



**MyTalk**

## Definizione di prodotto

---

### Informazioni sul documento

---

<b>Nome file:</b>	definizione_di_prodotto.1.0.pdf
<b>Versione:</b>	1.0
<b>Data creazione:</b>	2013-03-04
<b>Data ultima modifica:</b>	2013-03-19
<b>Stato:</b>	Approvato
<b>Uso:</b>	Esterno
<b>Lista di distribuzione:</b>	Prof. Tullio Vardanega Prof. Riccardo Cardin Dott. Gregorio Piccoli Team SoftwareSynthesis
<b>Redattori:</b>	Andrea Rizzi Elena Zecchinato Marco Schivo Stefano Farronato
<b>Approvato da:</b>	Diego Beraldin
<b>Verificatori:</b>	Andrea Meneghinello

---

## Storia delle modifiche

Versione	Descrizione intervento	Membro	Ruolo	Data
1.0	Approvazione documento	Diego Beraldin	Responsabile	2013-03-09
0.10	Correzioni errori rilevati da verificatori	Schivo Marco	Progettista	2013-03-08
0.9	Verifica totale documento	Andrea Meneghinello	Verificatore	2013-03-08
0.8	Descritto classi di abook.authentication, server.authentication.servlet, server.connection	Elena Zecchinato	Progettista	2013-03-07
0.7	Descritto PresenterMediator, CallHistoryPanelPresenter, SearchResultPanelPresenter, ToolsPanelPresenter, AccountSettingsPanelPresenter	Schivo Marco	Progettista	2013-03-07
0.7	Descritto AddressBookPanelPresenter, GroupPanelPresenter, LoginPanelPresenter	Schivo Marco	Progettista	2013-03-06
0.6	Descritto classi di server.abook.servlet, server.call, server.message	Elena Zecchinato	Progettista	2013-03-06
0.5	Descritto RegisterPanelPresenter, CommunicationPanelPresenter, ContactPanelPresenter, MainPanelPresenter	Stefano Farronato	Progettista	2013-03-05
0.4	Descritte classi package server.dao e server.abook	Andrea Rizzi	Progettista	2013-03-05
0.3	Inizio descrizione parte client-presenter	Stefano Farronato	Progettista	2013-03-04
0.2	Stesura sezione “Standard di progetto” ed inizio descrizione server	Andrea Rizzi	Progettista	2013-03-04
0.1	Creazione del documento e stesura delle sezioni “Introduzione” e “Riferimenti”	Elena Zecchinato	Progettista	2013-03-04



## Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	Scopo del prodotto . . . . .	1
1.2	Scopo del documento . . . . .	1
1.3	Glossario . . . . .	1
1.4	Convenzioni di scrittura . . . . .	1
<b>2</b>	<b>Riferimenti</b>	<b>2</b>
2.1	Normativi . . . . .	2
2.2	Informativi . . . . .	2
<b>3</b>	<b>Standard di progetto</b>	<b>3</b>
3.1	Standard di progettazione architetturale . . . . .	3
3.2	Standard di documentazione del codice . . . . .	3
3.3	Standard di denominazione di entità e relazioni . . . . .	3
3.4	Standard di programmazione . . . . .	3
3.5	Strumenti di lavoro . . . . .	3
<b>4</b>	<b>Specifica sotto-architettura sever</b>	<b>4</b>
4.1	Package org.softwaresynthesis.mytalk.server.dao . . . . .	4
4.1.1	HibernateUtil . . . . .	4
4.1.2	UserDataDAO . . . . .	4
4.1.3	GroupDAO . . . . .	8
4.1.4	AddressBookEntryDAO . . . . .	11
4.1.5	CallDAO . . . . .	14
4.1.6	CallListDAO . . . . .	17
4.1.7	MessageDAO . . . . .	19
4.2	Package org.softwaresynthesis.mytalk.server.abook . . . . .	22
4.2.1	IUserData . . . . .	22
4.2.2	IGroup . . . . .	23
4.2.3	IAddressBookEntry . . . . .	24
4.2.4	UserData . . . . .	24
4.2.5	Group . . . . .	27
4.2.6	AddressBookEntry . . . . .	27
4.3	Package org.softwaresynthesis.mytalk.server.abook.servlet . . . . .	28
4.3.1	AddressBookDoAddContactServlet . . . . .	28
4.3.2	AddressBookDoRemoveContactServlet . . . . .	30
4.3.3	AddressBookDoCreateGroupServlet . . . . .	31
4.3.4	AddressBookDoDeleteGroupServlet . . . . .	33
4.3.5	AddressBookDoInsertInGroupServlet . . . . .	34
4.3.6	AddressBookDoRemoveFromGroupServlet . . . . .	36
4.3.7	AddressBookDoBlockServlet . . . . .	38
4.3.8	AddressBookDoUnblockServlet . . . . .	39
4.3.9	AddressBookDoSearchServlet . . . . .	41
4.3.10	AddressBookGetContactsServlet . . . . .	43
4.3.11	AddressBookGetGroupsServlet . . . . .	45
4.4	Package org.softwaresynthesis.mytalk.server.call . . . . .	47
4.4.1	ICall . . . . .	47
4.4.2	Call . . . . .	47
4.4.3	ICallList . . . . .	48
4.4.4	CallList . . . . .	49
4.5	Package org.softwaresynthesis.mytalk.server.call.servlet . . . . .	50

