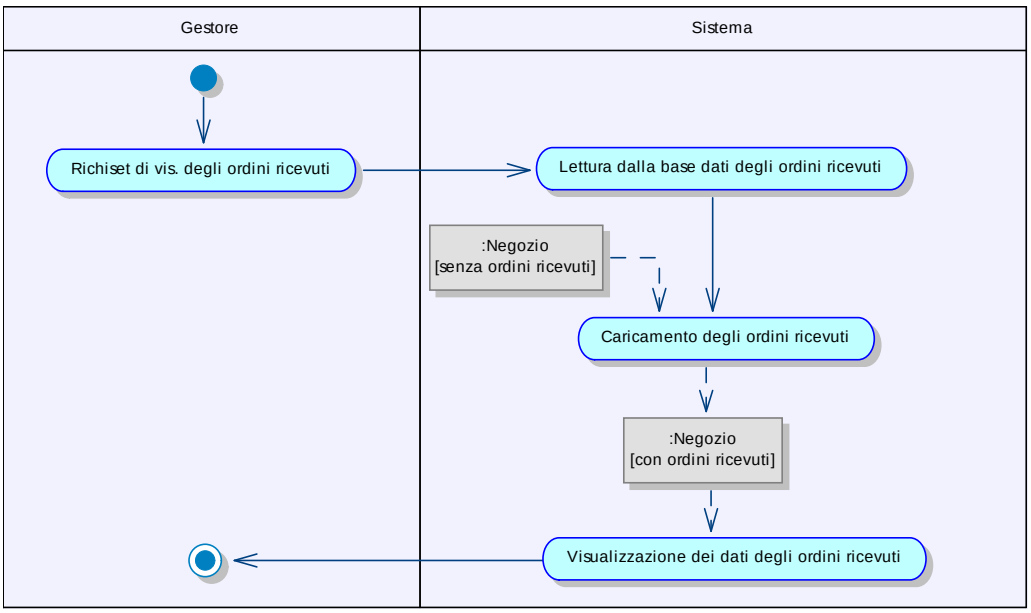

Figura 3.42: Activity Diagram Visualizzazione degli Ordini Ricevuti

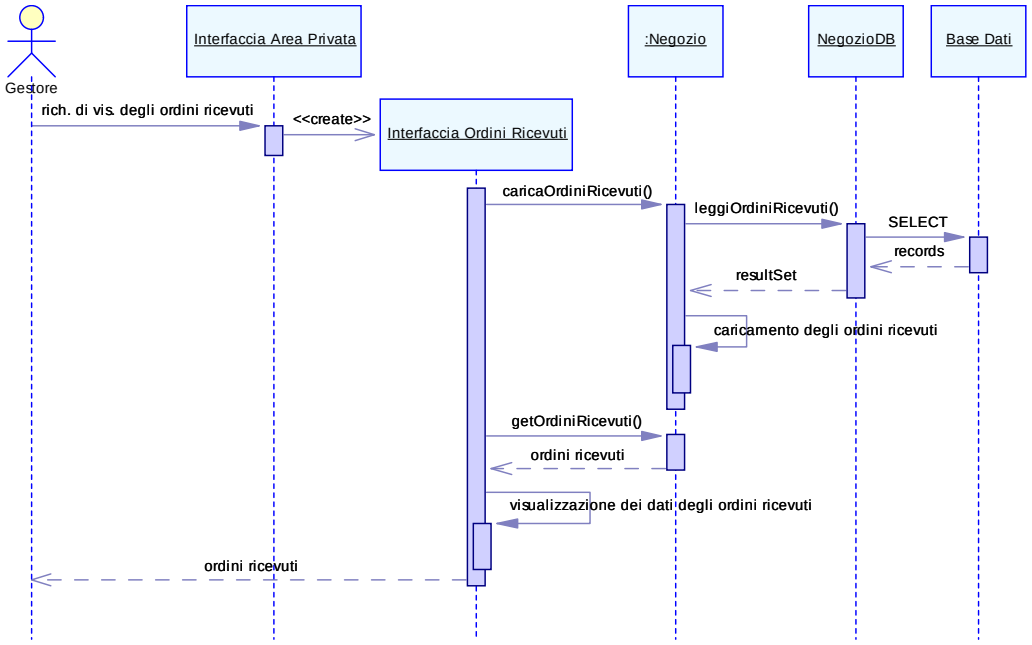

Figura 3.43: Sequence Diagram Visualizzazione degli Ordini Ricevuti

3.5.2.12 Visualizzazione dei Clienti

Permette all'*Amministratore* di poter visualizzare i clienti di NetSop. Dopo che l'*Amministratore* ha richiesto la visualizzazione dei clienti, il sistema legge dalla base dati i clienti, li carica all'interno dell'oggetto *NetShop* e infine ne visualizza i dati nell'interfaccia.

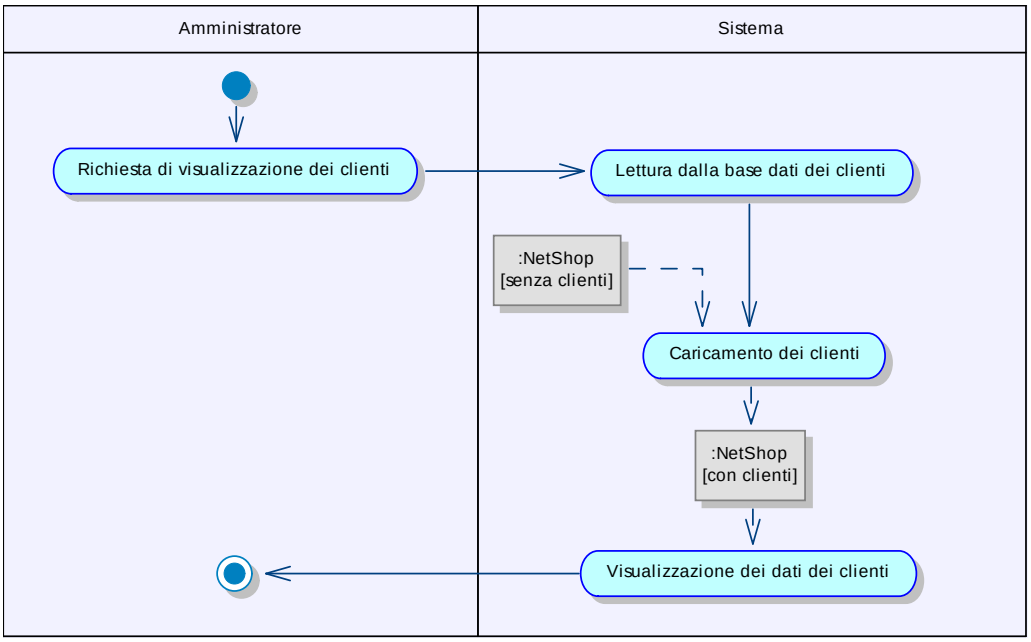

Figura 3.44: Activity Diagram Visualizzazione dei Clienti

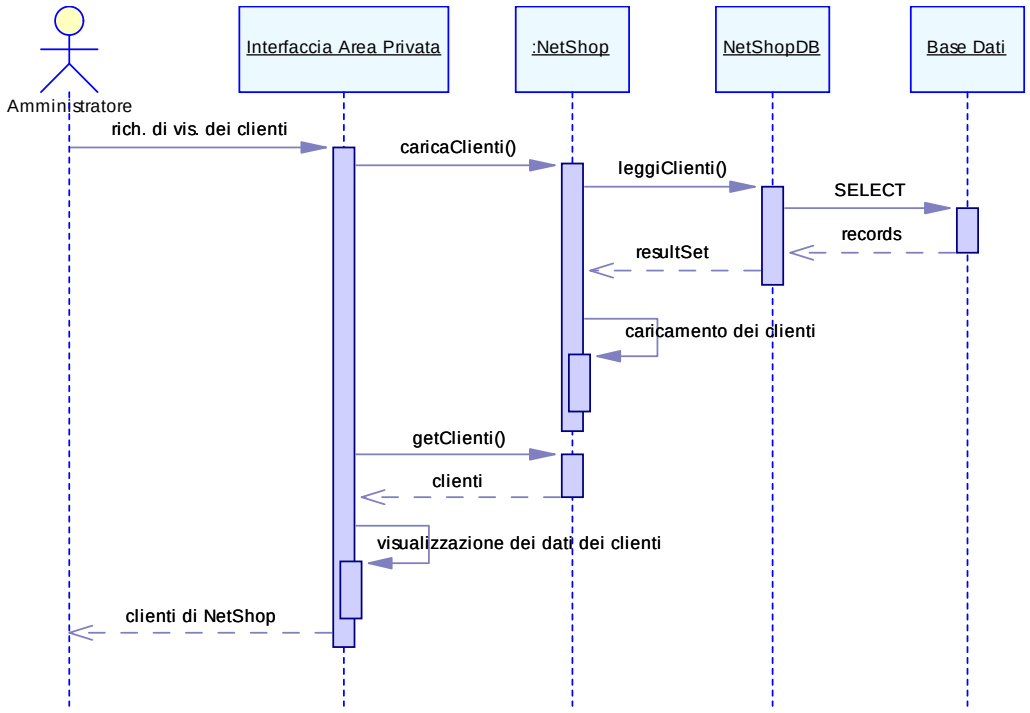

Figura 3.45: Sequence Diagram Visualizzazione dei Clienti

Capitolo 4

Implementazione

4.1 Introduzione

La Web application precedentemente progettata è stata implementata con il framework Jakarta Struts, mentre la base dati è stata realizzata con il DBMS MySQL. Gli strumenti software utilizzati per lo sviluppo e gli ambienti di esecuzione utilizzati sono stati i seguenti:

- Piattaforma *J2SE 5.0 (Java 2 Standard Edition)*;
- Ambiente IDE *Eclipse 3.2* con i plug-in di *Exadel Studio 4.0* per lo sviluppo grafico dei file di configurazione;
- Web Container *Apache Jakarta Tomcat 5.5*;
- *MySQL GUI Tool 1.2,* tool di programmi per la creazione e la gestione della base dati MySQL;
- *Macromedia Dreamweaver 8* e *Fireworks 8* per la creazione delle pagine JSP e delle immagini.

La piattaforma J2SE rappresenta la base per lo sviluppo e l'esecuzione delle applicazioni Java, in quanto mette a disposizione la *JVM (Java Virtual Machine)*, oltre a tutte le classi principali e le librerie del linguaggio.

Per l'ambiente di sviluppo *IDE (Integrated Development Environment)*, la scelta è caduta sul progetto Open Source Eclipse 3.2, dopo un rapido confronto con l'antagonista NetBeans. Ha nettamente prevalso il primo, considerando il supporto fornito per lo sviluppo di applicazioni Web basate appunto su Struts, grazie al plug-in free di Exadel Studio 4.0.

Per quanto riguarda il Web Container, la scelta è ovviamente caduta sul progetto Open Source Jakarta Apache Tomcat 5.5 sicuramente tra i migliori per l'esecuzione di applicazioni Web realizzate in ambiente Java.

E' altresì necessario specificare che, per quanto riguarda Struts, è stata utilizzata la versione 1.1, mentre per MySQL è stata utilizzata la versione 5.0.

4.2 Struttura dell'applicazione

Il primo aspetto preso in considerazione riguarda la struttura dell'applicazione, che essendo stata sviluppata con Struts ha richiesto l'implementazione del Model, della View e del Controller.

Per l'implementazione del Model sono stati realizzati due packages, uno contenente le classi del livello *business-logic* ed un altro quelle per il sottolivello *data-access*.

Per l'implementazione del Controller sono stati realizzati quattro packages, contenenti le action per eseguire le funzionalità previste per l'utente, il cliente, il gestore e l'amministratore, inoltre è stato realizzato un package di base contenente le funzionalità comuni a più action.

Per l'implementazione della View sono state realizzate le pagine JSP necessarie per la visualizzazione e l'acquisizione dei dati, separate in quattro cartelle relative al ruolo dell'utilizzatore; poi sono state realizzate le immagini e il foglio di stile (*stylesheet.css*) per la formattazione grafica che sono stati posti all'interno della cartella *Pics*.

Sempre per l'implementazione della View sono stati utilizzati i seguenti

framework, inclusi nella distribuzione di Struts e ad esso integrati attraverso il meccanismo dei plug-in:

- *Tiles*: framework per la definizione del layout delle pagine;
- *Validator*: framework per la validazione dei campi di input.

Un' ulteriore considerazione per la View, riguarda l'Internazionalizzazione (I18N), supportata dalla Web application attraverso la definizione di due Resource Bundle, uno per la localizzazione di default, ossia l'Italia, e l'altro per un'ulteriore localizzazione supportata, in lingua inglese.

Inoltre è stato realizzato un package contenente le classi di uso generale per ogni livello dell'applicazione.

Per quanto riguardo la sicurezza, è stata utilizzata l'estensione di Struts per la sicurezza (*SSLext*).

Per il logging è stato utilizzato *log4j*.

Per quanto riguarda le librerie, oltre a quelle proprie di Stuts, sono stati utilizzate le seguenti librerie:

- *JDBC MySQL Connector*, che contiene le classi relative ai driver *JDBC (Java DataBase Connectivity)* per l'accesso alla base di dati realizzata con il DBMS MySQL;
- *JavaMail*, che contiene le classi per l'invio delle mail basato su *JAF (JavaBeans Activation Framework)*;
- *SSLext*, che contiente le classi che estendono Struts per l'utilizzo del protocollo SSL;
- *Log4j*, che contiene le classi per effettuare il logging con log4j.

L'architettura del sistema, oltre al file di configurazione dell'applicazione (*web.xml*), ha previsto un unico modulo al quale è ovviamente associato il file di configurazione di Struts (*struts-config.xml*), nel quale sono stati configurati gli elementi dell'applicazione, che sono:

- il datasource, utilizzato per la connessione alla base dati;
- i formbeans di tipo dinamico, utilizzati per il recupero dei dati dai form;
- le eccezzioni globali, per il recupero da situazioni di errore;
- i forward globali, per l'accesso a risorse globali;
- il mapping delle azioni;
- la configurazione dei resource bundles;
- la configurazione dei plug-in utilizzati.

I file di configurazione sono stati realizzati sfruttando la modalità grafica ad albero offerta dai plug-in di Exadel Studio per l'ambiente Eclipse. Inoltre per quanto riguardo il file *struts-config.xml* è stato utilizzata un ulteriore modalità grafica a diagramma, offerta sempre da Exadel Studio, che ha permesso di stabilire le relazioni che ci sono tra le pagine, le action ed i forward.

Essendo il file di configurazione di Struts di grandi dimensioni è stato suddiviso concettualmente in quattro aree, rappresentanti le funzionalità accessibili dai quattro tipi di utilizzatori del sistema. Di seguito viene presentato, in modalità grafica a diagramma, il file *struts-config.xm*l mentre il codice relativo ai file di configurazione e alla creazione della base dati è presente in appendice.

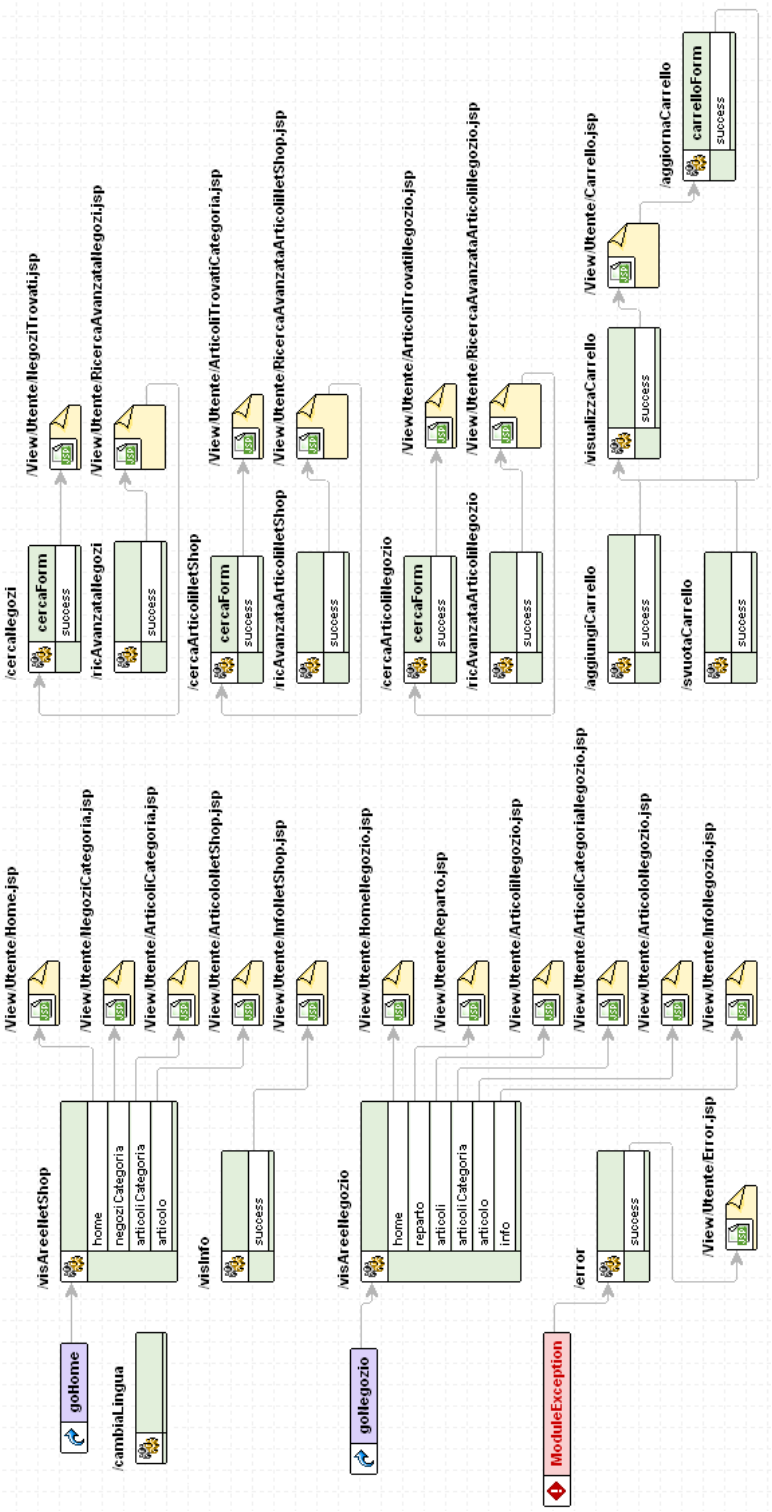

Figura 4.1: struts-config.xml Area Utente (1a parte)

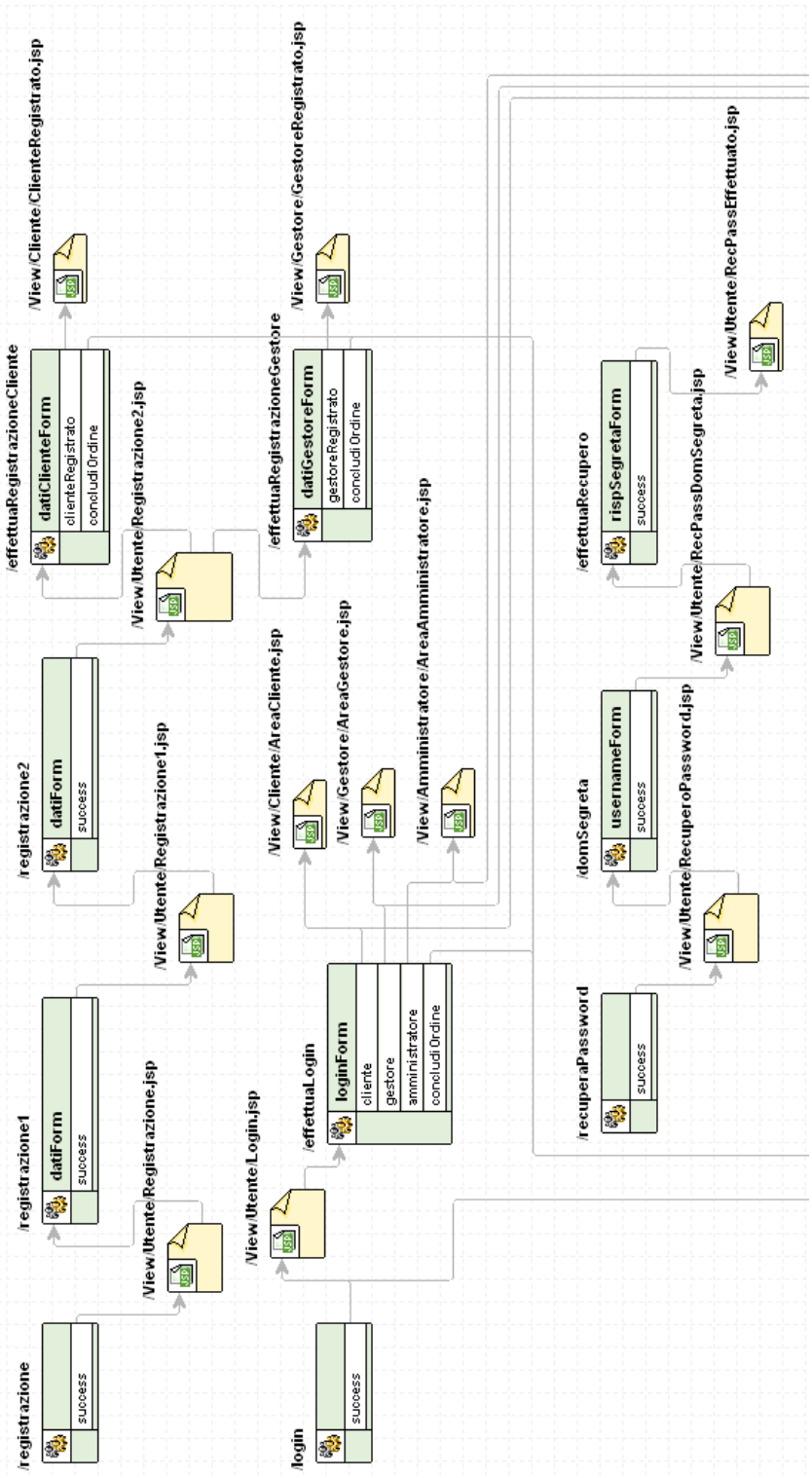

Figura 4.2: struts-config.xml Area Utente (2a parte)

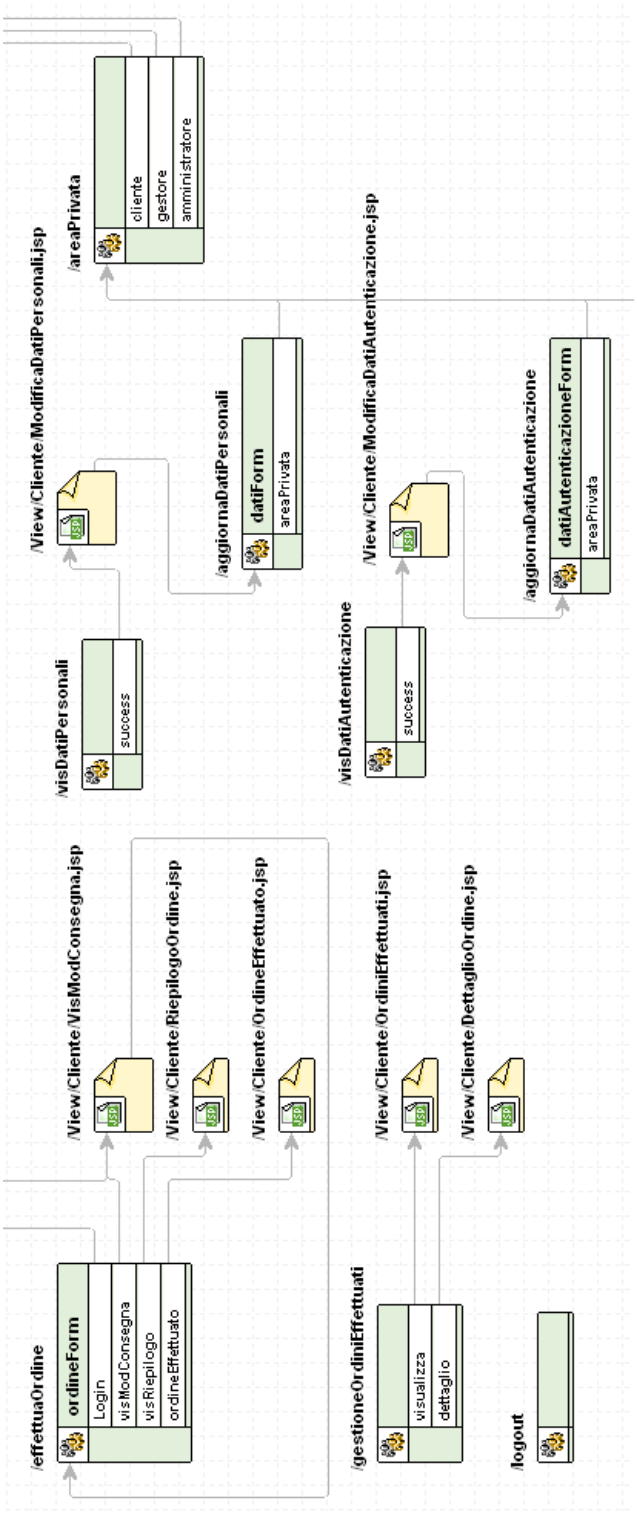

Figura 4.3: struts-config.xml Area Cliente

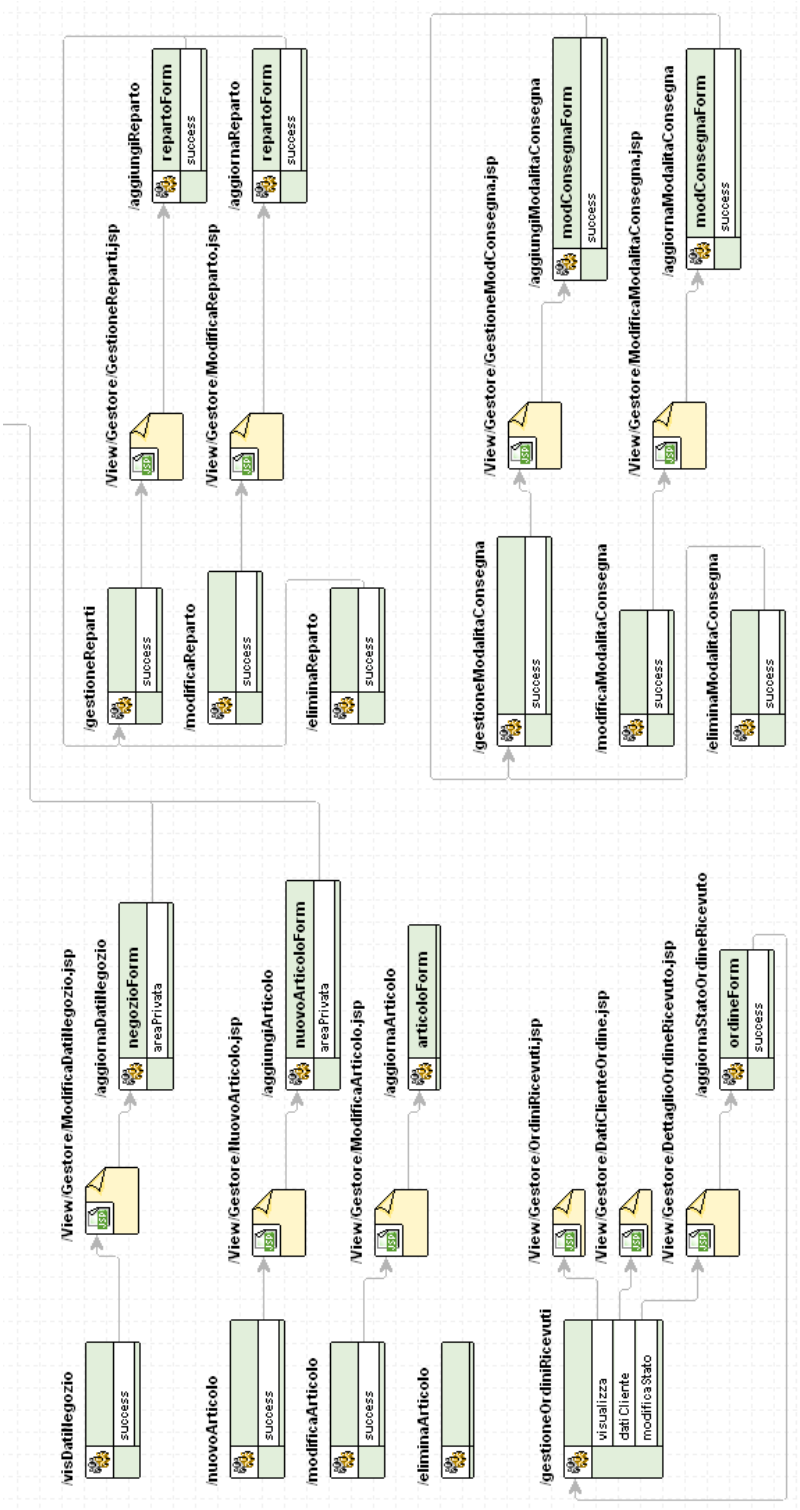

Figura 4.4: struts-config.xml Area Gestore

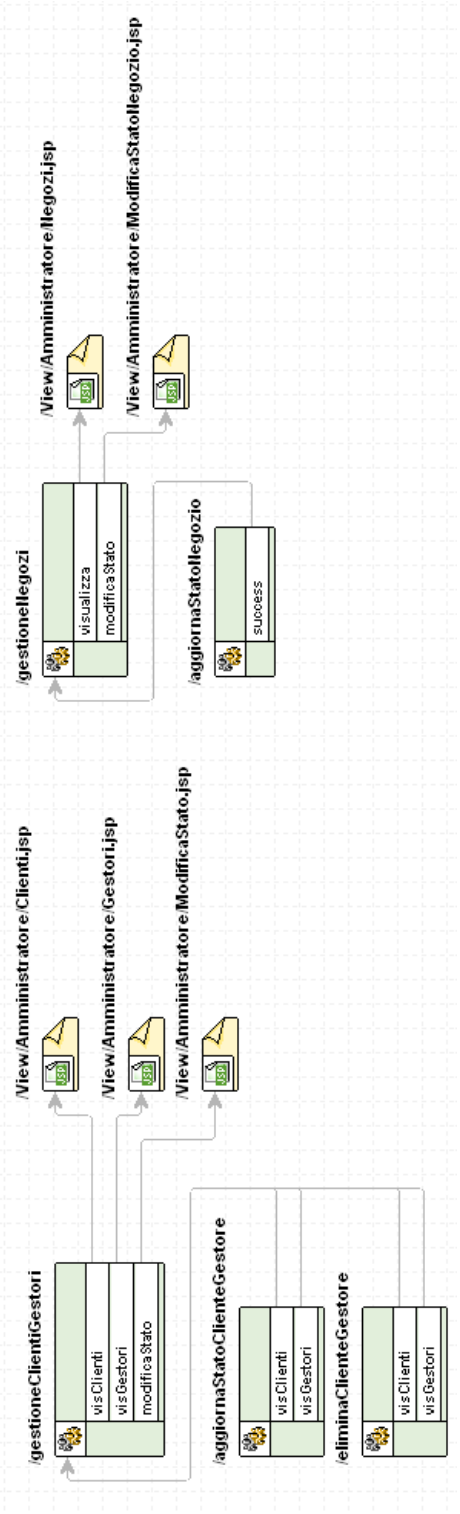

Figura 4.5: struts-config.xml Area Amministratore

4.3 Datasource

Per l'applicazione sviluppata è stato definito, nel file di configurazione di Struts, un datasource per la connessione al database MySQL. Il codice XML riguardante il datasource risulta essere il seguente:

```
<data-source key="dataSource"
  type="org.apache.struts.util.GenericDataSource">
  <set-property property="maxWait" value="5000"/>
  <set-property property="description"
    value="MySql Data Source"/>
  <set-property property="minCount" value="2"/>
  <set-property property="driverClass"
    value="com.mysql.jdbc.Driver"/>
  <set-property property="user" value="root"/>
  <set-property property="autoCommit" value="true"/>
  <set-property property="password" value="123456"/>
  <set-property property="url"
    value="jdbc:mysql://localhost:3306/NetShop?autoReconnect=true"
      />
  <set-property property="maxCount" value="10"/>
  <set-property property="maxActive" value="20"/>
</data-source>
```

4.4 Model

Il Model può essere considerato il cuore dell'applicazione realizzata, in quanto è costituito da tutte quelle classi che definiscono la logica dell'applicazione, ovvero la *business logic*.

Avendo utilizzato Struts per l'implementazione, si è resa necessaria la realizzazione delle *Action*, per cui le classi della *business logic* risultano più "leggere" e quindi la logica che sta alla base delle funzionalità è distribuita fra gli oggetti della *business logic* e le *Action*. Le ultime prevedono tutto ciò che riguarda l'accesso agli elementi del framework, mentre i primi eseguono le operazioni previste dall'applicazione.

Oltre alle classi della *business logic*, sono state implementate anche una serie di classi che gestiscono le operazioni di accesso alla base di dati, andando così a definire un ulteriore sottolivello detto appunto *data access*.

4.4.1 Business Logic

Le classi che appartengono alla *business logic* sono quelle che fanno parte del class diagram descritto in fase di progettazione, la solo differenza sta nel fatto che per alcune di esse sono stati aggiunti o modificati attributi e metodi legati all'implementazione. La maggior parte delle classi contiene metodi che utilizzano le classi del sottolivello *data access* per poter effettuare operazioni sulla base dati, tali metodi in generale risultano essere i seguenti:

- *carica()*: carica i dati dell'oggetto al quale viene applicato; chiama il metodo *ClasseDB.leggi(int id)* che ritorna un *ResultSet* dal quale vengono recuperati i dati dell'oggetto; il codice di tale metodo è in generale il seguente:

```
public void carica(){

  try{
    // Carica i dati dell'oggetto
    ResultSet rs=ClasseDB.leggi(id);

    rs.next();

    proprietà1=rs.getString("campo1");
    proprietà2=rs.getInt("campo2");
    ...
  }catch(Exception e){
     System.out.println("Errore " + e.getMessage());}
}
```

- *aggiorna()*: aggiorna nella base dati i dati dell'oggetto al quale viene applicato; chiama il metodo *ClasseDB.aggiorna(Oggetto ogg)* che riceve come parametro l'oggetto stesso;

- *inserisci()*: inserisce nella base dati il record corrispondente all'oggetto al quale viene applicato; chiama il metodo *ClasseDB.inserisci(Oggetto ogg)* che riceve come parametro l'oggetto stesso;

- *elimina()*: elimina dalla base dati il record corrispondente all'oggetto al quale viene applicato; chiama il metodo *ClasseDB.elimina(int id)* che si occupa dell'eliminazione;

- *caricaElementi()*: carica dalla base dati i dati di altri elementi associati all'oggetto al quale viene applicato; chiama il metodo *ClasseDB. leggiElementi(int id)* che ritorna un *ResultSet* dal quale attraverso un'iterazione vengono recuperati i suoi dati e creati ed aggiunti i nuovi elementi; il codice di tale metodo è in generale il seguente:

```
public void caricaElementi(){

  try{
    // Carica gli elementi associati
    ResultSet rs=ClasseDB.leggiElementi(id);

    while(rs.next()){
      listaElementi.add(
        new Elemento(rs.getInt("campo1"),
          rs.getString("campo2"), ...));}

  }catch(Exception e){
   System.out.println("Errore " + e.getMessage());}
}
```

- *cercaElementi(parametri)*: cerca e carica dalla base dati i dati di altri elementi associati all'oggetto al quale viene applicato, in base ai parametri inseriti dall'utente; il codice di tale metodo è simile al metodo *caricaElementi()*, solo che in questo caso vengono caricati gli elementi trovati a partire dai parametri inseriti dall'utente.

4.4.2 Data Access

Nella realizzazione di un sistema software che debba funzionare in locale oppure in rete, ci sono tipicamente degli oggetti per i quali va garantita la persistenza attraverso l'utilizzo della

basi di dati.