4.5.1	DownloadCallHistoryServlet . . . . .	50
4.6	Package org.softwaresynthesis.mytalk.server.message . . . . .	51
4.6.1	IMessage . . . . .	51
4.6.2	Message . . . . .	52
4.7	Package org.softwaresynthesis.mytalk.server.message.servlet . . . . .	53
4.7.1	InsertMessageServlet . . . . .	53
4.7.2	DeleteMessageServlet . . . . .	54
4.7.3	UpdateStatusMessageServlet . . . . .	56
4.7.4	DownloadMessageListServlet . . . . .	57
4.8	Package org.softwaresynthesis.mytalk.server.authentication . . . . .	58
4.8.1	ISecurityStrategy . . . . .	58
4.8.2	AESAlgorithm . . . . .	59
4.8.3	PrincipalImpl . . . . .	60
4.8.4	IAuthenticationData . . . . .	60
4.8.5	AuthenticationData . . . . .	61
4.8.6	AuthenticationModule . . . . .	61
4.8.7	CredentialLoader . . . . .	63
4.9	Package org.softwaresynthesis.mytalk.server.authentication.servlet . . . . .	64
4.9.1	LogoutServlet . . . . .	64
4.9.2	LoginServlet . . . . .	65
4.9.3	RegisterServlet . . . . .	66
4.10	Package org.softwaresynthesis.mytalk.server.connection . . . . .	68
4.10.1	PushInbound . . . . .	68
4.10.2	ChanelServlet . . . . .	70
<b>5</b>	<b>Specifica sotto-architettura clientpresenter</b>	<b>72</b>
5.1	Package org.softwaresynthesis.mytalk.clientpresenter.guicontrol . . . . .	73
5.1.1	AddressBookPanelPresenter . . . . .	73
5.1.2	LoginPanelPresenter . . . . .	78
5.1.3	RegisterPanelPresenter . . . . .	81
5.1.4	CommunicationPanelPresenter . . . . .	83
5.1.5	ContactPanelPresenter . . . . .	85
5.1.6	MainPanelPresenter . . . . .	87
5.1.7	PresenterMediator . . . . .	87
5.1.8	MessagePanelPresenter . . . . .	91
5.1.9	CallHistoryPanelPresenter . . . . .	92
5.1.10	SearchresultPanelPresenter . . . . .	93
5.1.11	GroupPanelPresenter . . . . .	94
5.1.12	ToolsPanelPresenter . . . . .	94
5.1.13	AccountSettingsPanelPresenter . . . . .	95
5.2	Package org.softwaresynthesis.mytalk.clientpresenter.kernel . . . . .	97
5.2.1	CommunicationCenter . . . . .	97
<b>6</b>	<b>Specifica sotto-architettura clientview</b>	<b>100</b>

## 1 Introduzione

### 1.1 Scopo del prodotto

Con il progetto “MyTalk” si intende un sistema software di comunicazione tra utenti mediante browser senza la necessità di installazione di plugin e/o software esterni. L’utente avrà la possibilità di interagire con un altro utente tramite una comunicazione audio - audio/video - testuale e, inoltre, ottenere delle statistiche sull’attività in tempo reale.

### 1.2 Scopo del documento

Il presente documento presenta una descrizione dettagliata dell’architettura del sistema software destinata alla realizzazione del prodotto MyTalk coerentemente con la progettazione ad alto livello descritta nell’allegato *specifica\_tecnica.2.0.pdf*.

A tal fine si riporta per ognuno dei componenti definiti nel documento di specifica tecnica una descrizione delle classi in termini di operazioni disponibili, proprietà, responsabilità e collaborazioni. Il contenuto del presente documento ha inoltre valore vincolante per i programmatori, pertanto avranno l’obbligo di attenersi alle disposizioni in esso contenute senza alcuna possibilità di deroga.

### 1.3 Glossario

Al fine di evitare incomprensioni dovute all’uso di termini tecnici nei documenti, viene redatto e allegato il documento *glossario.3.0.pdf* dove vengono definiti e descritti tutti i termini marcati con una sottolineatura.

### 1.4 Convenzioni di scrittura

Al fine di rendere quanto più agevole possibile la consultazione del documento da parte dei programmatori e del committente, è stata adottata una serie di accorgimenti sia a livello di riferimenti sulla nomenclatura delle classi sia a livello cromatico per campi dati e metodi. Tali norme possono essere consultate in dettaglio nel documento *norme\_di\_progetto.3.0.pdf* allegato.

## 2 Riferimenti

### 2.1 Normativi

*piano\_di\_qualifica.3.0.pdf* allegato.

*norme\_di\_progetto.3.0.pdf* allegato.

*specifica\_tecnica.2.0.pdf* allegato

### 2.2 Informativi

Capitolato d'appalto: MyTalk, v1.0, redatto e rilasciato dal proponente Zucchetti s.r.l. reperibile all'indirizzo <http://www.math.unipd.it/~tullio/IS-1/2012/Progetto/C1.pdf>;

testo di consultazione: *Software Engineering (8th edition) Ian Sommerville, Pearson Education / Addison Wesley*;

manuale all'utilizzo dei design patterns: *Design Patterns, Elementi per il riutilizzo di software a oggetti – (1/Ed. italiana) Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides, Pearson Education*;

*glossario.3.0.pdf* allegato.

### 3 Standard di progetto

#### 3.1 Standard di progettazione architetturale

Lo sviluppo del progetto ha seguito le regole architettureali specificate nel documento *norme\_di\_progetto.3.0.pdf* allegato.

#### 3.2 Standard di documentazione del codice

Le regole che definiscono la documentazione del codice relativo al funzionamento del prodotto sono specificate nel documento *norme\_di\_progetto.3.0.pdf* allegato.

#### 3.3 Standard di denominazione di entità e relazioni

Le convenzioni relative alla denominazione delle entità e le relative relazioni sono specificate nel documento *norme\_di\_progetto.3.0.pdf* allegato.

#### 3.4 Standard di programmazione

Le regole relative agli standard di programmazione sono enunciate nel documento *norme\_di\_progetto.3.0.pdf* allegato.

#### 3.5 Strumenti di lavoro

Gli strumenti utilizzati per la stesura e lo sviluppo sono specificati nei documenti *norme\_di\_progetto.3.0.pdf* e *piano\_di\_qualifica.3.0.pdf* allegati.

## 4 Specifica sotto-architettura sever

### 4.1 Package org.softwaresynthesis.mytalk.server.dao

#### 4.1.1 HibernateUtil

##### Funzione

Inizializza un'unica *factory* per le sessioni, utilizzate da *Hibernate*, per comunicare con il database.

##### Relazioni d'uso

- `org.hibernate.SessionFactory`: necessaria per interrogare il database.
- `org.hibernate.cfg.Configuration`: definisce i parametri necessari per la connessione con il database. Inoltre definisce i *mapping* necessari tra le classi *transfer object* e le relative tabelle nel database.

##### Attributi

- - `instance`: `HibernateUtil`  
Attributo usato per implementare il *pattern* singleton. Tale istanza verrà inizializzata tramite il metodo statico `getInstance()`, assicurando che l'attributo non sia già stato inizializzato in precedenza.
- - `sessionFactory`: `SessionFactory`  
Attributo contenente la *factory* delle sessioni verso il database.

##### Metodi

- `HibernateUtil()`  
Costruttore privato della classe, definito *private* in correlazione all'applicazione del *pattern* singleton.
- + `getInstance(): HibernateUtil`  
Metodo pubblico che ritorna l'istanza `instance`. Il metodo controlla se `instance` è già stata inizializzata, nel caso in cui non lo sia il metodo dovrà generare un'istanza di `HibernateUtil` richiamando il costruttore privato `HibernateUtil()` e assegnare il valore ritornato all'attributo `instance`. Il programma termina restituendo `instance`.
- + `getSessionFactory(): SessionFactory`  
Metodo che ritorna l'attributo `sessionFactory`.