Per poter gestire le operazioni di lettura e scrittura nel database, sono previste due possibili soluzioni:

- ogni oggetto gestisce la propria persistenza in maniera autonoma, prevedendo dei metodi che accedono alla base di dati;

- ogni oggetto delegata la gestione delle propria persistenza ad un'altra classe che mette a disposizione una serie di metodi di accesso al database.

Con lo scopo di disaccoppiare le funzionalità della *business logic* dalle operazioni di accesso alla base di dati, generalmente si preferisce adottare la seconda soluzione. Ciò è stato fatto anche nella Web application realizzata, implementando il sottolivello *data access* del Model attraverso un'ulteriore insieme di classi, una per ciascuna corrispondente classe della *business logic* che necesitasse dell'accesso alla base dati. Inoltre è stata creata una classe *BaseDB*, che mantiene la connessione al datasource e viene utilizzata da tutte le altre classi per accedere al datasource, tale classe è descritta nel paragrafo successivo.

Le classi del *data access* contengono i metodi necessari per l'accesso ad una base dati, tali metodi in generale risultano essere:

- *leggi(int idEle)*: legge dalla base dati, in particolare da una tabella o dalla join di più tabelle, il record corrispondente ad idEle;
- *aggiorna(Oggetto ogg)*: aggiorna nella base dati i dati relativi all'oggetto ogg;
- *inserisci(Oggetto ogg)*: inserisce nella base un nuovo record con i dati di ogg;
- *elimina(int idEle)*: elimina dalla base dati il record corrispondente ad idEle.

Nelle classi del *data access* per l'accesso vero e proprio alla base di dati, sono state utilizzate le seguenti classi e interfacce *JDBC (Java DataBase Connectivity)*:

- *Connection*: rappresenta una connessione alla base di dati;
- *Statement*: definisce una generica query da eseguire sul database;

- *PreparateStatement*: permette di preparare una query, dando la possibiltà di inserire valori particolari (es: stringhe con apici e virgolette, valori binari, etc...)
- *ResultSet:* rappresenta un gruppo di record, risultato di una query eseguita sulla base di dati;
- *SQLException*: definisce un'eccezione generica che è sollevata nel caso in cui si verifichi un errore di accesso al database.

Classe BaseDB

Mantiene la connessione al datasource dell'applicazione. Al lancio dell'applicazione viene creata la connesione e mantenuta in questa classe. Tutte le altre classi del *data access* utilizzano questa classe per poter recuperare la connessione e poter effettuare le operazioni sulla base dati.

Possiede i seguenti attributi static:

- *conn*: connessione al datasource;
- *stm*: statement sul quale può essere eseguita una query.

Possiede i seguenti metodi static:

- *creaConn(DataSource ds)*: crea la connessione al datasource e lo statement;
- *getConn()*: restituisce la connessione;
- *getStm()*: restituisce lo statement.

4.5 Controller

4.5.1 Generale

Le action che fanno parte del controller sono state organizzate in packages in base ai ruoli che possono assumere gli utilizzatori di NetShop. Inoltre è stato definito un package *base* che contiene le action per le funzionalità comuni a più action. In generale ogni action crea degli oggetti, ne carica i dati e ne esegue delle funzionalità. Gli oggetti della *business logic*, che contengono i dati che dovranno essere visualizzati, vengono passati alle viste attaverso la *request* o la *sessione*.

Tutti i packages e le relative action sono descritti di seguito.

4.5.2 Package base

Contiene due action di base per l'applicazione, che permettono di visualizzare le categorie nelle aree di NetShop e i dati del negozio e i suoi reparti nelle aree di un negozio. Le action presenti in questo package sono:

- *BaseAction*: estende la classe *Action*; viene utilizzata sia come action per la visualizzazione di semplici viste che come supporto, attraverso la sua estensione, per le action degli altri package fornendo i suoi attributi e i suoi metodi;
- *BaseDispatchAction*: estende la classe *DispatchAction*; viene estesa dalle action degli altri package fornendo i suoi attributi e i suoi metodi come supporto.

Entrambe le classi possiedono i seguenti attributi *protected*:

- *nShop*: rappresenta l'oggetto di classe *NetShop* sul quale vengono eseguite le funzionalità generali del centro commerciale;
- *neg*: rappresenta l'oggetto di classe *Negozio* sul quale vengono eseguite le funzionalità generali di un negozio.

Inoltre entrambe le classi possiedono i seguenti metodi:

- *creaNetShop()*: crea l'oggetto *nShop* e ne carica le categorie;
- *creaNegozio(int idNeg)*: crea l'oggetto *neg*, ne carica i dati e i reparti.
- *check()*: si occupa del controllo della sicurezza (viene descritto nel paragrafo riguardante la sicurezza).

La action *BaseAction*, essendo una comune action, possiede il metodo *execute()* che le consente di poter essere utilizzata come action per la visualizzazione di semplici viste di NetShop o di un negozio; il metodo *execute()* invoca il metodo *creaNetShop()* e il metodo *creaNegozio(int idNeg)* ed infine scrive nella *request* gli oggetti *nShop* e *neg*.

La action *BaseDispatchAction* presenta invece il metodo *creaConn()* che viene utilizzato al lancio dell'applicazione per stabilire la connessione al datasource all'interno della classe *BaseDB*, il suo codice risulta essere il seguente:

```
private void creaConn(){

  if(BaseDB.getConn()==null){
    ServletContext context=servlet.getServletContext();
    DataSource dataSource=
      (DataSource)context.getAttribute("dataSource");
    dataaccess.model.BaseDB.creaConn(dataSource);}
}
```

Infine tutte le action che estendono queste due action si occupano della creazione di NetShop, della creazione del negozio o di entrambi invocando i metodi *creaNetShop()* e *creaNegozio(int idNeg)* ed infine scrivono nella *request* gli oggetti *nShop* e *neg*.

4.5.3 Package utente

Contiene le action per eseguire le funzionalità dell'*Utente*. Le classi presenti in questo package sono:

- *VisAreeNetShopAction*: permette di visualizzare le aree di consultazione di NetShop;
- *VisAreeNegozioAction*: permette di visualizzare le aree di consultazione di un negozio;
- *CercaAction*: consente di poter cerca i negozi e gli articoli;
- *GestioneCarrelloAction*: consente di poter gestire il carrello virtuale;
- *RegistrazioneAction*: permette di poter effettuare la registrazione;
- *LoginAction*: permette di poter effettuare il login;
- *RecuperoPasswordAction*: permette di poter effettuare il recupero della password di un cliente.

Classe VisAreeNetShopAction

Estende la classe *BaseDispatchAction* e contiene i metodi necessari per la visualizzazione delle aree di consultazione di NetShop, essi sono:

- *visHome()*: permette di visualizzare la home di NetShop;
- *visNegoziCategoria()*: permette di visualizzare i negozi con articoli in una determinata categoria;
- *visArticoliCategoria()*: permette di visualizzare gli articoli presenti in una categoria;
- *visArticolo()*: permette di visualizzare il dettaglio di un articolo all'interno di NetShop.

Classe VisAreeNegozioAction

Estende la classe *BaseDispatchAction* e contiene i metodi necessari per la visualizzazione delle aree di consultazione di un negozio, essi sono:

- *visHome()*: permette di visualizzare la home del negozio;
- *visReparto()*: permette gli articoli in un determinato reparto;
- *visArticoli()*: permette di visualizzare tutti gli articoli del negozio;
- *visArticoliCategoria()*: permette di visualizzare gli articoli del negozio presenti in una categoria;
- *visArticolo()*: permette di visualizzare il dettaglio di un articolo;
- *visInfo()*: permette di visualizzare le informazioni del negozio.

Classe CercaAction

Estende la classe *BaseDispatchAction* e contiene i metodi per la ricerca dei negozi e degli articoli, essi sono:

- *cercaNegozi()*: cerca i negozi e ne permette la visualizzazione;
- *cercaArticoliNetShop()*: cerca gli articoli all'interno di NetShop e ne permette la visualizzazione;
- *cercaArticoliNegozio()*: cerca gli articoli all'interno di un negozio e ne permette la visualizzazione.

Classe GestioneCarelloAction

Estende la classe *BaseDispatchAction* e contiene i metodi per la gestione del carrello, essi sono:

- *visualizza()*: permette di visualizzare tutti gli articoli che sono presenti nel carrello;

- *aggiungi()*: aggiunge un nuovo articolo nel carrello;
- *aggiorna()*: aggiorna le linee del carrello in base ai parametri passati nel form;
- *svuota()*: svuota il carrello.

Classe RegistrazioneAction

Estende la classe *BaseDispatchAction* e contiene i metodi per effettuare la registrazione, essi sono:

- *registraCliente()*: registra l'utente come cliente in base ai dati passati nei form della registrazione;
- *registraGestore()*: registra l'utente come gestore in base ai dati passati nei form della registrazione.

Classe LoginAction

Estende la classe *Action* e contiene il solo metodo *execute()* che permette di poter effettuare il login.

Classe RecuperoPasswordAction

Estende la classe *BaseDispatchAction* e contiene i metodi per poter effettuare il recupero della password di un cliente, essi sono:

- *recuperaDomandaSegreta()*: recupera la domanda segreta da fare all'utente;
- *recuperaPassword()*: recupera la password del cliente e la invia al suo indirizzo email.

4.5.4 Package cliente

Contiene le action per eseguire le funzionalità del *Cliente*. Le classi presenti in questo package sono:

- *EffettuaOrdineAction()*: permette di poter effettuare un ordine;
- *AreaPrivataAction()*: permette di poter visualizzare l'area privata;
- *AggiornaDatiAction()*: consente di poter gestire i dati del cliente;
- *GestioneOrdiniEffettuatiAction()*: consente di poter gestire gli ordini effettuati;
- *LogoutAction()*: permette di poter effettuare il loguot.

Classe EffettuaOrdineAction

Estende la classe *BaseDispatchAction* e contiene i metodi per poter effettuare un ordine degli articoli nel carrello verso un negozio, essi sono:

- *visModConsegna()*: permette di visualizzare le modalità di spedizione e di pagamento disponibili nel negozio verso cui si sta effettuando l'ordine;
- *visRiepilogo()*: permette di visualizzare il riepilogo dell'ordine che si sta effettuando;
- *effettua()*: effettua l'ordine ed invia un email al cliente che ha effettuato l'ordine e al gestore del negozio.

Classe AreaPrivataAction

Estende la classe *BaseAction* e contiene il solo metodo *execute()* che consente di poter visualizzare l'area privata in funzione del tipo del cliente.

Classe AggiornaDatiAction

Estende la classe *BaseDispatchAction* e contiene i metodi per poter aggiornare i dati del cliente, essi sono:

- *aggiornaDatiPersonali()*: consente di poter aggiornare i dati personali del cliente;
- *aggiornaDatiAutenticazione()*: consente di poter aggiornare i dati di autenticazione del cliente.

Classe GestioneOrdiniEffettuatiAction

Estende la classe *BaseDispatchAction* e contiene i metodi per poter gestire gli ordini effettuati dal cliente, essi sono:

- *visualizza()*: permette di visualizzare tutti gli ordini effettuati dal cliente;
- *elimina()*: permette di poter eliminare un ordine effettuato non ancora in preparazione;
- *visDettaglio()*: permette di visualizzare il dettaglio di un ordine effettuato.

Classe LogoutAction

Estende la classe *Action* e contiene il solo metodo *execute()* che permette di poter effettuare il logout.

4.5.5 Package gestore

Contiene le action per eseguire le funzionalità del *Gestore*. Le classi presenti in questo package sono:

- *AggiornaDatiNegozioAction()*: permette di aggiornare i dati del negozio;

- *GestioneRepartiAction()*: permette di poter gestire i reparti del negozio;
- *GestioneArticoliAction()*: permette di poter gestire gli articoli del negozio;
- *GestioneOrdiniRicevutiAction()*: permette di poter gestire gli ordini ricevuti presso il negozio;
- *GestioneModConsegnaAction()*: permette di poter gestire le modalità di spedizione e di pagamento.

Classe AggiornaDatiNegozioAction

Estende la classe *BaseAction* e contiene il solo metodo *execute()* che permette di poter aggiornare i dati del negozio.

Classe GestioneRepartiAction

Estende la classe *BaseDispatchAction* e contiene i metodi per poter effettuare la gestione dei reparti, essi sono:

- *visualizza()*: permette di visualizzare tutti i reparti del negozio;
- *aggiungi()*: aggiunge un nuovo reparto;
- *modifica()*: permette di poter modificare il nome di un reparto;
- *aggiorna()*: aggiorna il nome di un reparto;
- *elimina()*: elimina un reparto.

Classe GestioneArticoliAction

Estende la classe *BaseDispatchAction* e contiene i metodi per poter effettuare la gestione degli articoli, essi sono:

- *nuovo()*: permette di poter inserire un nuovo articolo;

- *aggiungi()*: aggiunge un nuovo articolo;
- *modifica()*: permette di poter modificare i dati di un articolo;
- *aggiorna()*: aggiorna i dati di un articolo;
- *elimina()*: elimina un articolo.

Classe GestioneOrdiniRicevutiAction

Estende la classe *BaseDispatchAction* e contiene i metodi per poter effettuare la gestione degli ordini ricevuti, essi sono:

- *visualizza()*: permette di visualizzare tutti gli ordini ricevuti;
- *dettaglio()*: permette di visualizzare il dettaglio di un ordine ricevuto;
- *aggiornaStato()*: aggiorna lo stato di un ordine ricevuto;
- *visDatiCliente()*: permette di visualizzare i dati del cliente che ha effettuato un ordine.

Classe GestioneModConsegnaAction

Estende la classe *BaseDispatchAction* e contiene i metodi per poter effettuare la gestione delle modalità di spedizione e di pagamento, essi sono:

- *visualizza()*: permette di visualizzare le modalità di spedizione o di pagamento;
- *aggiungi()*: aggiunge una nuova modalità di spedizione o di pagamento;
- *modifica()*: permette di poter modificare i dati di una modalità di spedizione o di pagamento;
- *aggiorna()*: aggiorna i dati di una modalità di spedizione o di pagamento;
- *elimina()*: elimina una modalità di spedizione o di pagamento.

4.5.6 Package amministrazione

Contiene le action per eseguire le funzionalità del *Amministratore*. Le classi presenti in questo package sono:

- *AmministClientiGestoriAction*: permette di poter gestire tutti i clienti e i gestori;
- *AmministNegoziAction*: permette di poter gestire tutti i negozi.

Classe AmministClientiGestoriAction

Estende la classe *BaseDispatchAction* e contiene i metodi per poter effettuare la gestione dei clienti e dei gestori, essi sono:

- *visualizzaClienti()*: permette di visualizzare tutti i clienti;
- *visualizzaGestori()*: permette di visualizzare tutti i gestori;
- *modificaStato()*: permette di poter modificare lo stato di un cliente o di un gestore;
- *aggiornaStato()*: aggiorna lo stato di un cliente o di un gestore;
- *eliminaCliente()*: elimina un cliente;
- *eliminaGestore()*: elimina un gestore.

Classe AmministNegoziAction

Estende la classe *BaseDispatchAction* e contiene i metodi per poter effettuare la gestione dei negozi, essi sono:

- *visualizzaNegozi()*: permette di visualizzare tutti i negozi;
- *modificaStato()*: permette di poter modificare lo stato di un negozio;
- *aggiornaStato()*: aggiorna lo stato di un negozio.

4.6 View

Le viste risultano essere delle pagine JSP che sono state realizzate utilizzando sia i tag generici HTML e JSP che le librerie dei tag fornite da Struts. Le librerie dei tag fornite da Struts sono state utilizzate per l'accesso alle proprietà dei bean, per la costruzione dei form, per la definizione dei costrutti condizionali e per la visualizzazione dei messaggi generici e dei messaggi di errore che vengono prelevati dal Resource Bundle.

E' stato utilizzato il framework Tiles per la definizione del layout delle pagine, il framework Validator per la validazione dei dati inseriti dall'utente con la definizione di formbeans dinamici per l'aquisizione dei dati e infine si è fatto uso dell'Internazionalizzazione (I18N) per la visualizzazione delle pagine sia in italiano che in inglese.

4.6.1 Tiles

Il framework Tiles è stato utilizzato per poter dare alla Web application una struttura basata sul concetto di template, in modo che nel momento in cui si renda necessaria una modifica al contenuto delle pagine, questa debba essere eseguita una sola volta e non replicata.

Sono stati definiti due tipi di layout per le pagine:

- *layoutNetShop*, per le pagine di NetShop in generale;
- *layoutNegozio*, per le pagine relative ai negozi.

I layout utilizzati sono proposti nella figura seguente.

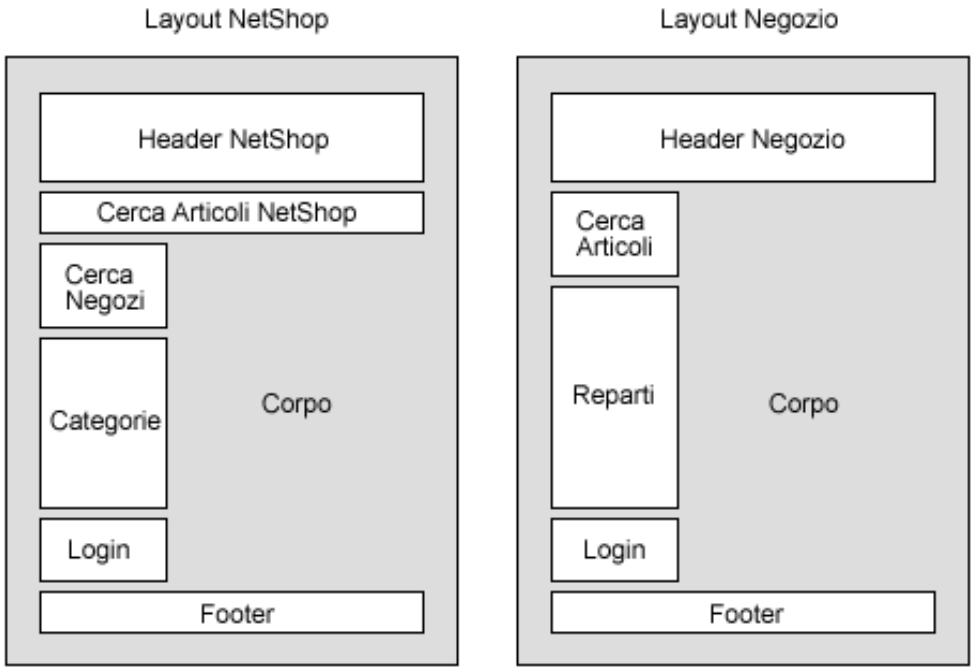

Figura 4.6: Layout NetShop, Layout Negozio

Le definizioni dei layout sono contenute nel file *tiles-defs.xml*, che è stato specificato nella registrazione del plug-in per Tiles all'interno del file *struts-confing.xml*:

```
<plug-in className="org.apache.struts.tiles.TilesPlugin">
  <set-property property="moduleAware" value="true"/>
  <set-property property="definitions-config"
    value="/WEB-INF/tiles-defs.xml"/>
</plug-in>
```

Il codice XML relativo alle definizioni dei layout è il seguente:

```
<tiles-definitions>
  <definition name="layoutNetShop"
    path="/View/Layout/LayoutNetShop.jsp">
  <put name="header"
    value="/View/Layout/HeaderNetShop.jsp"/>
  <put name="cercaArticoli"
    value="/View/Layout/CercaArticoliNetShop.jsp"/>
  <put name="cercaNegozi"
    value="/View/Layout/CercaNegozi.jsp"/>
  <put name="categorie"
    value="/View/Layout/Categorie.jsp"/>
```

```
  <put name="login"
    value="/View/Layout/Login.jsp"/>
  <put name="footer"
    value="/View/Layout/Footer.jsp"/>
</definition>

<definition name="layoutNegozio"
path="/View/Layout/LayoutNegozio.jsp">
<put name="header"
value="/View/Layout/HeaderNegozio.jsp"/>
<put name="cercaArticoli"
value="/View/Layout/CercaArticoliNegozio.jsp"/>
<put name="reparti"
value="/View/Layout/Reparti.jsp"/>
<put name="login"
value="/View/Layout/Login.jsp"/>
<put name="footer"
value="/View/Layout/Footer.jsp"/>
</definition>
</tiles-definitions>
```

I vari elementi appartenenti ai due layout sono stati posti in una cartella denominata appunto *Layout*.

4.6.2 Validator

Per la validazione dei dati immessi dall'utente è stato utilizzato il framework Validator, andando a utilizzare le regole di validazione fornite.

Per poter utilizzare il Validator è stato necessario registrare il plug-in per il Validator all'interno del file *struts-confing.xml*, specificando il percorso del file delle regole di validazione fornite (*validator-rules.xml*) e del file in cui sono dichiarati i form da validare (*validation.xml*):

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
  <set-property property="pathnames"
    value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"/>
</plug-in>
```

Per quanto riguarda le regole di validazione fornite dal Validator nel file *validator-rules.xml*, queste ultime non sono state assolutamente modificate e non ne sono state create delle ulteriori, in quanto il Validator mette a disposizione le funzionalità di validazione più comuni, tra cui anche il controllo

di correttezza di un indirizzo email. Le principali validation-rules utilizzate sono le seguenti:

- *required*: controlla che un campo non sia vuoto, in quanto è obbligatoria l'immissione di un valore;
- *email*: verifica che un valore abbia il formato corretto di un indirizzo email;
- *minlength*: valuta se il valore di un campo rispetti una lunghezza minima prefissata;
- *mask*: controlla che un certo valore sia conforme ad un pattern specificato;
- *intRange* e *floatRange*: valutano che un certo valore sia un int o un float e sia contenuto in un intervallo di valori.

All'interno del file *validation.xml* sono stati dichiarati tutti i form ed i relativi campi sui quali sono state applicate delle regole di validazione specifiche. Tali form corrispondono esattamente a i formbean registrati nel file di configurazione *struts-config.xml*, così come le proprietà di questi ultimi che coincidono con i campi soggetti a validazione.

4.6.3 Internazionalizzazione (I18N)

Attraverso il meccanismo dell'Internazionalizzazione (I18N), è possibile visualizzare il testo presente nella Web application in lingue diverse, in relazione alla localizzazione geografica da cui si collega l'utente oppure sulla base delle impostazioni della lingua del browser.

Nel caso dell'applicazione realizzata, si è previsto il supporto per la lingua italiana, considerata di default, e per la lingua inglese. In questo modo, senza la necessità di dover sviluppare due volte la medesima Web application, è possibile comunque garantire l'utilizzo di quest'ultima a persone di lingua differente.

Sono stati realizzati due Resource Bundle contenenti i messaggi da visualizzare nelle due lingue, essi sono:

- *NetShopMessageResources.properties*, per la lingua di default, ovvero quella italiana;
- *NetShopMessageResources_ en.properties*, per la lingua inglese.

Ciascuno di essi è costituito da una serie di coppie *key-message*, attraverso le quali è possibile visualizzare un certo messaggio semplicemente specificando la chiave ad esso associata.

Per poter utilizzare i Resource Bundle definiti è stata inserita nel file *struts-confing.xml* la seguente linea di codice XML:

```
<message-resources parameter="NetShopMessageResources"/>
```

4.6.4 View Utente

Rappresenta le viste alle quali può accedere l'*Utente*. Tali viste vengono presentate di seguito.

Home di NetShop

Rappresenta la vista iniziale di NetShop, nella quale possiamo distinguere, in base al layout che è stato definito con il framework Tiles, le seguenti aree:

- aree che fanno parte di tutte le viste di NetShop;
- area che caratterizza la vista stessa.

Le aree presenti nella vista iniziale e in tutte le altre viste di NetShop sono:

- *Header*, che comprende il logo di NetShop e il menu; dal menu è possibile cambiare la lingua, visualizzare gli articoli nel carrello, ritornare alla home, entrare nell'area della registrazione, andare all'area del login ed avere informazioni su NetShop;

- *Categorie degli articoli*, che permette di poter accedere ai negozi e gli articoli con una determinata categoria;
- *Cerca Articoli per nome e categoria*, che consente di poter cercare subito gli articoli desiderati;
- *Cerca Negozi per nome*, che consente di trovare subito dei negozi in base al loro nome;
- *Login*, che consente di effettuare subito il login, di procedere per il recupero della password o di entrare nell'area della registrazione;
- *Footer*, che mostra informazioni su chi ha sviluppato l'applicazione.

L'area che caratterizza la vista iniziale è quella dei negozi in rotazione. Essa visualizza il logo dei negozi posti in rotazione consentendo di poter accedere ai rispettivi negozi.

Figura 4.7: View Home di NetShop

Negozi con Articoli in una Categoria

Visualizza i dati dei negozi fornendone una breve descrizione. Essa permette di accedere ai negozi visualizzati oppure visualizzare tutti gli articoli con la categoria che è stata scelta. E' possibile accedere a questa vista a partire dalle categorie degli articoli.

Figura 4.8: View Negozi con Articoli in una Categoria

Articoli in una Categoria

Visualizza i dati di tutti gli articoli presenti in una determinata categoria fornendo per loro una breve descrizione. Essa permette di andare al dettaglio degli articoli direttamente da NetShop senza entrare nel loro negozio, di aggiungere gli articoli al carrello oppure di entrare nel loro negozio. E' possibile accedere a questa vista dopo aver visualizzato i negozi con articoli in una categoria oppure dal dettaglio di un articolo all'interno di NetShop.

Ricerca Avanzata Negozi

Permette di poter cercare i negozi inserendo il nome e la descrizione. E' possibile accedere a questa vista dall'area di ricerca dei negozi per nome presente in tutte le viste di NetShop.

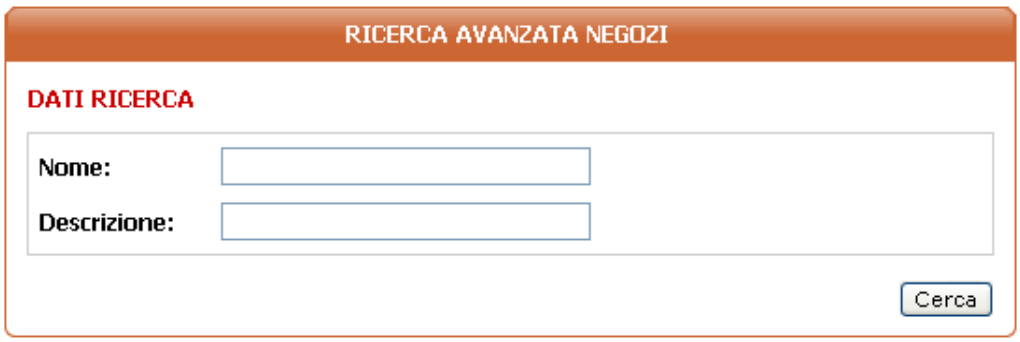

Figura 4.9: View Ricerca Avanzata Negozi

Negozi Trovati

Visualizza i dati dei negozi trovati. Questa vista è simile a quella dei che visualizza i negozi con articoli in una categoria. Da essa è possibile accedere ai negozi visualizzati.

Ricerca Avanzata Articoli in NetShop

Permette di poter cercare gli articoli in NetShop inserendo il nome, la descrizione, la categoria, il prezzo minimo e il prezzo massimo. E' possibile accedere a questa vista dall'area di ricerca degli articoli per nome e categoria presente in tutte le viste di NetShop.

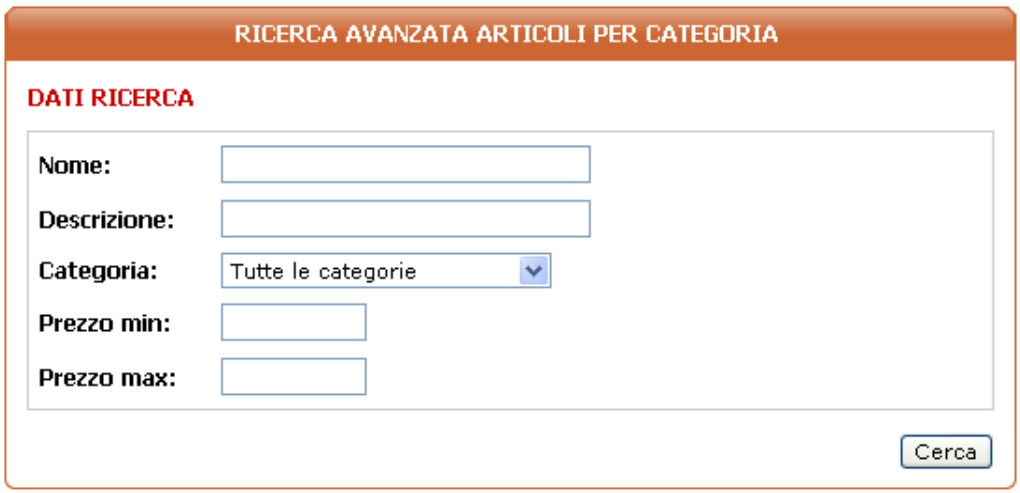

Figura 4.10: View Ricerca Avanzata Articoli in NetShop

Articoli Trovati in NetShop

Visualizza i dati degli articoli trovati in NetShop. Questa vista è simile a quella che visualizza gli articoli in una categoria. Essa permette di andare al dettaglio degli articoli direttamente da NetShop senza entrare nel loro negozio, di aggiungere gli articoli al carrello oppure di entrare nel loro negozio.