#### 4.1.2 UserDataDAO

##### Funzione

Classe che implementa il *pattern* DAO. Definisce le procedure d'inserimento, eliminazione ed aggiornamento di *entry* inerenti alla tabella `UserData`. La classe inoltre fornisce metodi per interrogare il database ed ottenere i dati utente con i quali costruire istanze di oggetti di tipo `abook.IUserData`.



## Relazioni d'uso

- `java.util.List`: usata per memorizzare e ritornare i dati restituiti dalle interrogazioni a database.
- `org.hibernate.Query`: classe rappresentante le *query* d'interrogazione verso il database
- `org.hibernate.Session`: classe che rappresenta la sessione di "lavoro" verso il database.
- `org.hibernate.SessionFactory`: classe avente compito di instanziare nuove sessioni.
- `org.hibernate.Transaction`: classe che rappresenta una transazione del database.
- `abook.IUserData`: interfaccia del package *abook* usata per definire un generico utente. La classe può ritornare istanze di tipo `abook.IUserData` come risposta a delle *query* d'interrogazione al database.

## Attributi

Nessun attributo evidenziato.

## Metodi

+ `insert(user: IUserData): boolean`

Metodo usato per inserire un utente registrato nel sistema, riceve come parametro un istanza di un oggetto `abook.IUserData`. Il metodo dovrà eseguire nell'ordine le seguenti operazioni:

- 1) ottenere dalla classe `HibernateUtil` l'istanza tramite il metodo statico `getInstance()`;
- 2) ottenere dall'istanza di tipo `HibernateUtil` (ottenuta al punto precedente), un oggetto di tipo `org.hibernate.SessionFactory`, tramite il metodo `getSessionFactory()`;
- 3) aprire la sessione ottenuta al punto precedente;
- 4) avviare una transizione a partire dall'istanza `org.hibernate.SessionFactory` del punto 2. Tale istanza va memorizzata in un opportuno attributo (interno al metodo stesso) di tipo `org.hibernate.Transaction`;
- 5) eseguire il comando *save* a partire dalla sessione aperta e passando come parametro l'attributo `user`.
- 6) eseguire il comando *commit* per confermare l'operazione del punto precedente.

Poiché l'iter presentato potrà causare il lancio di un'eccezione, il codice dovrà essere gestito in un blocco *try-catch*. Nel caso incorra tale evento, il flusso di processo dovrà ristabilire la situazione iniziale richiamando il metodo `rollback()` dell'istanza di tipo `org.hibernate.Transaction` creata al punto 4 dell'iter. In ogni caso il metodo deve terminare chiudendo la sessione aperta al punto 2.

+ `delete(user: IUserData): boolean`

Metodo usato per eliminare un utente registrato nel sistema, riceve come parametro un istanza di un oggetto `abook.IUserData`. Il metodo dovrà eseguire nell'ordine le seguenti operazioni:

- 1) ottenere dalla classe `HibernateUtil` l'istanza tramite il metodo statico `getInstance()`;
- 2) ottenere dall'istanza di tipo `HibernateUtil` (ottenuta al punto precedente), un oggetto di tipo `org.hibernate.SessionFactory`, tramite il metodo `getSessionFactory()`;
- 3) aprire la sessione ottenuta al punto precedente;

- 4) avviare una transizione a partire dall'istanza `org.hibernate.SessionFactory` del punto 2. Tale istanza va memorizzato in un opportuno attributo (interno al metodo stesso) di tipo `org.hibernate.Transaction`;
- 5) eseguire il comando *delete* a partire dalla sessione aperta e passando come parametro l'attributo `user`.
- 6) eseguire il comando *commit* per confermare l'operazione del punto precedente.

Poiché l'iter presentato può causare il lancio di un'eccezione, il codice dovrà essere gestito in un blocco *try-catch*. Nel caso incorra tale evento, il flusso di processo dovrà ristabilire la situazione iniziale richiamando il metodo `rollback()` dell'istanza di tipo `org.hibernate.Transaction` creata al punto 4 dell'iter. In ogni caso il metodo deve terminare chiudendo la sessione aperta al punto 2.

+ `update(user: IUserData): boolean`

Metodo usato per modificare i dati di un utente registrato nel sistema, riceve come parametro un'istanza di un oggetto `abook.IUserData`. Il metodo dovrà eseguire nell'ordine le seguenti operazioni:

- 1) ottenere dalla classe `HibernateUtil` l'istanza tramite il metodo statico `getInstance()`;
- 2) ottenere dall'istanza di tipo `HibernateUtil` (ottenuta al punto precedente), un oggetto di tipo `org.hibernate.SessionFactory`, tramite il metodo `getSessionFactory()`;
- 3) aprire la sessione ottenuta al punto precedente;
- 4) avviare una transizione a partire dall'istanza `org.hibernate.SessionFactory` del punto 2. Tale istanza va memorizzato in un opportuno attributo (interno al metodo stesso) di tipo `org.hibernate.Transaction`;
- 5) eseguire il comando *update* a partire dalla sessione aperta e passando come parametro l'attributo `user`.
- 6) eseguire il comando *commit* per confermare l'operazione del punto precedente.

Poiché l'iter presentato può causare il lancio di un'eccezione, il codice dovrà essere gestito in un blocco *try-catch*. Nel caso incorra il lancio tale evento, il flusso di processo dovrà ristabilire la situazione iniziale richiamando il metodo `rollback()` dell'istanza di tipo `org.hibernate.Transaction` creata al punto 4 dell'iter. In ogni caso il metodo deve terminare chiudendo la sessione aperta al punto 2.

+ `getByNameAndSurname(name: String, surname: String): List<IUserData>`

Metodo usato per effettuare un'interrogazione al database atta ad ottenere la lista degli utenti aventi un determinato nome e cognome, riceve come parametri il nome e il cognome da ricercare. Il metodo dovrà eseguire nell'ordine le seguenti operazioni:

- 1) ottenere dalla classe `HibernateUtil` l'istanza tramite il metodo statico `getInstance()`;
- 2) ottenere dall'istanza di tipo `HibernateUtil` (ottenuta al punto precedente), un oggetto di tipo `org.hibernate.SessionFactory`, tramite il metodo `getSessionFactory()`;
- 3) aprire la sessione ottenuta al punto precedente;
- 4) eseguire il comando *createQuery()* a partire dall'istanza che definisce la sessione, passando come parametro una stringa rappresentante la *query* d'interrogazione al database (la *query* deve restituire dati validi ad essere passati a tale metodo). Una prima analisi evidenzia che la *query* corretta è:

```
from UserData as u where u.name = :name or u.surname = :surname
```

- 5) eseguire una serie di comandi di tipo `setString` per ogni elemento della *query* d'interrogazione (dove per elemento si intende ogni valore di filtraggio presente nelle clausole *where* delle *query* HQL, segnato come `:nome_variabile`).

- 6) eseguire il comando *commit* per l'interrogare il database ed ottenere la lista di valori da ritornare.

Il metodo deve terminare chiudendo la sessione aperta al punto 2.

+ **getByEmail(mail: String): IUserData**

Metodo usato per eseguire un interrogazione al database atta ad ottenere l'utente avente un determinato indirizzo e-mail, esso riceve come parametro l'indirizzo e-mail da ricercare. Il metodo dovrà eseguire nell'ordine le seguenti operazioni:

- 1) ottenere dalla classe `HibernateUtil` l'istanza tramite il metodo statico `getInstance()`;
- 2) ottenere dall'istanza di tipo `HibernateUtil` (ottenuta al punto precedente), un oggetto di tipo `org.hibernate.SessionFactory`, tramite il metodo `getSessionFactory()`;
- 3) aprire la sessione ottenuta al punto precedente;
- 4) eseguire il comando `createQuery()` a partire dall'istanza che definisce la sessione, passando come parametro una stringa rappresentante la *query* d'interrogazione al database (la *query* deve restituire dati validi ad essere passati a tale metodo). Una prima analisi evidenzia che la *query* corretta è:

```
from UserData as u where u.mail = :mail
```

- 5) eseguire una serie di comandi di tipo `setString` per ogni elemento della *query* d'interrogazione (per elemento si intende ogni valore di filtraggio presente nelle clausole *where* delle *query* HQL, segnato come `:nome_variabibile`).
- 6) eseguire il comando *commit* per eseguire l'interrogazione a database ed ottenere la lista di valori da ritornare.

Il metodo deve terminare chiudendo la sessione aperta al punto 2.

+ **searchGeneric(value: String): List<IUserData>**

Metodo usato per eseguire un interrogazione generica al database atta ad ottenere una lista di utenti aventi: o un determinato indirizzo e-mail, o un determinato nome o un determinato cognome; riceve come parametro la stringa da ricercare e dovrà eseguire nell'ordine le seguenti operazioni:

- 1) ottenere dalla classe `HibernateUtil` l'istanza tramite il metodo statico `getInstance()`;
- 2) ottenere dall'istanza di tipo `HibernateUtil` (ottenuta al punto precedente), un oggetto di tipo `org.hibernate.SessionFactory`, tramite il metodo `getSessionFactory()`;
- 3) aprire la sessione ottenuta al punto precedente;
- 4) eseguire il comando `createQuery()` a partire dall'istanza che definisce la sessione, passando come parametro una stringa rappresentante la *query* d'interrogazione al database (la *query* deve restituire dati validi ad essere passati a tale metodo). Una prima analisi evidenzia che la *query* corretta è:

```
from UserData as u where u.mail like :mail or u.name like :name or
u.surname like :surname
```

- 5) eseguire una serie di comandi di tipo `setString` per ogni elemento della *query* d'interrogazione (per elemento si intende ogni valore di filtraggio presente nelle clausole *where* delle *query* HQL, segnato come `:nome_variabibile`).
- 6) eseguire il comando *commit* per eseguire l'interrogazione a database ed ottenere la lista di valori da ritornare.

Il metodo deve terminare chiudendo la sessione aperta al punto 2.

### 4.1.3 GroupDAO

#### Funzione

Classe che implementa il pattern DAO. Definisce le procedure d'inserimento, eliminazione ed aggiornamento di entry inerenti alla tabella Group. La classe inoltre fornisce metodi per interrogare il database ed ottenere i dati utente con i quali costruire istanze di oggetti di tipo `abook.IGroup`.

#### Relazioni d'uso

- `java.util.List`:
- `org.hibernate.Query`:
- `org.hibernate.Session`:
- `org.hibernate.SessionFactory`:
- `org.hibernate.Transaction`:
- `abook.IGroup`:

#### Attributi

Nessun attributo evidenziato.

#### Metodi

##### + `delete(group: IGroup): boolean`

Metodo usato per cancellare un gruppo dal database. Per gruppo si intende un qualsiasi oggetto implementante l'interfaccia `abook.IGroup`. Il metodo riceve come parametro d'ingresso un oggetto `abook.IGroup` e restituisce un valore booleano pari a *true* se l'operazione è andata a buon fine, *false* altrimenti. Il flusso principale inizia definendo le seguenti variabili:

- viene inizializzata una variabile di tipo boolean nominata *flag*, a *false*;
- si crea un'istanza di `HibernateUtil` *util*;
- si crea un'istanza di `Session` *session*;
- si crea un'istanza di `SessionFactory` *factory*;
- si crea un'istanza di `Transaction` *transaction*;

Quindi nel metodo deve essere definito un blocco try catch finally:

- nel blocco try:
  - *util* deve essere inizializzato mediante la chiamata `getInstance()` di `HibernateUtil`;
  - quindi a partire da *util* si inizializza *factory* mediante una chiamata `getFactory()`;
  - analogamente a quanto fatto in precedenza, si inizializza *session* tramite una chiamata `openSession()` eseguita a partire da *factory*;
  - *transaction* viene inizializzata con una chiamata al metodo `beginTransaction()` eseguita da *session*;
  - quindi si è pronti per eseguire l'istruzione di delete a partire da *session* e passando come parametro *group*;
  - si esegue il commit;
  - si pone *flag* a *true*;

- nel blocco `catch`: si effettua un `rollback` a partire da `transaction`;
- nel blocco `finally`: se `session` è diverso da `null`, si chiude mediante una chiamata di tipo `close()`;

Infine il metodo termina ritornando il valore di `flag`;

#### + `update(group: IGroup): boolean`

Metodo usato per modificare un gruppo dal database. Il metodo riceve come parametro d'ingresso un oggetto `abook.IGroup` e restituisce un valore booleano pari a `true` se l'operazione è andata a buon fine, `false` altrimenti. Il flusso principale inizia definendo le seguenti variabili:

- viene inizializzata una variabile di tipo boolean nominata `flag`, a `false`;
- si crea un'istanza di `HibernateUtil` `util`;
- si crea un'istanza di `Session` `session`;
- si crea un'istanza di `SessionFactory` `factory`;
- si crea un'istanza di `Transaction` `transaction`;

Quindi nel metodo deve essere definito un blocco `try catch finally`:

- nel blocco `try`:
  - `util` deve essere inizializzato mediante la chiamata `getInstance()` di `HibernateUtil`;
  - quindi a partire da `util` si inizializza `factory` mediante una chiamata `getFactory()`;
  - analogamente a quanto fatto in precedenza, si inizializza `session` tramite una chiamata `openSession()` eseguita a partire da `factory`;
  - `transaction` viene inizializzata con una chiamata al metodo `beginTransaction()` eseguita da `session`;
  - quindi si è pronti per eseguire l'istruzione di `update` a partire da `session` e passando come parametro `group`;
  - si esegue il `commit`;
  - si pone `flag` a `true`;
- nel blocco `catch`: si effettua un `rollback` a partire da `transaction`;
- nel blocco `finally`: se `session` è diverso da `null`, si chiude mediante una chiamata di tipo `close()`;

Infine il metodo termina ritornando il valore di `flag`;

#### + `insert(group: IGroup): boolean`

Metodo usato per inserire un gruppo nel database, riceve come parametro d'ingresso un oggetto `abook.IGroup` e restituisce un valore booleano pari a `true` se l'operazione è andata a buon fine, `false` altrimenti. Il flusso principale inizia definendo le seguenti variabili:

- viene inizializzata una variabile di tipo boolean nominata `flag`, a `false`;
- si crea un'istanza di `HibernateUtil` `util`;
- si crea un'istanza di `Session` `session`;
- si crea un'istanza di `SessionFactory` `factory`;
- si crea un'istanza di `Transaction` `transaction`;

Quindi nel metodo deve essere definito un blocco `try-catch finally`:

- nel blocco `try`:

- util deve essere inizializzato mediante la chiamata `getInstance()` di `HibernateUtil`;
- quindi a partire da util si inizializza factory mediante una chiamata `getFactory()`;
- analogamente a quanto fatto in precedenza, si inizializza *session* tramite una chiamata `openSession()` eseguita a partire da *factory*;
- transaction viene inizializzata con una chiamata a metodo `beginTransaction()` eseguita da *session*;
- quindi si è pronti per eseguire l'istruzione di *save* a partire da *session* e passando come parametro *group*;
- si esegue il commit;
- si pone *flag* a *true*;
- nel blocco catch: si effettua un *rollback* a partire da *transaction*;
- nel blocco finally: se *session* è diverso da *null*, si chiude mediante una chiamata di tipo `close()`;

Infine il metodo termina ritornando il valore di *flag*;

+ `getByID(identifier: long): IGroup`

Metodo usato per interrogare il database ed ottenere un istanza di `IGroup` avente come identificativo, l'id del gruppo ricevuto come parametro d'ingresso. Il flusso principale inizia definendo le seguenti variabili:

- si crea un istanza di `HibernateUtil` *util*;
- si crea un istanza di `List<IGroup>` *groups*;
- si crea un istanza di `Query` *query*;
- si crea una stringa *hqlQuery* impostata per default a:  
"from Group as g where g.id = :id"
- si crea un istanza di `Session` *session*;
- si crea un istanza di `SessionFactory` *factory*;
- si crea un istanza di `Transaction` *transaction*;

Quindi nel metodo deve essere definito un blocco try catch finally:

- nel blocco try:
  - util deve essere inizializzato mediante la chiamata `getInstance()` di `HibernateUtil`;
  - quindi a partire da *util* si inizializza *factory* mediante una chiamata `getFactory()`;
  - analogamente a quanto fatto in precedenza, si inizializza *session* tramite una chiamata `openSession()` eseguita a partire da *factory*;
  - transaction viene inizializzata con una chiamata a metodo `beginTransaction()` eseguita da *session*;
  - query viene inizializzata mediante una chiamata `createQuery`, chiamata da *session* e a cui deve essere passata come parametro, la stringa *hqlQuery*;
  - quindi si è pronti per eseguire query ed ottenere la lista di `IGroup` desiderata. Tale operazione si effettua richiamando `query.list()`;
  - si esegue il *commit*;
- nel blocco catch: si effettua un *rollback* a partire da *transaction*;
- nel blocco finally: se *session* è diverso da *null*, si chiude mediante una chiamata di tipo `close()`;

Il programma termina controllando se *groups* è diversa da *null* e contiene almeno un elemento. Nel caso il programma termina ritornando il primo contatto presente nella lista. Altrimenti termina ritornando *null*.

+ `getByOwner(owner: long): List<IGroup>`

Metodo usato per interrogare il database ed ottenere una lista di istanze di *IGroup* aventi come possessore dei gruppi l'utente con id uguale al valore *owner* passato come parametro. Il flusso principale inizia definendo le seguenti variabili:

- si crea un'istanza di *HibernateUtil* *util*;
- si crea un'istanza di *List<IGroup>* *groups*;
- si crea un'istanza di *Query* *query*;
- si crea una stringa *hqlQuery* impostata per default a:  
"from Group as g where g.owner = :owner"
- si crea un'istanza di *Session* *session*;
- si crea un'istanza di *SessionFactory* *factory*;
- si crea un'istanza di *Transaction* *transaction*;

Quindi nel metodo deve essere definito un blocco *try-catch* finally:

- nel blocco *try*:
  - *util* deve essere inizializzato mediante la chiamata `getInstance()` di *HibernateUtil*;
  - quindi a partire da *util* si inizializza *factory* mediante una chiamata `getFactory()`;
  - analogamente a quanto fatto in precedenza, si inizializza *session* tramite una chiamata `openSession()` eseguita a partire da *factory*;
  - *transaction* viene inizializzata con una chiamata a metodo `beginTransaction()` eseguita da *session*;
  - *query* viene inizializzata mediante una chiamata `createQuery`, chiamata da *session* e a cui deve essere passata come parametro, la stringa *hqlQuery*;
  - quindi si è pronti per eseguire *query* ed ottenere la lista di *IGroup* desiderata. Tale operazione si effettua richiamando `query.list()`;
  - si esegue il commit;
- nel blocco *catch*: si effettua un *rollback* a partire da *transaction*;
- nel blocco *finally*: se *session* è diverso da *null*, si chiude mediante una chiamata di tipo `close()`;

Il programma termina restituendo *groups*.

#### 4.1.4 AddressBookEntryDAO

##### Funzione

Classe che implementa il pattern DAO. Definisce le procedure d'inserimento, eliminazione ed aggiornamento di entry inerenti alla tabella *AddressBookEntry*.

##### Relazioni d'uso

- `org.hibernate.Session`;
- `org.hibernate.SessionFactory`;
- `org.hibernate.Transaction`;
- `abook.IAddressBookEntry`;



## Attributi

Nessun attributo evidenziato.