Informazioni di NetShop

Visualizza delle informazioni generali su NetShop. Essa permette di poter inviare un email all'amministratore di NetShop attraverso un link al suo indirizzo email. E' possibile accedere a questa vista dal menù di NetShop.

Home del Negozio

Rappresenta la vista iniziale di un negozio, nella quale possiamo distinguere, in base al layout che è stato definito con il framework Tiles, le seguenti aree:

- aree che fanno parte di tutte le viste del negozio;
- area che caratterizza la vista stessa.

Le aree presenti nella vista iniziale e in tutte le altre viste del negozio sono:

- *Header*, che comprende il logo e il nome del negozio, la sua descrizione e il menu; dal menu è possibile cambiare la lingua, visualizzare gli articoli nel carrello, andare alla home di NetShop, ritornare alla home del negozio, entrare nell'area della registrazione, andare all'area del login ed avere informazioni sul negozio;
- *Reparti*, che permette di poter accedere agli articoli presenti nei vari reparti;
- *Link tutti gli articoli*, che permette di poter visualizzare tutti gli articoli del negozio;

- *Cerca Articoli per nome*, che consente di poter cercare subito gli articoli desiderati all'interno del negozio;
- *Login*, che consente di effettuare subito il login, di procedere per il recupero della password o di entrare nell'area della registrazione;
- *Footer*, che mostra informazioni su chi ha sviluppato l'applicazione.

L'area che caratterizza la vista iniziale è quella degli articoli in vetrina. Essa visualizza il nome, l'immagine e il prezzo degli articoli in vetrina consentendo di poterli porre nel carrello o di accedere al loro dettaglio.

Figura 4.11: View Home del Negozio

Articoli nel Reparto

Visualizza i dati degli articoli in un reparto fornendo per loro una breve descrizione. Essa permette di porre gli articoli nel carrello o di entrare nella visualizzazione del loro dettaglio. E' possibile accedere a questa vista a partire dai reparti o dal dettaglio di un articolo.

Figura 4.12: View Articoli nel Reparto

Articoli del Negozio

Visualizza i dati di tutti gli articoli del negozio fornendo per loro una breve descrizione. Questa vista è simile a quella che visualizza gli articoli in un reparto, solo che in questo caso vengono visualizzati tutti gli articoli. E' possibile accedere a questa vista attraverso l'apposito link posto sotto ai reparti.

Articoli del Negozio in una Categoria

Visualizza i dati degli articoli del negozio in una determinata categoria fornendo per loro una breve descrizione. Questa vista è simile a quella che visualizza gli articoli del negozio, solo che in questo caso vengono visualizzati gli articoli appartenenti ad un unica categoria. E' possibile accedere a questa vista a partire dal dettaglio di un articolo.

Dettaglio di un Articolo

Visualizza i dati di un articolo fornendo una descrizione completa. Essa permette di aggiungere l'articolo al carrello, di andare al reparto dell'articolo o di accedere agli articoli del negozio nella categoria dell'articolo. E' possibile accedere a questa vista dall'interno del negozio.

Figura 4.13: View Dettaglio di un Articolo

Dettaglio di un Articolo da NetShop

Visualizza i dati di un articolo dall'interno di NetShop senza entrare direttamente nel suo negozio. Questa vista è simile a quella che visualizza il dettaglio di un articolo nel negozio. Essa permette di aggiungere l'articolo al carrello, di accedere al negozio a cui appartiene l'articolo o di accedere a tutti gli articoli nella categoria dell'articolo. E' possibile accedere a questa vista dall'interno di NetShop.

Informazioni del Negozio

Visualizza i dati del negozio, del gestore e delle modalità di spedizione e pagamento del negozio. Essa permette di poter inviare un email al gestore del negozio attraverso un link al suo indirizzo email. E' possibile accedere a questa vista dal menù del negozio.

Ricerca Avanzata Articoli nel Negozio

Permette di poter cercare gli articoli nel negozio inserendo il nome, la descrizione, la categoria, il reparto, il prezzo minimo e il prezzo massimo. Questa vista è simile a quella della Ricerca Avanza Articoli in NetShop, ma in questo caso è possibile inserire anche il reparto degli articoli. E' possibile accedere a questa vista dall'area di ricerca degli articoli per nome presente in tutte le viste del negozio.

Articoli Trovati nel Negozio

Visualizza i dati degli articoli trovati nel negozio fornendo per loro una breve descrizione. Questa vista è simile a quelle che visualizza gli articoli in un reparto. Essa permette di porre gli articoli nel carrello o di entrare nella visualizzazione del loro dettaglio.

Carrello

Visualizza i dati degli articoli presenti nel carrello, organizzandoli in base al negozio di appartenenza. Essa permette di poter aggiornare il carrello segnando gli articoli da rimuovere e modificando le quantità degli articoli ed inoltre permette di poter svuotare il carrello; permette ancora di poter effettuare un ordine presso un determinato negozio, di tornare all'ultimo negozio che si è visitato, di tornare ai negozi a cui appartengono gli articoli presenti e di andare al dettaglio degli articoli. E' possibile accedere a questa vista dal menù di NetShop o dal menù del negozio, inoltre si accede a questa vista ogni volta che viene inserito un nuovo articolo nel carrello.

Figura 4.14: View Carrello

Registrazione come Cliente

Permette di poter registrare il cliente attraverso due viste. Nella prima vista vengono inseriti i dati personali, suddivisi in: dati anagrafici, dati di spedizione e dati di contatto. Nella seconda vista vengono inseriti i dati di autenticazione. E' possibile accedere a queste viste dopo la richiesta di registrazione scegliendo la registrazione come cliente.

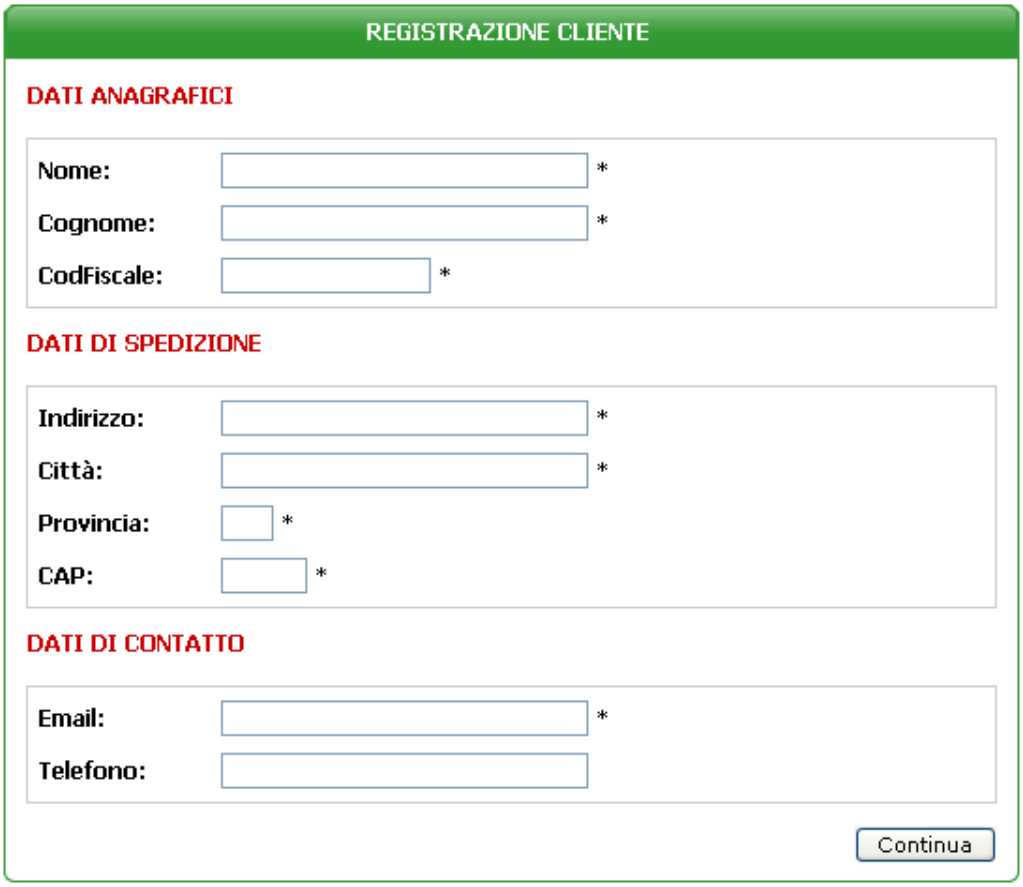

Figura 4.15: View Inserimenro dei Dati Personali

Figura 4.16: View Inserimenro dei Dati di Autenticazione

Registrazione come Gestore

Permette di poter registrare il gestore e di poter creare il suo negozio attraverso due viste. Nella prima vista vengono inseriti i dati personali, come nel caso del cliente. Nella seconda vista vengono inseriti i dati di autenticazione, i dati del negozio e i dati delle modalità di consegna iniziali. E' possibile accedere a queste viste dopo la richiesta di registrazione scegliendo la registrazione come gestore.

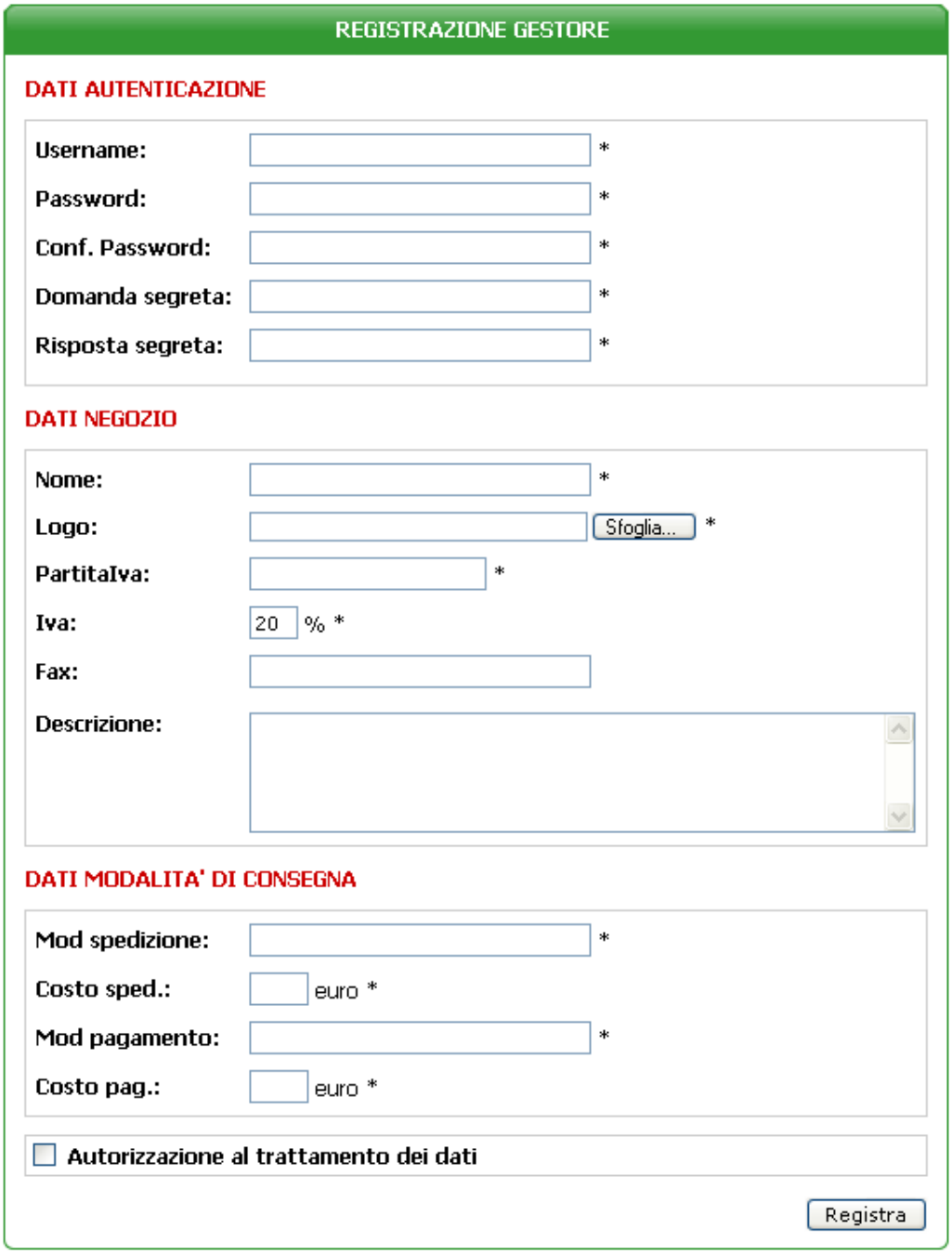

Figura 4.17: View Inserimenro dei Dati di Autenticazione e del Dati del Negozio

Login

Permette di poter effettuare il login inserendo username e password. Essa permette inoltre di poter effettuare il recupero della password e di poter accedere all'area di registrazione. E' possibile accedere a questa vista dal menù di NetShop o dal menù del negozio.

Figura 4.18: View Login

Recupero Password

Permette di poter effettuare il recupero della password attraverso due viste. Nella prima vista viene inserita l'username. Nella seconda vista viene inserita la risposta alla domanda segreta. E' possibile accedere al recupero della password dall'area del login.

Figura 4.19: View Inserimento della tua Username

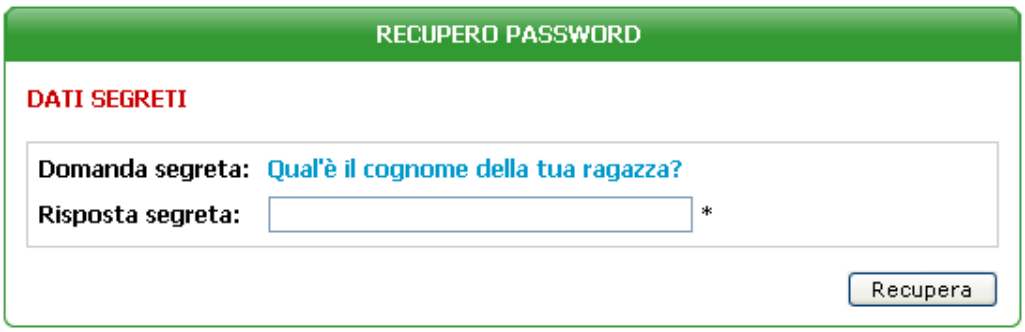

Figura 4.20: View Risposta alla Domanda Segreta

4.6.5 View Cliente

Rappresenta le viste alle quali può accedere il *Cliente*. Tali viste vengono presentate di seguito.

Effettuazione Ordine

Permette di poter effettuare un ordine degli articoli nel carrello, per un determinato negozio, attraverso due viste. La prima vista consente di poter scegliere la modalità di spedizione e la modalità di pagamento tra quelle fornite dal negozio. La seconda vista visualizza il riepilogo dell'ordine, il cliente può scegliere o meno di concludere l'ordine. Dopo che l'ordine è stata concluso viene inviata un'email al cliente e al gestore del negozio. E' possibile accedere all'effettuazione dell'ordine a partire dal carrello.

Figura 4.21: View Scelta della Modalità di Spedizione e di Pagamento

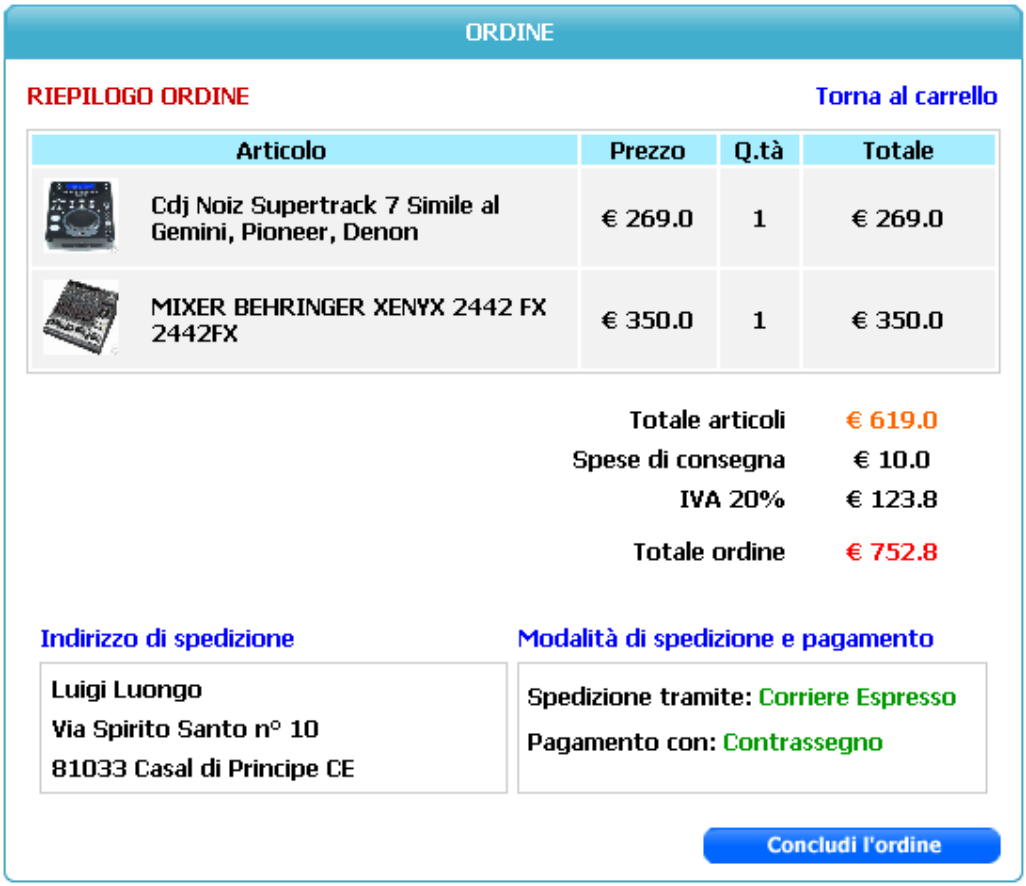

Figura 4.22: View Riepilogo dell'Ordine

Conferma d'ordine num. 28 su NETSHOP

luigiluong@tele2.it

A: luigiluong@tiscali.it