## Metodi

+ `delete(entry: IAddressBookEntry): boolean`

Metodo usato per cancellare una *entry* dal database. Per *entry* si intende un qualsiasi oggetto implementante l'interfaccia `IAddressBookEntry`. Il metodo riceve come parametro d'ingresso un oggetto `abook.IAddressBookEntry` e restituisce un valore booleano *true* se l'operazione è andata a buon fine, *false* altrimenti. Il flusso principale inizia definendo le seguenti variabili:

- viene inizializzata una variabile di tipo *boolean* nominata *flag*, a *false*;
- si crea un'istanza di `HibernateUtil` *util*;
- si crea un'istanza di `Session` *session*;
- si crea un'istanza di `SessionFactory` *factory*;
- si crea un'istanza di `Transaction` *transaction*;

Quindi nel metodo deve essere definito un blocco *try-catch finally*:

- nel blocco *try*:
  - *util* deve essere inizializzato mediante la chiamata `getInstance()` di `HibernateUtil`;
  - quindi a partire da *util* si inizializza *factory* mediante una chiamata `getFactory()`;
  - analogamente a quanto fatto in precedenza, si inizializza *session* tramite una chiamata `openSession()` eseguita a partire da *factory*;
  - *transaction* viene inizializzata con una chiamata al metodo `beginTransaction()` eseguita da *session*;
  - quindi si è pronti per eseguire l'istruzione di inglese *delete* a partire da *session* e passando come parametro *entry*;
  - si esegue il *commit*;
  - si pone *flag* a *true*;
- nel blocco *catch*: si effettua un *rollback* a partire da *transaction*;
- nel blocco *finally*: se *session* è diverso da *null*, si chiude mediante una chiamata di tipo `close()`;

Infine il metodo termina ritornando il valore di *flag*;

+ `update(entry: IAddressBookEntry): boolean`

Metodo usato per modificare una *entry* presente nel database. Il metodo riceve come parametro d'ingresso un oggetto `abook.IAddressBookEntry` e restituisce un valore booleano *true* se l'operazione è andata a buon fine, *false* altrimenti. Il flusso principale inizia definendo le seguenti variabili:

- viene inizializzata una variabile di tipo *boolean* nominata *flag*, a *false*;
- si crea un'istanza di `HibernateUtil` *util*;
- si crea un'istanza di `Session` *session*;
- si crea un'istanza di `SessionFactory` *factory*;
- si crea un'istanza di `Transaction` *transaction*;

Quindi nel metodo deve essere definito un blocco *try-catch finally*:



- nel blocco *try*:
  - *util* deve essere inizializzato mediante la chiamata `getInstance()` di `HibernateUtil`;
  - quindi a partire da *util* si inizializza *factory* mediante una chiamata `getFactory()`;
  - analogamente a quanto fatto in precedenza, si inizializza *session* tramite una chiamata `openSession()` eseguita a partire da *factory*;
  - *transaction* viene inizializzata con una chiamata a metodo `beginTransaction()` eseguita da *session*;
  - quindi si è pronti per eseguire l'istruzione di *update* a partire da *session* e passando come parametro *entry*;
  - si esegue il *commit*;
  - si pone *flag* a *true*;
- nel blocco *catch*: si effettua un *rollback* a partire da *transaction*;
- nel blocco *finally*: se *session* è diverso da *null*, si chiude mediante una chiamata di tipo `close()`;

Infine il metodo termina ritornando il valore di *flag*;

+ `insert(entry: IAddressBookEntry): boolean`

Metodo usato per inserire una *entry* nel database. Il metodo riceve come parametro d'ingresso un oggetto `abook.IAddressBookEntry` e restituisce un valore booleano *true* se l'operazione è andata a buon fine, *false* altrimenti. Il flusso principale inizia definendo le seguenti variabili:

- viene inizializzata una variabile di tipo *boolean* nominata *flag*, a *false*;
- si crea un'istanza di `HibernateUtil` *util*;
- si crea un'istanza di `Session` *session*;
- si crea un'istanza di `SessionFactory` *factory*;
- si crea un'istanza di `Transaction` *transaction*;

Quindi nel metodo deve essere definito un blocco *try-catch finally*:

- nel blocco *try*:
  - *util* deve essere inizializzato mediante la chiamata `getInstance()` di `HibernateUtil`;
  - quindi a partire da *util* si inizializza *factory* mediante una chiamata `getFactory()`;
  - analogamente a quanto fatto in precedenza, si inizializza *session* tramite una chiamata `openSession()` eseguita a partire da *factory*;
  - *transaction* viene inizializzata con una chiamata a metodo `beginTransaction()` eseguita da *session*;
  - quindi si è pronti per eseguire l'istruzione di *save* a partire da *session* e passando come parametro *entry*;
  - si esegue il *commit*;
  - si pone *flag* a *true*;
- nel blocco *catch*: si effettua un *rollback* a partire da *transaction*;
- nel blocco *finally*: se *session* è diverso da *null*, si chiude mediante una chiamata di tipo `close()`;

Infine il metodo termina ritornando il valore di *flag*;

#### 4.1.5 CallDAO

##### Funzione

Classe che implementa il pattern DAO. Definisce le procedure d'inserimento, eliminazione ed aggiornamento di entry inerenti alla tabella Call. La classe inoltre fornisce metodi per interrogare il database ed ottenere la lista delle chiamate.

##### Relazioni d'uso

- `java.util.List`:
- `org.hibernate.Query`:
- `org.hibernate.Session`:
- `org.hibernate.SessionFactory`:
- `org.hibernate.Transaction`:
- `abook.ICall`:

##### Attributi

Nessun attributo evidenziato.

##### Metodi

+ `delete(call: ICall): boolean`

Metodo usato per cancellare una chiamata dal database. Il flusso principale inizia definendo le seguenti variabili:

- viene inizializzata una variabile di tipo boolean nominata *flag*, a *false*;
- si crea un'istanza di `HibernateUtil` *util*;
- si crea un'istanza di `Session` *session*;
- si crea un'istanza di `SessionFactory` *factory*;
- si crea un'istanza di `Transaction` *transaction*;

Quindi nel metodo deve essere definito un blocco `try catch finally`:

- nel blocco `try`:
  - *util* deve essere inizializzato mediante la chiamata `getInstance()` di `HibernateUtil`;
  - quindi a partire da *util* si inizializza *factory* mediante una chiamata `getFactory()`;
  - analogamente a quanto fatto in precedenza, si inizializza *session* tramite una chiamata `openSession()` eseguita a partire da *factory*;
  - *transaction* viene inizializzata con una chiamata al metodo `beginTransaction()` eseguita da *session*;
  - quindi si è pronti per eseguire l'istruzione di *delete* a partire da *session* e passando come parametro *call*;
  - si esegue il commit;
  - si pone *flag* a *true*;
- nel blocco `catch`: si effettua un rollback a partire da *transaction*;
- nel blocco `finally`: se *session* è diverso da *null*, si chiude mediante una chiamata di tipo `close()`;

Infine il metodo termina ritornando il valore di *flag*;

+ `update(call: ICall): boolean`

Metodo usato per modificare una chiamata presente nel database. Il metodo riceve come parametro d'ingresso un oggetto `call.Call` e restituisce un valore booleano pari a *true* se l'operazione è andata a buon fine, *false* altrimenti. Il flusso principale inizia definendo le seguenti variabili:

- viene inizializzata una variabile di tipo boolean nominata *flag*, a *false*;
- si crea un'istanza di `HibernateUtil` *util*;
- si crea un'istanza di `Session` *session*;
- si crea un'istanza di `SessionFactory` *factory*;
- si crea un'istanza di `Transaction` *transaction*;

Quindi nel metodo deve essere definito un blocco try catch finally:

- nel blocco try:
  - *util* deve essere inizializzato mediante la chiamata `getInstance()` di `HibernateUtil`;
  - quindi a partire da *util* si inizializza *factory* mediante una chiamata `getFactory()`;
  - analogamente a quanto fatto in precedenza, si inizializza *session* tramite una chiamata `openSession()` eseguita a partire da *factory*;
  - *transaction* viene inizializzata con una chiamata al metodo `beginTransaction()` eseguita da *session*;
  - quindi si è pronti per eseguire l'istruzione di *update* a partire da *session* e passando come parametro *call*;
  - si esegue il *commit*;
  - si pone *flag* a *true*;
- nel blocco catch: si effettua un *rollback* a partire da *transaction*;
- nel blocco finally: se *session* è diverso da null, si chiude mediante una chiamata di tipo `close()`;

Infine il metodo termina ritornando il valore di *flag*;

+ `insert(call: ICall): boolean`

Metodo usato per inserire una chiamata nel database, riceve come parametro d'ingresso un oggetto `call.ICall` e restituisce un valore booleano pari a *true* se l'operazione è andata a buon fine, *false* altrimenti. Il flusso principale inizia definendo le seguenti variabili:

- viene inizializzata una variabile di tipo boolean nominata *flag*, a *false*;
- si crea un'istanza di `HibernateUtil` *util*;
- si crea un'istanza di `Session` *session*;
- si crea un'istanza di `SessionFactory` *factory*;
- si crea un'istanza di `Transaction` *transaction*;

Quindi nel metodo deve essere definito un blocco try-catch finally:

- nel blocco try:
  - *util* deve essere inizializzato mediante la chiamata `getInstance()` di `HibernateUtil`;
  - quindi a partire da *util* si inizializza *factory* mediante una chiamata `getFactory()`;
  - analogamente a quanto fatto in precedenza, si inizializza *session* tramite una chiamata `openSession()` eseguita a partire da *factory*;

- transaction viene inizializzata con una chiamata a metodo `beginTransaction()` eseguita da *session*;
- quindi si è pronti per eseguire l'istruzione di *save* a partire da *session* e passando come parametro *group*;
- si esegue il commit;
- si pone *flag* a *true*;
- nel blocco catch: si effettua un *rollback* a partire da *transaction*;
- nel blocco finally: se *session* è diverso da *null*, si chiude mediante una chiamata di tipo `close()`;

Infine il metodo termina ritornando il valore di *flag*;

+ `getByID(identifier: long): ICall`

Metodo usato per interrogare il database ed ottenere un istanza di `ICall` avente come identificativo, l'id della chiamata ricevuta come parametro d'ingresso. Il flusso principale inizia definendo le seguenti variabili:

- si crea un istanza di `HibernateUtil` *util*;
- si crea un istanza di `List<ICall>` *calls*;
- si crea un istanza di `Query` *query*;
- si crea una stringa *hqlQuery* impostata per default a:  
"from call as c where c.mittente = :id or c.destinatario =:id"
- si crea un istanza di `Session` *session*;
- si crea un istanza di `SessionFactory` *factory*;
- si crea un istanza di `Transaction` *transaction*;

Quindi nel metodo deve essere definito un blocco try catch finally:

- nel blocco try:
  - *util* deve essere inizializzato mediante la chiamata `getInstance()` di `HibernateUtil`;
  - quindi a partire da *util* si inizializza *factory* mediante una chiamata `getFactory()`;
  - analogamente a quanto fatto in precedenza, si inizializza *session* tramite una chiamata `openSession()` eseguita a partire da *factory*;
  - transaction viene inizializzata con una chiamata a metodo `beginTransaction()` eseguita da *session*;
  - query viene inizializzata mediante una chiamata `createQuery`, chiamata da *session* e a cui deve essere passata come parametro, la stringa *hqlQuery*;
  - quindi si è pronti per eseguire query ed ottenere la lista di `ICall` desiderata. Tale operazione si effettua richiamando `query.list()`;
  - si esegue il *commit*;
- nel blocco catch: si effettua un *rollback* a partire da *transaction*;
- nel blocco finally: se *session* è diverso da *null*, si chiude mediante una chiamata di tipo `close()`;

Il programma termina controllando se *calls* è diversa da *null* e contiene almeno un elemento. Nel caso il programma termina ritornando la lista così ottenuta. Altrimenti termina ritornando *null*.

#### 4.1.6 CallListDAO

##### Funzione

Classe che implementa il pattern DAO. Definisce le procedure d'inserimento, eliminazione ed aggiornamento di entry inerenti alla tabella CallLists. La classe inoltre fornisce metodi per interrogare il database ed ottenere dati utente.

##### Relazioni d'uso

- java.util.List:
- org.hibernate.Query:
- org.hibernate.Session:
- org.hibernate.SessionFactory:
- org.hibernate.Transaction:
- call.ICallList:

##### Attributi

Nessun attributo evidenziato.