```
Grazie per aver acquistato su NETSHOP presso il negozio MY CLUB

Numero ordine: 28

Data ordine: 06-09-2007

Dettaglio dell'ordine:

ARTICOLO: Cdj Noiz Supertrack 7 Simile al Gemini, Pioneer, Denon
PREZZO: 269.0 euro
QUANTITA': 1
TOTALE: 269.0 euro

ARTICOLO: MIXER BEHRINGER XENYX 2442 FX 2442FX
PREZZO: 350.0 euro
QUANTITA': 1
TOTALE: 350.0 euro

Totale articoli: 619.0 euro
Spese di consegna: 10.0 euro
Iva 20%: 123.8 euro
---------------------------------
Totale ordine: 752.8 euro

Indirizzo di spedizione:
Luigi Luongo
Via Spirito Santo n° 10
81033 Casal di Principe CE

Modalità di spedizione: Corriere Espresso
Modalità di pagamento: Contrassegno
```

Figura 4.23: View Email Ordine Effettuato

Area Cliente

Rappresenta la vista privata del cliente. Da essa è possibile accedere alle funzionalità previste per il cliente. Per quanto riguarda la gestione dei dati, è possibile accedere alla visualizzazione e la modifica dei dati personali e alla visualizzazione e la modifica dei dati di autenticazione, mentre per quanto riguarda la gestione degli ordini è possibile accedere alla visualizzazione degli

ordini effettuati. Infine è possibile ritornare al negozio che si stava visitando. E' possibile accedere a questa vista dal menù di NetShop, dal menù del negozio o dall'area di benvenuto, disponibile dopo il login in tutte le viste al posto dell'area di login .

Figura 4.24: View Area Cliente

Visualizza/Modifica dati personali

Visualizza i dati personali del cliente dando la possibilità di poterli modificare ed aggiornare. E' possibile accedere a queste vista dall'area del cliente.

Visualizza/Modifica dati autenticazione

Visualizza i dati di autenticazione del cliente dando la possibilità di poterli modificare ed aggiornare. E' possibile accedere a queste vista dall'area del cliente.

Ordini Effettuati

Visualizza il numero, la data, lo stato e il prezzo degli ordini effettuati. Essa permette di poter eliminare un ordine non ancora in preparazione e di visualizzare il dettaglio di un ordine. Gli ordini vengono organizzati in base

al nome negozio verso il quale sono stati effettuati ed è possibile riaccedervi attraverso un link. E' possibile accedere a questa vista dall'area del cliente.

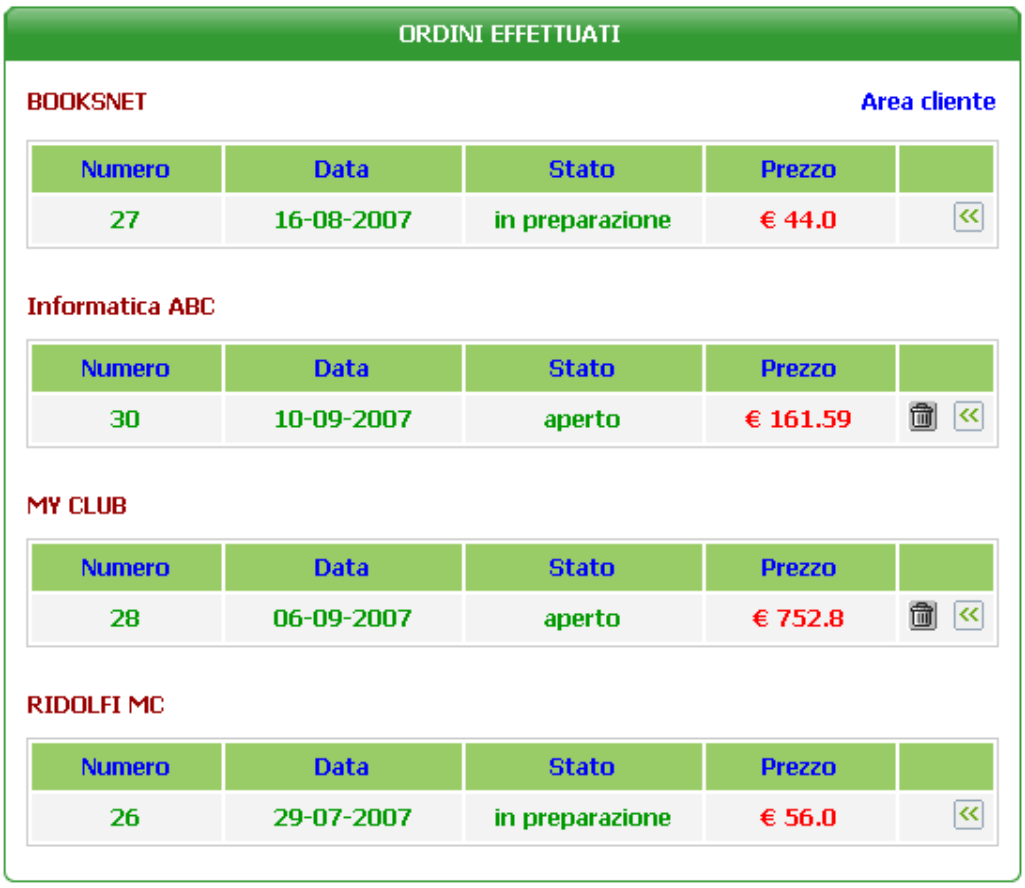

Figura 4.25: View Ordini Effettuati

Dettaglio Ordine Effettuato

Visualizza i dati di un ordine effettuato. E' possibile accedere a questa vista dagli ordini effettuati.

Figura 4.26: View Dettaglio Ordine Effettuato

4.6.6 View Gestore

Rappresenta le viste alle quali può accedere il *Gestore*. Tali viste vengono presentate di seguito.

Area Gestore

Rappresenta la vista privata del gestore. Da essa è possibile accedere alle funzionalità previste per il gestore. Per quanto riguarda la gestione dei dati, oltre alle funzionalità ereditate dal cliente, è possibile poter accedere alla visualizzazione e la modifica dei dati del negozio. Per la gestione del catalogo degli articoli è possibile accedere alla gestione dei reparti e all'inserimento di un nuovo articolo. E' da notare che la modifica e l'eliminazione di un articolo

vengono effettuate direttamente all'interno del negozio gestito, tali operazioni vanno a sostituire l'operazione di inserimento dell'articolo nel carrello non disponibile per il negozio gestito. Per la gestione degli ordini, oltre alle funzionalità ereditate dal cliente, è possibile accedere alla visualizzazione e alla modifica dello stato degli ordini ricevuti. Per le modalità di consegna è possibile accedere alla gestione delle modalità di spedizione e alla gestione delle modalità di pagamento. Infine è possibile andare al negozio gestito o ritornare al negozio che si stava visitando. E' possibile accedere a questa vista dal menù di NetShop, dal menù del negozio o dall'area di benvenuto, disponibile dopo il login in tutte le viste al posto dell'area di login.

Figura 4.27: View Area Gestore

Visualizza/Modifica Dati Negozio

Visualizza i dati del negozio gestito dando la possibilità di poterli modificare ed aggiornare. E' possibile accedere a questa vista dall'area del gestore.

Figura 4.28: View Visualizza/Modifica Dati Negozio

Gestione Reparti

Visualizza il nome dei reparti e il numero degli articoli presenti in ogni reparto. Essa permette di poter accedere alla modifica del nome di un reparto o di eliminare un reparto che non contiene nessun articolo. Inoltre è possibile inserire un nuovo reparto. E' possibile accedere a questa vista dall'area del gestore.

Figura 4.29: View Gestione Reparti

Modifica Reparto

Permette di poter modificare il nome di un reparto. E' possibile accedere a questa vista dalla gestione dei reparti.

Nuovo Articolo

Permette di poter inserire un nuovo articolo. E' possibile accedere a questa vista dall'area del gestore.

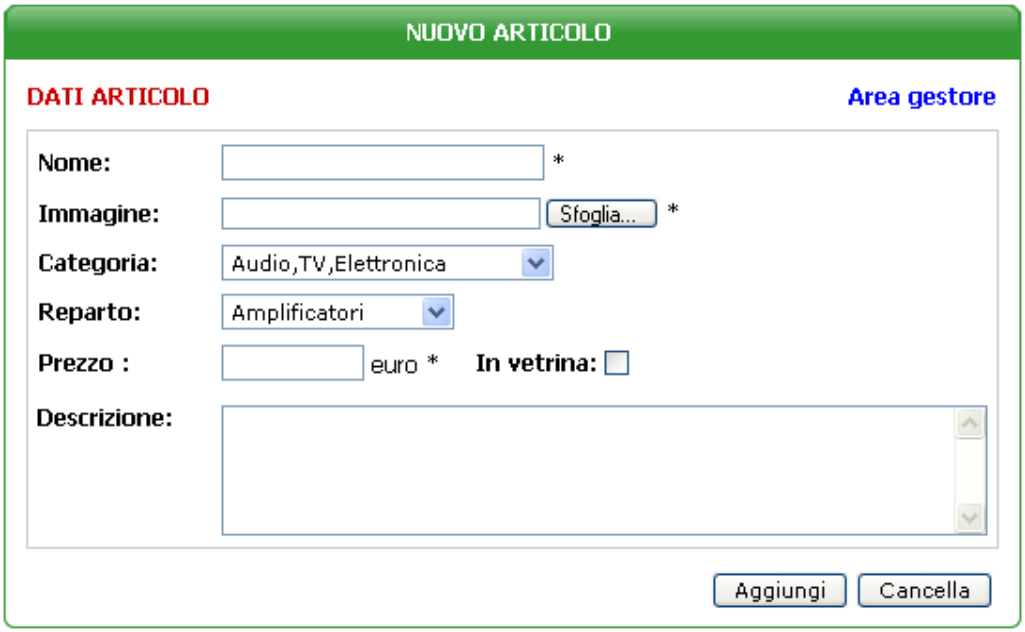

Figura 4.30: View Nuovo Articolo

Modifica Articolo

Permette di poter modificare i dati di un articolo. E' possibile accedere a questa vista dal negozio gestito, attraverso un apposito link di modifica disponibile per ogni articolo.

Figura 4.31: View Modifica Dati Articolo

Ordini Ricevuti

Visualizza il numero, la data, lo stato e il prezzo degli ordini ricevuti. Essa permette di poter visualizzare il dettaglio di un ordine ricevuto per poterne eventualmente modificare lo stato. Questa vista è simile a quelle degli ordini effettuati, solo che in questo caso gli ordini vengono organizzati in base all'username del cliente che li ha effettuati ed è possibile visualizzare i suoi dati attraverso un link. E' possibile accedere a questa vista dall'area del cliente.

Dati Cliente dell'Ordine Ricevuto

Visualizza i dati del cliente che ha effettuato l'ordine. Permette di poter contattare il cliente attraverso un link alla sua email. E' possibile accedere a questa vista dagli ordini ricevuti.

Dettaglio Ordine Ricevuto

Visualizza i dati di un ordine ricevuto. Questa vista è simile a quella del dettaglio di un ordine effettuato solo che in questo caso è possibile modificare anche lo stato dell'ordine. E' possibile accedere a questa vista dagli ordini ricevuti.

Gestione Modalità di Spedizione

Visualizza la descrizione e il costo delle modalità di spedizione. Essa permette di poter accedere alla modifica di una modalità di spedizione o di eliminarne una a patto che non non sia l'unica presente. Inoltre è possibile inserire una nuova modalità di spedizione. E' possibile accedere a questa vista dall'area del gestore.

Figura 4.32: View Gestione Modalità di Spedizione

Modifica Modalità di Spedizione

Permette di modificare la descrizione e il costo di una modalità di spedizione. E' possibile accedere a questa vista a partire dalla gestione delle modalità di spedizione.

Gestione Modalità di Pagamento

Questa vista, a parte il tipo di modalità, è del tutto uguale a quella per la gestione delle modalità di spedizione.

Modifica Modalità di Pagamento

Questa vista, a parte il tipo di modalità, è del tutto uguale a quella per la gestione delle modalità di spedizione.

4.6.7 View Amministratore

Rappresenta le viste alle quali può accedere l'*Amministratore*. Tali viste vengono presentate di seguito.

Area Amministratore

Rappresenta la vista privata dell'amministratore. Da essa è possibile accedere alle funzionalità previste per l'amministratore. Per quanto riguarda le funzionalità ereditate dal cliente, è possibile effettuare la gestione dei dati e la gestione degli ordini effettuati. Per le funzionalità di amministrazione dei clienti, dei gestori e dei negozi è possibile accedere all'aree di visualizzazione, di modifica dello stato e di eliminazione. Infine è possibile ritornare al negozio che si stava visitando. E' possibile accedere a questa vista dal menù di NetShop, dal menù del negozio o dall'area di benvenuto, disponibile dopo il login in tutte le viste al posto dell'area di login.

Figura 4.33: View Area Amministratore

Visualizza/Modifica stato/Elimina Clienti

Visualizza l'username, la data di registrazione e lo stato dei clienti registrati a NetShop. Da essa è possibile passare alla visualizzazione dei dati di un cliente, cambiarne lo stato o eliminarlo definitivamente. E' possibile accedere a queste vista dall'area dell'amministratore.

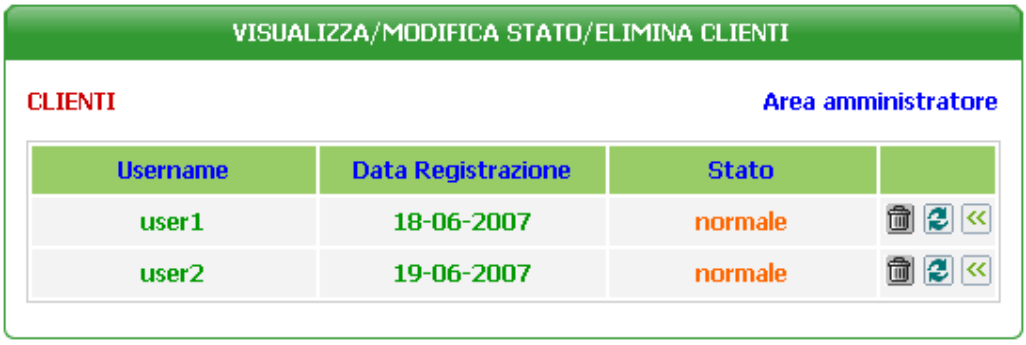

Figura 4.34: View Visualizza/Modifica Stato/Elimina Clienti

Visualizza Dati/Modifica Stato Cliente

Visualizza i dati di un cliente dando la possibilità di poterne modificare lo stato. Da essa è possibile contattare il cliente attraverso un link al suo indirizzo email. E' possibile accedere a questa vista dall'area di visualizzazione dei clienti.

Visualizza/Modifica stato/Elimina Gestori

Visualizza l'username, il nome del negozio, la data di registrazione e lo stato dei gestori registrati a NetShop. Da essa è possibile passare alla visualizzazione dei dati di un gestore, cambiarne lo stato o eliminarlo definitivamente con il suo negozio. E' possibile accedere a queste vista dall'area dell'amministratore.

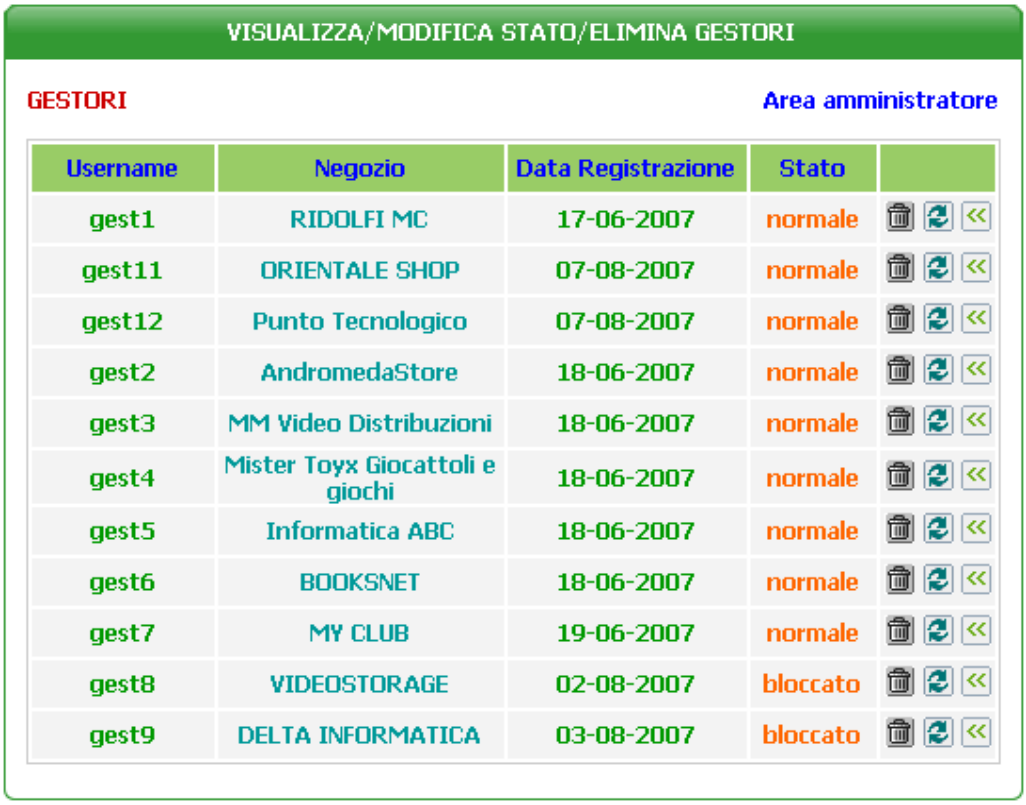

Figura 4.35: View Visualizza/Modifica Stato/Elimina Gestori

Visualizza Dati/Modifica Stato Gestore

Visualizza i dati di un gestore dando la possibilità di poterne modificare lo stato. Da essa è possibile contattare il gestore attraverso un link al suo indirizzo email. E' possibile accedere a questa vista dall'area di visualizzazione dei gestori.

Visualizza/Modifica Stato Negozi

Visualizza il nome, l'username del gestore, la data di creazione e lo stato dei negozi presenti in NetShop. Da essa è possibile passare alla visualizzazione dei dati di un negozio o aggiornare lo stato. E' possibile accedere a questa vista dall'area dell'amministratore.

Figura 4.36: View Visualizza/Modifica Stato Negozi

Visualizza Dati/Modifica Stato Negozio

Visualizza i dati di un negozio dando la possibilità di poterne modificare lo stato. E' possibile accedere a questa vista dall'area di visualizzazione dei negozi.

4.7 Classi di uso generale

Per l'applicazione sviluppata sono state realizzate delle classi di uso generale, che permettono di eseguire delle funzionalità per i diversi livelli dell'applicazione.

Le classi di uso generale implementate sono le seguenti:

- *ChangeLocaleAction*: action che viene utilizzata per il cambio del *locale*; permette di cambiare la lingua dall'italiano all'inglese e viceversa;
- *Immagine*: servlet che viene utilizzata per visualizzare le immagini contenute nella base dati, all'interno delle pagine JSP;
- *PasswordDynaValidatorForm*: classe che estende la classe *DynaValidatorForm* fornendo la validazione per la verifica della password con la sua conferma;
- *MailUtility*: classe contenente metodi *static* per l'invio di email verso gli utenti dell'applicazione.

4.8 Eccezioni

La gestione delle eccezioni è stata effettuata utilizzando un eccezione globale definita nel file di configurazione di Struts, che viene lanciata ogni volta che si verifica una situazione di errore. Al lancio dell'eccezione le viene passata la chiave per il messaggio di errore da visualizzare, contenuto nel Resource Bundle.

Si è scelto di lanciare tale eccezione solo all'interno delle action, mentre le situazioni di errore legate al Model, che riguardano in particolare l'accesso alla base dati, vengono gestite visualizzando un messaggio di errore solo in fase di debug nella finestra della console, questo per disaccoppiare il Model dal framework e quindi rendere meno complesso il riutilizzo del Model con un altro framework.

L'eccezione globale definita risulta avere il seguente codice XML:

```
<global-exceptions>
  <exception key="error" path="/error.do"
    type="org.apache.struts.util.ModuleException"/>
</global-exceptions>
```

4.9 Sicurezza

Per quanto riguarda la sicurezza l'applicazione ha previsto l'implementazione delle due seguenti forme:

- divieto di accesso alle aree private per gli utenti non autorizzati;
- trasferimento sicuro dei dati attraverso l'utilizzo del protocollo SSL (Secure Socket Layer).

La prima forma di sicurezza è stata realizzata attraverso l'utilizzo della funzione *check()*, che è definita nella classe *BaseAction* e nella classe *BaseDispatchAction* ed è quindi ereditata da tutte le action che estendono quest'ultime. La funzione *check()* è richiamata dalle action che visualizzano dati contenuti in aree private, essa effettua il controllo del tipo dell'utente che sta accendo alla risorsa, nel caso in cui il tipo è diverso da quello richiesto ritorna al chiamante un valore falso che impedisce all'accesso da parte dall'utente alla risorsa richiesta. Il codice della funzione *check()* risulta essere il seguente:

```
public boolean check(HttpServletRequest request,String tipo){

  HttpSession session = request.getSession();
  if(session==null) return false;
  else{
    if(session.getAttribute("utente")==null) return false;

    Utente utente = (Utente)session.getAttribute("utente");

    // Verifica se il tipo dell'utente è uguale a quello richiesto
    if(utente.getTipo().equals(tipo)) return true;
    else return false;
  }
}
```

La seconda forma di sicurezza ha permesso di crittografare i dati inviati attraverso la rete, in modo da renderli incomprensibili a chiunque riesca ad intercettarli in maniera fraudolenta. Per implementare tale forma di sicurezza si sono rese necessarie le seguenti operazioni:

- creazione di un certificato digitale di esempio;
- abilitazione del Connector SSL nel Web Container Apache Tomcat;
- configurazione della pagine protette nel Deployment Descriptor dell'applicazione.

La creazione del certificato digitale è stata effettuta facendo uso di uno degli strumenti messi a disposizione dall'SDK di Java2, ossia *keytool*, che produce un cosiddetto *keystore*. Quest'ultimo permette inoltre di specificare il tipo di algoritmo da utilizzare per la crittografia, che nel caso specifico è l'*RSA*. L'istruzione da lanciare a riga di comando è la seguente:

```
keytool -genkey -alias tomcat -keyalg RSA
```

Una volta creato il certificato, è stato abilitato il Connector SSL del Web Container Tomcat, mediante il quale è possible redirezionare tutte le comunicazioni protette, verso una porta specificata e tramite protocollo HTTP. Per fare questo è stato necessario apportare una modifica al file di configurazione *server.xml*, all'interno del quale è stato specificato il keystore da utilizzare e la relativa password.

```
<Connector port="8443" maxThreads="150"
  minSpareThreads="25" maxSpareThreads="75"
  enableLookups="false"
  disableUploadTimeout="true"
  acceptCount="100" debug="0"
  scheme="https" secure="true"
  clientAuth="false"
  sslProtocol="TLS" keystorePass="changeit"
  keystoreFile="C:\Documents and Settings\Luigi\.keystore"/>
```

Per quanto riguarda la configurazione delle pagine protette, all'interno del Deployment Descriptor *web.xml*, si è deciso di rendere tutta l'applicazione

sicura, garantendo la trasmissione tramite SSL per tutte le pagine a partire dalla pagina di iniziale *index.jsp*.

```
<security-constraint>
  <display-name>SSL Constraint</display-name>
  <web-resource-collection>
    <web-resource-name>Automatic SSL Forwarding
    </web-resource-name>
    <url-pattern>/index.jsp</url-pattern>
    <http-method>GET</http-method>
    <http-method>PUT</http-method>
    <http-method>POST</http-method>
    <http-method>DELETE</http-method>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

In questo modo, alla richiesta dell'URL iniziale viene segnalata la redirezione verso una connessione sicura garantita da un certificato digitale. La trasmissione dei dati tra client e server avverrà tramite il protocollo SSL in modalità crittografata.

4.10 Logging

Per quanto riguardo il loggin, si fatto uso di *log4j*. La configurazione è stata effettua tramite il file di proprietà *lo4j.properties*, con il seguente codice:

```
log4j.rootLogger=INFO,rolling
log4j.appender.rolling=org.apache.log4j.RollingFileAppender
log4j.appender.rolling.File=NetShop.log
log4j.appender.rolling.MaxFileSize=100KB
log4j.appender.rolling.MaxBackupIndex=1
log4j.appender.rolling.layout=org.apache.log4j.PatternLayout
log4j.appender.rolling.layout.ConversionPattern=
  %d{ABSOLUTE} - %p %c - %m%n
```

Dal file di configurazione si osserva che tutti i messaggi di log generati vengono salvati nel file *NetShop.log*, il quale di default viene creato nella directory di Eclipse. Si osserva ancora che per la generazione automatica dei messaggi di log è stato utilizzato il livello INFO, che permette di prelevare

tra i messaggi generati solo quelli relativi agli eventi significativi nel ciclo di esecuzione dell'applicazione.

Per quanto riguarda invece la generazione dei messaggi di log da parte dell'applicazione, essa è stata inserita nei punti importanti dell'applicazione, per poter venire a conoscenza delle esecuzione di particolari operazioni da parte di utenti autorizzati e non autorizzati.

Conclusioni

Giunti al termine di questo lavoro tesi, è possibile affermare che l'obiettivo prefissato è stato raggiunto.

L'applicazione realizzata soddisfa fedelmete tutte le specifiche richieste in fatto di funzionalità, sicurezza e facilità d'uso. Inoltre, è stata acquisita dell'esperienza nella progettazione e nella realizzazione di Web applications utilizzando il framework Jakarta Struts.

Riguardo agli sviluppi futuri sono rimaste aperte varie opportunità. Una di queste è l'aggiunta di un vero e proprio motore di indicizzazione per dare l'opportunità agli utenti di fare ricerche veloci e mirate.

Infine, nonostante Struts sia un framework completo e potente, in questi ultimi anni il mondo dell'informatica è in agitazione per l'imminente avvento del *Web 2.0*, con l'utilizzo massivo di *AJAX (Asynchronous JavaScript & XML)* per la realizzazione di applicazioni che non risiedono più solo sui server ma che coinvolgono soprattutto i client, spostando molte delle funzionalità su di essi. Proprio a questo scopo è nato il framework *JSF (Java Server Faces)* che, volendo, può anche essere usato in combinazione con Struts per poter migliorare l'interfaccia utente.

Appendice A

Codice File di configurazione

A.1 File di configurazione dell'applicazione

Viene presentato di seguito il codice relativo al file *web.xml*.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
<display-name>NetShop</display-name>

<!-- Action Servlet Configuration -->

<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet</servlet-
     class>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet>
  <servlet-name>img</servlet-name>
  <servlet-class>general.mvc.Immagine</servlet-class>
</servlet>

<!-- Action Servlet Mapping -->

<servlet-mapping>
```

```
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>img</servlet-name>
  <url-pattern>/img</url-pattern>
</servlet-mapping>

<!-- The Usual Welcome File List -->

<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>

<taglib>
  <taglib-uri>/WEB-INF/struts-html</taglib-uri>
  <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
</taglib>
<taglib>
  <taglib-uri>/WEB-INF/struts-bean</taglib-uri>
  <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
</taglib>
<taglib>
  <taglib-uri>/WEB-INF/struts-logic</taglib-uri>
  <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
</taglib>
<taglib>
  <taglib-uri>/WEB-INF/struts-tiles</taglib-uri>
  <taglib-location>/WEB-INF/struts-tiles.tld</taglib-location>
</taglib>

<security-constraint>
  <display-name>SSL Constraint</display-name>
  <web-resource-collection>
    <web-resource-name>Automatic SSL Forwarding
    </web-resource-name>
    <url-pattern>/index.jsp</url-pattern>
    <http-method>GET</http-method>
    <http-method>PUT</http-method>
    <http-method>POST</http-method>
    <http-method>DELETE</http-method>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>

<login-config>
  <auth-method>BASIC</auth-method>
</login-config>
</web-app>
```

A.2 File di configurazione di Struts

Viene presentato di seguito il codice relativo al file *struts-config.xml*.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//
    DTD Struts Configuration 1.1//EN" "http://jakarta.apache.org/
    struts/dtds/struts-config_1_1.dtd">

<struts-config>

<data-sources>
  <data-source key="dataSource"
    type="org.apache.struts.util.GenericDataSource">
    <set-property property="maxWait" value="5000"/>
    <set-property property="url"
      value="jdbc:mysql://localhost:3306/NetShop?autoReconnect=
          true"/>
    <set-property property="description"
      value="MySql Data Source"/>
    <set-property property="minCount" value="2"/>
    <set-property property="driverClass"
      value="com.mysql.jdbc.Driver"/>
    <set-property property="password" value="123456"/>
    <set-property property="maxActive" value="20"/>
    <set-property property="autoCommit" value="true"/>
    <set-property property="maxCount" value="10"/>
    <set-property property="user" value="root"/>
  </data-source>
</data-sources>

<form-beans>
  <form-bean name="cercaForm"
    type="org.apache.struts.validator.DynaValidatorForm">
    <form-property name="prezzoMin" type="java.lang.String"/>
    <form-property name="nome" type="java.lang.String"/>
    <form-property name="prezzoMax" type="java.lang.String"/>
    <form-property initial="0" name="idReparto" type="java.lang.
        Integer"/>
    <form-property initial="0" name="idCategoria" type="java.lang.
        Integer"/>
    <form-property name="descrizione" type="java.lang.String"/>
  </form-bean>

  <form-bean name="carrelloForm"
   type="org.apache.struts.validator.DynaValidatorForm">
   <form-property name="rimuovi" type="java.lang.Integer[]"/>
   <form-property name="quantita" type="java.lang.String[]"/>
  </form-bean>

  <form-bean name="datiForm"
```

```
  type="org.apache.struts.validator.DynaValidatorForm">
  <form-property name="provincia" type="java.lang.String"/>
  <form-property name="cognome" type="java.lang.String"/>
  <form-property name="scelta" type="java.lang.String"/>
  <form-property name="nome" type="java.lang.String"/>
  <form-property name="indirizzo" type="java.lang.String"/>
  <form-property name="telefono" type="java.lang.String"/>
  <form-property name="email" type="java.lang.String"/>
  <form-property name="cap" type="java.lang.String"/>
  <form-property name="codFiscale" type="java.lang.String"/>
  <form-property name="citta" type="java.lang.String"/>
</form-bean>
<form-bean name="datiClienteForm"
  type="general.mvc.PasswordDynaValidatorForm">
  <form-property name="rispSegreta" type="java.lang.String"/>
  <form-property name="password" type="java.lang.String"/>
  <form-property name="username" type="java.lang.String"/>
  <form-property name="confermaPassword" type="java.lang.String"
     />
  <form-property name="domSegreta" type="java.lang.String"/>
  <form-property name="autorizzazione" type="java.lang.Boolean"/
     >
</form-bean>
<form-bean name="datiGestoreForm"
  type="general.mvc.PasswordDynaValidatorForm">
  <form-property name="costoPagamento" type="java.lang.String"/>
  <form-property name="logo" type="org.apache.struts.upload.
     FormFile"/>
  <form-property name="domSegreta" type="java.lang.String"/>
  <form-property name="descrizione" type="java.lang.String"/>
  <form-property name="nomeNegozio" type="java.lang.String"/>
  <form-property name="rispSegreta" type="java.lang.String"/>
  <form-property name="partitaIva" type="java.lang.String"/>
  <form-property name="costoSpedizione" type="java.lang.String"/
     >
  <form-property name="modPagamento" type="java.lang.String"/>
  <form-property name="username" type="java.lang.String"/>
  <form-property name="password" type="java.lang.String"/>
  <form-property name="modSpedizione" type="java.lang.String"/>
  <form-property name="autorizzazione" type="java.lang.Boolean"/
     >
  <form-property name="confermaPassword" type="java.lang.String"
     />
  <form-property initial="20" name="iva" type="java.lang.String"
     />
  <form-property name="fax" type="java.lang.String"/>
</form-bean>

<form-bean name="loginForm"
  type="org.apache.struts.validator.DynaValidatorForm">
  <form-property name="username" type="java.lang.String"/>
  <form-property name="password" type="java.lang.String"/>
```

```
</form-bean>

 <form-bean name="usernameForm"
  type="org.apache.struts.validator.DynaValidatorForm">
  <form-property name="username" type="java.lang.String"/>
 </form-bean>
 <form-bean name="rispSegretaForm"
  type="org.apache.struts.validator.DynaValidatorForm">
  <form-property name="rispSegreta" type="java.lang.String"/>
 </form-bean>

 <form-bean name="ordineForm"
  type="org.apache.struts.validator.DynaValidatorForm">
  <form-property name="idModPagamento" type="java.lang.Integer"/
     >
  <form-property name="idModSpedizione" type="java.lang.Integer"
     />
  <form-property name="stato" type="java.lang.String"/>
  <form-property name="id" type="java.lang.Integer"/>
 </form-bean>

 <form-bean name="datiAutenticazioneForm"
  type="general.mvc.PasswordDynaValidatorForm">
  <form-property name="confermaPassword" type="java.lang.String"
     />
  <form-property name="password" type="java.lang.String"/>
  <form-property name="rispSegreta" type="java.lang.String"/>
  <form-property name="domSegreta" type="java.lang.String"/>
 </form-bean>

 <form-bean name="negozioForm"
  type="org.apache.struts.validator.DynaValidatorForm">
  <form-property name="descrizione" type="java.lang.String"/>
  <form-property name="nuovoLogo" type="org.apache.struts.upload
     .FormFile"/>
  <form-property name="partitaIva" type="java.lang.String"/>
  <form-property name="iva" type="java.lang.String"/>
  <form-property name="nome" type="java.lang.String"/>
  <form-property name="fax" type="java.lang.String"/>
 </form-bean>

 <form-bean name="repartoForm"
  type="org.apache.struts.validator.DynaValidatorForm">
  <form-property name="nome" type="java.lang.String"/>
 </form-bean>

 <form-bean name="nuovoArticoloForm"
  type="org.apache.struts.validator.DynaValidatorForm">
  <form-property name="prezzo" type="java.lang.String"/>
  <form-property name="idReparto" type="java.lang.Integer"/>
  <form-property name="idCategoria" type="java.lang.Integer"/>
  <form-property name="descrizione" type="java.lang.String"/>
```

```
    <form-property name="nome" type="java.lang.String"/>
    <form-property initial="false" name="inVetrina" type="java.
       lang.Boolean"/>
    <form-property name="immagine" type="org.apache.struts.upload.
       FormFile"/>
  </form-bean>
  <form-bean name="articoloForm"
    type="org.apache.struts.validator.DynaValidatorForm">
    <form-property name="idReparto" type="java.lang.Integer"/>
    <form-property name="nome" type="java.lang.String"/>
    <form-property name="idCategoria" type="java.lang.Integer"/>
    <form-property name="prezzo" type="java.lang.String"/>
    <form-property name="nuovaImmagine" type="org.apache.struts.
       upload.FormFile"/>
    <form-property initial="false" name="inVetrina" type="java.
       lang.Boolean"/>
    <form-property name="descrizione" type="java.lang.String"/>
  </form-bean>

  <form-bean name="modConsegnaForm"
    type="org.apache.struts.validator.DynaValidatorForm">
    <form-property name="tipo" type="java.lang.String"/>
    <form-property name="descrizione" type="java.lang.String"/>
    <form-property name="costo" type="java.lang.String"/>
  </form-bean>
</form-beans>

<global-exceptions>
<exception key="error" path="/error.do" type="org.apache.struts.
   util.ModuleException"/>
</global-exceptions>

<global-forwards>
<forward name="goHome"
path="/visAreeNetShop.do?action=visHome"/>
<forward name="goNegozio"
path="/visAreeNegozio.do?action=visHome"/>
</global-forwards>

<action-mappings type="org.apache.struts.config.
   SecureActionConfig">

  <!-- Actions Area Utente -->

  <action path="/cambiaLingua"
    type="general.mvc.SetLocaleAction"/>

  <action path="/error" type="base.controller.BaseAction">
    <forward name="success" path="/View/Utente/Error.jsp"/>
  </action>

  <action path="/visAreeNetShop"
```

```
  parameter="action"
  type="utente.controller.VisAreeNetShopAction">
  <forward name="home"
    path="/View/Utente/Home.jsp"/>
  <forward name="negoziCategoria"
    path="/View/Utente/NegoziCategoria.jsp"/>
  <forward name="articoliCategoria"
    path="/View/Utente/ArticoliCategoria.jsp"/>
  <forward name="articolo"
    path="/View/Utente/ArticoloNetShop.jsp"/>
</action>
<action path="/visInfo" type="base.controller.BaseAction">
  <forward name="success"
    path="/View/Utente/InfoNetShop.jsp"/>
</action>

<action path="/visAreeNegozio"
  parameter="action"
  type="utente.controller.VisAreeNegozioAction">
  <forward name="home"
    path="/View/Utente/HomeNegozio.jsp"/>
  <forward name="reparto"
    path="/View/Utente/Reparto.jsp"/>
  <forward name="articoli"
    path="/View/Utente/ArticoliNegozio.jsp"/>
  <forward name="articoliCategoria"
    path="/View/Utente/ArticoliCategoriaNegozio.jsp"/>
  <forward name="articolo"
    path="/View/Utente/ArticoloNegozio.jsp"/>
  <forward name="info"
    path="/View/Utente/InfoNegozio.jsp"/>
</action>

<action path="/cercaNegozi"
  input="/ricAvanzataNegozi.do"
  parameter="action"
  name="cercaForm" scope="request"
  type="utente.controller.CercaAction">
  <forward name="success"
    path="/View/Utente/NegoziTrovati.jsp"/>
</action>
<action path="/ricAvanzataNegozi"
  type="base.controller.BaseAction">
  <forward name="success"
    path="/View/Utente/RicercaAvanzataNegozi.jsp"/>
</action>
<action path="/cercaArticoliNetShop"
  input="/ricAvanzataArticoliNetShop.do"
  parameter="action"
  name="cercaForm" scope="request"
  type="utente.controller.CercaAction" validate="true">
  <forward name="success"
```

```
    path="/View/Utente/ArticoliTrovatiCategoria.jsp"/>
</action>
<action path="/ricAvanzataArticoliNetShop"
  type="base.controller.BaseAction">
  <forward name="success"
    path="/View/Utente/RicercaAvanzataArticoliNetShop.jsp"/>
</action>
<action path="/cercaArticoliNegozio"
  input="/ricAvanzataArticoliNegozio.do"
  parameter="action"
  name="cercaForm" scope="request"
  type="utente.controller.CercaAction">
  <forward name="success"
    path="/View/Utente/ArticoliTrovatiNegozio.jsp"/>
</action>
<action path="/ricAvanzataArticoliNegozio"
  type="base.controller.BaseAction">
  <forward name="success"
    path="/View/Utente/RicercaAvanzataArticoliNegozio.jsp"/>
</action>

<action path="/visualizzaCarrello"
  parameter="action"
  type="utente.controller.GestioneCarrelloAction">
  <forward name="success"
    path="/View/Utente/Carrello.jsp"/>
</action>
<action path="/aggiungiCarrello"
  parameter="action"
  type="utente.controller.GestioneCarrelloAction">
  <forward name="success"
    path="/visualizzaCarrello.do?action=visualizza"/>
</action>
<action path="/aggiornaCarrello"
  parameter="action"
  name="carrelloForm" scope="request"
  type="utente.controller.GestioneCarrelloAction">
  <forward name="success"
    path="/visualizzaCarrello.do?action=visualizza"/>
</action>
<action path="/svuotaCarrello"
  parameter="action"
  type="utente.controller.GestioneCarrelloAction">
  <forward name="success"
    path="/visualizzaCarrello.do?action=visualizza"/>
</action>

<action path="/registrazione" type="base.controller.BaseAction"
    >
<forward name="success"
  path="/View/Utente/Registrazione.jsp"/>
</action>
```

```
<action path="/registrazione1"
  input="/View/Utente/Registrazione1.jsp"
  name="datiForm" scope="session"
  type="base.controller.BaseAction">
  <forward name="success"
    path="/View/Utente/Registrazione1.jsp"/>
</action>
<action path="/registrazione2"
  input="/registrazione1.do"
  name="datiForm" scope="session"
  type="base.controller.BaseAction" validate="true">
  <forward name="success"
    path="/View/Utente/Registrazione2.jsp"/>
</action>
<action path="/effettuaRegistrazioneCliente"
  input="/registrazione2.do"
  name="datiClienteForm"
  parameter="action" scope="request"
  type="utente.controller.RegistrazioneAction"
  validate="true">
  <forward name="clienteRegistrato"
    path="/View/Cliente/ClienteRegistrato.jsp"/>
  <forward name="concludiOrdine"
    path="/View/Cliente/VisModConsegna.jsp"/>
</action>
<action path="/effettuaRegistrazioneGestore"
  input="/registrazione2.do" parameter="action"
  name="datiGestoreForm" scope="request"
  type="utente.controller.RegistrazioneAction" validate="true">
  <forward name="gestoreRegistrato"
    path="/View/Gestore/GestoreRegistrato.jsp"/>
  <forward name="concludiOrdine"
    path="/View/Cliente/VisModConsegna.jsp"/>
</action>

<action path="/login" type="base.controller.BaseAction">
  <forward name="success"
    path="/View/Utente/Login.jsp"/>
</action>
<action path="/effettuaLogin"
  input="/login.do"
  name="loginForm" scope="request"
  type="utente.controller.LoginAction"
  validate="true">
  <forward name="cliente"
    path="/View/Cliente/AreaCliente.jsp"/>
  <forward name="gestore"
    path="/View/Gestore/AreaGestore.jsp"/>
  <forward name="amministratore"
    path="/View/Amministratore/AreaAmministratore.jsp"/>
  <forward name="concludiOrdine"
    path="/View/Cliente/VisModConsegna.jsp"/>
```

```
</action>

<action path="/recuperaPassword"
  type="base.controller.BaseAction">
  <forward name="success"
    path="/View/Utente/RecuperoPassword.jsp"/>
</action>
<action path="/domSegreta"
  input="/recuperaPassword.do"
  parameter="action"
  name="usernameForm" scope="session"
  type="utente.controller.RecuperoPasswordAction">
  <forward name="success"
    path="/View/Utente/RecPassDomSegreta.jsp"/>
</action>
<action path="/effettuaRecupero"
  input="/domSegreta.do?action=recuperaDomandaSegreta"
  parameter="action"
  name="rispSegretaForm"
  type="utente.controller.RecuperoPasswordAction">
  <forward name="success"
    path="/View/Utente/RecPassEffettuato.jsp"/>
</action>

<!-- Actions Area Cliente -->

<action path="/effettuaOrdine"
  parameter="action"
  name="ordineForm"
  type="cliente.controller.EffettuaOrdineAction">
  <forward name="visModConsegna"
    path="/View/Cliente/VisModConsegna.jsp"/>
  <forward name="Login"
    path="/View/Utente/Login.jsp"/>
  <forward name="visRiepilogo"
    path="/View/Cliente/RiepilogoOrdine.jsp"/>
  <forward name="ordineEffettuato"
    path="/View/Cliente/OrdineEffettuato.jsp"/>
</action>

<action path="/areaPrivata"
  type="cliente.controller.AreaPrivataAction">
  <forward name="cliente"
    path="/View/Cliente/AreaCliente.jsp"/>
  <forward name="gestore"
    path="/View/Gestore/AreaGestore.jsp"/>
  <forward name="amministratore"
    path="/View/Amministratore/AreaAmministratore.jsp"/>
</action>

<action path="/visDatiPersonali"
  type="base.controller.BaseAction">
```

```
  <forward name="success"
    path="/View/Cliente/ModificaDatiPersonali.jsp"/>
</action>
<action path="/aggiornaDatiPersonali"
  input="/visDatiPersonali.do?notIniz=true"
  parameter="action"
  name="datiForm" scope="request"
  type="cliente.controller.AggiornaDatiAction" validate="true">
  <forward name="areaPrivata" path="/areaPrivata.do"/>
</action>
<action path="/visDatiAutenticazione"
  type="base.controller.BaseAction">
  <forward name="success"
    path="/View/Cliente/ModificaDatiAutenticazione.jsp"/>
</action>
<action path="/aggiornaDatiAutenticazione"
  input="/visDatiAutenticazione.do?notIniz=true"
  parameter="action"
  name="datiAutenticazioneForm" scope="request"
  type="cliente.controller.AggiornaDatiAction"
  validate="true">
  <forward name="areaPrivata" path="/areaPrivata.do"/>
</action>

<action path="/gestioneOrdiniEffettuati"
  parameter="action"
  type="cliente.controller.GestioneOrdiniEffettuatiAction">
  <forward name="visualizza"
    path="/View/Cliente/OrdiniEffettuati.jsp"/>
  <forward name="dettaglio"
    path="/View/Cliente/DettaglioOrdine.jsp"/>
</action>

<action path="/logout" type="cliente.controller.LogoutAction"/>

<!-- Actions Area Gestore -->

<action path="/visDatiNegozio"
  type="base.controller.BaseAction">
  <forward name="success"
    path="/View/Gestore/ModificaDatiNegozio.jsp"/>
</action>
<action path="/aggiornaDatiNegozio"
  input="/visualizzaDatiNegozio.do?notIniz=true"
  name="negozioForm"
  type="gestore.controller.AggiornaDatiNegozioAction"
  validate="true">
  <forward name="areaPrivata"
    path="/areaPrivata.do"/>
</action>

<action path="/gestioneReparti"
```

```
  parameter="action"
  type="gestore.controller.GestioneRepartiAction">
 <forward name="success"
   path="/View/Gestore/GestioneReparti.jsp"/>
</action>
<action path="/aggiungiReparto"
  input="/gestioneReparti.do?action=visualizza&amp;&amp;notIniz=
     true"
  name="repartoForm"
  parameter="action" scope="request"
  type="gestore.controller.GestioneRepartiAction"
  validate="true">
  <forward name="success"
   path="/gestioneReparti.do?action=visualizza"/>
</action>
<action path="/modificaReparto"
  parameter="action"
  type="gestore.controller.GestioneRepartiAction">
  <forward name="success"
   path="/View/Gestore/ModificaReparto.jsp"/>
</action>
<action path="/aggiornaReparto"
  input="/modificaReparto.do?action=modifica&amp;&amp;notIniz=
     true"
  name="repartoForm"
  parameter="action" scope="request"
  type="gestore.controller.GestioneRepartiAction"
  validate="true">
  <forward name="success"
   path="/gestioneReparti.do?action=visualizza"/>
</action>
<action path="/eliminaReparto"
  parameter="action"
  type="gestore.controller.GestioneRepartiAction">
  <forward name="success"
   path="/gestioneReparti.do?action=visualizza"/>
</action>

<action path="/nuovoArticolo"
  parameter="action"
  type="gestore.controller.GestioneArticoliAction">
  <forward name="success"
   path="/View/Gestore/NuovoArticolo.jsp"/>
</action>
<action path="/aggiungiArticolo"
  input="/nuovoArticolo.do?action=nuovo&amp;&amp;notIniz=true"
  name="nuovoArticoloForm"
  parameter="action" scope="request"
  type="gestore.controller.GestioneArticoliAction"
  validate="true">
  <forward name="areaPrivata" path="/areaPrivata.do"/>
</action>
```

```
<action path="/modificaArticolo"
  parameter="action"
  type="gestore.controller.GestioneArticoliAction">
  <forward name="success"
    path="/View/Gestore/ModificaArticolo.jsp"/>
</action>
<action path="/aggiornaArticolo"
  input="/modificaArticolo.do?action=modifica&amp;&amp;notIniz=
     true"
  parameter="action"
  name="articoloForm" scope="request"
  type="gestore.controller.GestioneArticoliAction"
  validate="true"/>
<action path="/eliminaArticolo"
  parameter="action"
  type="gestore.controller.GestioneArticoliAction"/>

<action path="/gestioneOrdiniRicevuti"
  parameter="action"
  type="gestore.controller.GestioneOrdiniRicevutiAction">
  <forward name="visualizza"
    path="/View/Gestore/OrdiniRicevuti.jsp"/>
  <forward name="datiCliente"
    path="/View/Gestore/DatiClienteOrdine.jsp"/>
  <forward name="modificaStato"
    path="/View/Gestore/DettaglioOrdineRicevuto.jsp"/>
</action>
<action path="/aggiornaStatoOrdineRicevuto"
  parameter="action"
  name="ordineForm" scope="request"
  type="gestore.controller.GestioneOrdiniRicevutiAction">
  <forward name="success"
    path="/gestioneOrdiniRicevuti.do?action=visualizza"/>
</action>

<action path="/gestioneModalitaConsegna"
  parameter="action"
  type="gestore.controller.GestioneModConsegnaAction">
  <forward name="success"
    path="/View/Gestore/GestioneModConsegna.jsp"/>
</action>
<action path="/aggiungiModalitaConsegna"
  input="/gestioneModalitaConsegna.do?action=visualizza&amp;&amp
     ;notIniz=true"
  name="modConsegnaForm"
  parameter="action" scope="request"
  type="gestore.controller.GestioneModConsegnaAction"
  validate="true">
  <forward name="success"
    path="/gestioneModalitaConsegna.do?action=visualizza"/>
</action>
<action path="/modificaModalitaConsegna"
```

```
  parameter="action"
  type="gestore.controller.GestioneModConsegnaAction">
  <forward name="success"
    path="/View/Gestore/ModificaModalitaConsegna.jsp"/>
</action>
<action path="/aggiornaModalitaConsegna"
  input="/modificaModalitaConsegna.do?action=modifica&amp;&amp;
     notIniz=true"
  parameter="action"
  name="modConsegnaForm" scope="request"
  type="gestore.controller.GestioneModConsegnaAction"
  validate="true">
  <forward name="success"
    path="/gestioneModalitaConsegna.do?action=visualizza"/>
</action>
<action path="/eliminaModalitaConsegna"
  parameter="action"
  type="gestore.controller.GestioneModConsegnaAction">
  <forward name="success"
    path="/gestioneModalitaConsegna.do?action=visualizza"/>
</action>

<!-- Actions Area Amministratore -->

<action path="/amministClientiGestori"
  parameter="action"
  type="amministratore.controller.AmministClientiGestoriAction">
  <forward name="visClienti"
    path="/View/Amministratore/Clienti.jsp"/>
  <forward name="visGestori"
    path="/View/Amministratore/Gestori.jsp"/>
  <forward name="modificaStato"
    path="/View/Amministratore/ModificaStato.jsp"/>
</action>
<action path="/aggiornaStatoClienteGestore"
  parameter="action"
  type="amministratore.controller.AmministClientiGestoriAction">
  <forward name="visClienti"
    path="/amministClientiGestori.do?action=visualizzaClienti"/>
  <forward name="visGestori"
    path="/amministClientiGestori.do?action=visualizzaGestori"/>
</action>
<action path="/eliminaClienteGestore"
  parameter="action"
  type="amministratore.controller.AmministClientiGestoriAction">
  <forward name="visClienti"
    path="/amministClientiGestori.do?action=visualizzaClienti"/>
  <forward name="visGestori"
    path="/amministClientiGestori.do?action=visualizzaGestori"/>
</action>

<action path="/amministNegozi"
```

```
    parameter="action"
    type="amministratore.controller.AmministNegoziAction">
    <forward name="visualizza"
      path="/View/Amministratore/Negozi.jsp"/>
    <forward name="modificaStato"
      path="/View/Amministratore/ModificaStatoNegozio.jsp"/>
  </action>
  <action path="/aggiornaStatoNegozio"
    parameter="action"
    type="amministratore.controller.AmministNegoziAction">
    <forward name="success"
      path="/amministNegozi.do?action=visualizza"/>
  </action>
</action-mappings>

<controller locale="true"/>

<message-resources parameter="NetShopMessageResources"/>

<plug-in className="org.apache.struts.tiles.TilesPlugin">
  <set-property property="moduleAware" value="true"/>
  <set-property property="definitions-config"
    value="/WEB-INF/tiles-defs.xml"/>
</plug-in>
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
  <set-property property="pathnames"
    value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"/>
</plug-in>

</struts-config>
```

Appendice B

Codice Base dati

Viene presentato di seguito il codice relativo alla costruzione e alla inizializzazione della base dati.

```
/* Table: ARTICOLO */

create table ARTICOLO
(
  IDARTICOLO integer not null auto_increment,
  IDCATEGORIA integer not null,
  IDREPARTO integer not null,
  NOME varchar(50) not null,
  DESCRIZIONE long varchar,
  INVETRINA smallint default 0,
  IMMAGINE MEDIUMBLOB not null,
  PREZZO float not null,
  ELIMINATO smallint default false,
  constraint PK_ARTICOLO primary key (IDARTICOLO)
);

/* Table: CATEGORIA */

create table CATEGORIA
(
  IDCATEGORIA integer not null auto_increment,
  NOME varchar(50) not null,
  constraint PK_CATEGORIA primary key (IDCATEGORIA)
);

/* Table: CLIENTE */

create table CLIENTE
(
  USERNAME varchar(30) not null,
```

```
  IDNEGOZIO integer,
  PASSWORD varchar(30) not null,
  NOME varchar(50) not null,
  COGNOME varchar(50) not null,
  CODFISCALE char(16) not null,
  INDIRIZZO varchar(50) not null,
  CITTA varchar(50) not null,
  PROVINCIA char(2) not null,
  CAP char(5) not null,
  EMAIL varchar(50) not null,
  TELEFONO varchar(20),
  DOMSEGRETA varchar(50),
  RISPSEGRETA varchar(50),
  DATAREG date not null default '2007-01-01',
  BLOCCATO smallint default 0,
  TIPO varchar(20),
  constraint PK_CLIENTE primary key (USERNAME)
);

/* Table: LINEA */

create table LINEA
(
  IDLINEA integer not null auto_increment,
  IDARTICOLO integer not null,
  IDORDINE integer not null,
  QUANTITA integer not null,
  TOTALE float not null,
  constraint PK_LINEA primary key (IDLINEA)
);

/* Table: MODALITACONSEGNA */

create table MODALITACONSEGNA
(
  IDMODALITA integer not null auto_increment,
  IDNEGOZIO integer not null,
  COSTO float not null,
  DESCRIZIONE long varchar not null,
  ELIMINATA smallint default 0,
  TIPO varchar(20),
  constraint PK_MODALITACONSEGNA primary key (IDMODALITA)
);

/* Table: NEGOZIO */

create table NEGOZIO
(
  IDNEGOZIO integer not null default true auto_increment,
  USERNAME varchar(30) not null,
  NOME varchar(50) not null,
  DESCRIZIONE long varchar,
```

```
  LOGO MEDIUMBLOB not null,
  IVA integer not null default 20,
  PARTITAIVA char(11) not null,
  FAX varchar(20) not null,
  NASCOSTO smallint default true,
  constraint PK_NEGOZIO primary key (IDNEGOZIO)
);

/* Table: ORDINE */

create table ORDINE
(
  IDORDINE integer not null auto_increment,
  IDMODSPED integer not null,
  IDMODPAG integer not null,
  USERNAME varchar(30) not null,
  DATA date not null,
  IVA integer not null default 20,
  SPESE float not null,
  STATO varchar(20) not null,
  constraint PK_ORDINE primary key (IDORDINE)
);

/* Table: REPARTO */

create table REPARTO
(
  IDREPARTO integer not null auto_increment,
  IDNEGOZIO integer not null,
  NOME varchar(50) not null,
  constraint PK_REPARTO primary key (IDREPARTO)
);


/* Constraints */

alter table ARTICOLO
  add constraint FK_ARTICOLO_CATEGORIA foreign key (IDCATEGORIA)
  references CATEGORIA (IDCATEGORIA)
  on update restrict
  on delete restrict;

alter table ARTICOLO
  add constraint FK_ARTICOLO_REPARTO foreign key (IDREPARTO)
  references REPARTO (IDREPARTO)
  on update restrict
  on delete restrict;

alter table CLIENTE
  add constraint FK_CLIENTE_NEGOZIO foreign key (IDNEGOZIO)
  references NEGOZIO (IDNEGOZIO)
  on update restrict
```

```
  on delete restrict;

alter table LINEA
  add constraint FK_LINEA_ARTICOLO foreign key (IDARTICOLO)
  references ARTICOLO (IDARTICOLO)
  on update restrict
  on delete restrict;

alter table LINEA
  add constraint FK_LINEA_ORDINE foreign key (IDORDINE)
  references ORDINE (IDORDINE)
  on update restrict
  on delete restrict;

alter table MODALITACONSEGNA
  add constraint FK_MODALITA_NEGOZIO foreign key (IDNEGOZIO)
  references NEGOZIO (IDNEGOZIO)
  on update restrict
  on delete restrict;

alter table NEGOZIO
  add constraint FK_NEGOZIO_CLIENTE foreign key (USERNAME)
  references CLIENTE (USERNAME)
  on update restrict
  on delete restrict;

alter table ORDINE
  add constraint FK_ORDINE_MODPAG foreign key (IDMODPAG)
  references MODALITACONSEGNA (IDMODALITA)
  on update restrict
  on delete restrict;

alter table ORDINE
  add constraint FK_ORDINE_MODSPED foreign key (IDMODSPED)
  references MODALITACONSEGNA (IDMODALITA)
  on update restrict
  on delete restrict;

alter table ORDINE
  add constraint FK_ORDINE_CLIENTE foreign key (USERNAME)
  references CLIENTE (USERNAME)
  on update restrict
  on delete restrict;

alter table REPARTO
  add constraint FK_REPARTO_NEGOZIO foreign key (IDNEGOZIO)
  references NEGOZIO (IDNEGOZIO)
  on update restrict
  on delete restrict;
```

```
/* Inizializzazione */

insert into CATEGORIA(IDCATEGORIA,NOME) values (1,"Categoria1");
insert into CATEGORIA(IDCATEGORIA,NOME) values (2,"Categoria2");
insert into CATEGORIA(IDCATEGORIA,NOME) values (3,"Categoria3");
insert into CATEGORIA(IDCATEGORIA,NOME) values (4,"Categoria4");
insert into CATEGORIA(IDCATEGORIA,NOME) values (5,"Categoria5");
insert into CATEGORIA(IDCATEGORIA,NOME) values (6,"Categoria6");
insert into CATEGORIA(IDCATEGORIA,NOME) values (7,"Categoria7");
insert into CATEGORIA(IDCATEGORIA,NOME) values (8,"Categoria8");
insert into CATEGORIA(IDCATEGORIA,NOME) values (9,"Categoria9");
insert into CATEGORIA(IDCATEGORIA,NOME) values(10,"Categoria10");

insert into CLIENTE(USERNAME,PASSWORD,TIPO,NOME,COGNOME,
  CODFISCALE,INDIRIZZO,CITTA,PROVINCIA,CAP,
  EMAIL,TELEFONO,DOMSEGRETA,RISPSEGRETA)
values ("admin","pass","Amministratore","Luigi","Luongo",
  "ABCDEFGHIJ123456","Indirizzo","Città","CE","12345",
  "admin@admin.it","123-4567890","DomSegreta","RispSegreta");
```

Bibliografia e Sitografia

[1] Chuck Cavaness - *Programmare con Jakarta Struts* - HOEPLI

[2] Cay S. Horstmann, Gary Cornell - *Core Java 2 (settima edizione)* - Pearson, Prentice Hall

[3] C. Ghezzi e altri - *Ingegneria del Software* - Mondadori Informatica

[4] Simon Bennett e altri - *Introduzione a UML* - McGraw-Hill

[5] Luca Vetti Tagliati - *UML e ingegneria del software: dalla teoria alla pratica* - HOPS

[6] P. Atzeni e altri - *Basi di dati - Modelli e linguaggi di interrogazione* - McGraw-Hill

[7] Phill Hanna - *JSP - La guida completa* - McGraw-Hill

[8] David Harms - *JSP, Servlet e MySQL* - McGraw-Hill

[9] *The Jakarta Site* - `http://Jakarta.apache.org`

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

[11] *Exadel* - `http://www.exadel.com`

[12] *MySQL* - `http://www.mysql.com`

[13] *Java* - `http://www.java.com`

[14] *Java Italian Portal* - `http://www.javaportal.it`

[15] *JavaStaff* - `http://www.javastaff.com`

[16] *MokaByte* - `http://www.mokabyte.it`

[17] *HTML.it* - `www.html.it`

[18] *Sybase* - `http://www.sybase.com`

[19] *Struts* - `http://struts.apache.org`

[20] *TomCat* - `http://tomcat.apache.org`

[21] *Tiles* - `http://tiles.apache.org`

[22] *Validator* - `http://Jakarta.apache.org/commons/validator`

[23] *SSLext* - `http://sslext.sourceforge.net`

[24] *Log4j* - `http://logging.apache.org/log4j`