##### Metodi

+ delete(entry: ICallList): boolean

Metodo usato per cancellare una *entry* dal database. Per *entry* si intende un qualsiasi oggetto implementante l'interfaccia ICallList. Il metodo riceve come parametro d'ingresso un oggetto call.ICallList e restituisce un valore booleano *true* se l'operazione è andata a buon fine, *false* altrimenti. Il flusso principale inizia definendo le seguenti variabili:

- viene inizializzata una variabile di tipo *boolean* nominata *flag*, a *false*;
- si crea un istanza di *HibernateUtil* *util*;
- si crea un istanza di *Session* *session*;
- si crea un istanza di *SessionFactory* *factory*;
- si crea un istanza di *Transaction* *transaction*;

Quindi nel metodo deve essere definito un blocco *try-catch* finally:

- nel blocco *try*:
  - *util* deve essere inizializzato mediante la chiamata `getInstance()` di *HibernateUtil*;
  - quindi a partire da *util* si inizializza *factory* mediante una chiamata `getFactory()`;
  - analogamente a quanto fatto in precedenza, si inizializza *session* tramite una chiamata `openSession()` eseguita a partire da *factory*;
  - *transaction* viene inizializzata con una chiamata a metodo `beginTransaction()` eseguita da *session*;
  - quindi si è pronti per eseguire l'istruzione di inglese *delete* a partire da *session* e passando come parametro *entry*;
  - si esegue il *commit*;
  - si pone *flag* a *true*;
- nel blocco *catch*: si effettua un rollback a partire da *transaction*;

- nel blocco *finally*: se *session* è diverso da *null*, si chiude mediante una chiamata di tipo `close()`;

Infine il metodo termina ritornando il valore di *flag*;

#### + `update(entry: ICallList): boolean`

Metodo usato per modificare una *entry* presente nel database. Il metodo riceve come parametro d'ingresso un oggetto `call.ICallList` e restituisce un valore booleano *true* se l'operazione è andata a buon fine, *false* altrimenti. Il flusso principale inizia definendo le seguenti variabili:

- viene inizializzata una variabile di tipo *boolean* nominata *flag*, a *false*;
- si crea un'istanza di `HibernateUtil` *util*;
- si crea un'istanza di `Session` *session*;
- si crea un'istanza di `SessionFactory` *factory*;
- si crea un'istanza di `Transaction` *transaction*;

Quindi nel metodo deve essere definito un blocco *try-catch finally*:

- nel blocco *try*:
  - *util* deve essere inizializzato mediante la chiamata `getInstance()` di `HibernateUtil`;
  - quindi a partire da *util* si inizializza *factory* mediante una chiamata `getFactory()`;
  - analogamente a quanto fatto in precedenza, si inizializza *session* tramite una chiamata `openSession()` eseguita a partire da *factory*;
  - *transaction* viene inizializzata con una chiamata al metodo `beginTransaction()` eseguita da *session*;
  - quindi si è pronti per eseguire l'istruzione di *update* a partire da *session* e passando come parametro *entry*;
  - si esegue il commit;
  - si pone *flag* a *true*;
- nel blocco *catch*: si effettua un *rollback* a partire da *transaction*;
- nel blocco *finally*: se *session* è diverso da *null*, si chiude mediante una chiamata di tipo `close()`;

Infine il metodo termina ritornando il valore di *flag*;

#### + `insert(entry: ICallList): boolean`

Metodo usato per inserire una *entry* nel database. Il metodo riceve come parametro d'ingresso un oggetto `call.ICallList` e restituisce un valore booleano *true* se l'operazione è andata a buon fine, *false* altrimenti. Il flusso principale inizia definendo le seguenti variabili:

- viene inizializzata una variabile di tipo *boolean* nominata *flag*, a *false*;
- si crea un'istanza di `HibernateUtil` *util*;
- si crea un'istanza di `Session` *session*;
- si crea un'istanza di `SessionFactory` *factory*;
- si crea un'istanza di `Transaction` *transaction*;

Quindi nel metodo deve essere definito un blocco *try-catch finally*:

- nel blocco *try*:

- *util* deve essere inizializzato mediante la chiamata `getInstance()` di `HibernateUtil`;
- quindi a partire da *util* si inizializza *factory* mediante una chiamata `getFactory()`;
- analogamente a quanto fatto in precedenza, si inizializza *session* tramite una chiamata `openSession()` eseguita a partire da *factory*;
- *transaction* viene inizializzata con una chiamata a metodo `beginTransaction()` eseguita da *session*;
- quindi si è pronti per eseguire l'istruzione di *save* a partire da *session* e passando come parametro *entry*;
- si esegue il *commit*;
- si pone *flag* a *true*;
- nel blocco *catch*: si effettua un *rollback* a partire da *transaction*;
- nel blocco *finally*: se *session* è diverso da *null*, si chiude mediante una chiamata di tipo `close()`;

Infine il metodo termina ritornando il valore di *flag*;

#### 4.1.7 MessageDAO

##### Funzione

Classe che implementa il pattern DAO. Definisce le procedure d'inserimento, eliminazione ed aggiornamento di entry inerenti alla tabella Message. La classe inoltre fornisce metodi per interrogare il database ed ottenere la lista delle chiamate.

##### Relazioni d'uso

- `java.util.List`:
- `org.hibernate.Query`:
- `org.hibernate.Session`:
- `org.hibernate.SessionFactory`:
- `org.hibernate.Transaction`:
- `abook.IMessage`:

##### Attributi

Nessun attributo evidenziato.

##### Metodi

+ `delete(message: IMessage): boolean`

Metodo usato per cancellare una chiamata dal database. Il flusso principale inizia definendo le seguenti variabili:

- viene inizializzata una variabile di tipo boolean nominata *flag*, a *false*;
- si crea un'istanza di `HibernateUtil` *util*;
- si crea un'istanza di `Session` *session*;
- si crea un'istanza di `SessionFactory` *factory*;
- si crea un'istanza di `Transaction` *transaction*;

Quindi nel metodo deve essere definito un blocco try catch finally:

- nel blocco try:
  - util deve essere inizializzato mediante la chiamata `getInstance()` di `HibernateUtil`;
  - quindi a partire da util si inizializza factory mediante una chiamata `getFactory()`;
  - analogamente a quanto fatto in precedenza, si inizializza session tramite una chiamata `openSession()` eseguita a partire da factory;
  - transaction viene inizializzata con una chiamata a metodo `beginTransaction()` eseguita da session;
  - quindi si è pronti per eseguire l'istruzione di *delete* a partire da session e passando come parametro call;
  - si esegue il commit;
  - si pone flag a *true*;
- nel blocco catch: si effettua un rollback a partire da transaction;
- nel blocco finally: se session è diverso da null, si chiude mediante una chiamata di tipo `close()`;

Infine il metodo termina ritornando il valore di flag;

+ `update(message: IMessage): boolean`

Metodo usato per modificare una chiamata presente nel database. Il metodo riceve come parametro d'ingresso un oggetto `message.Message` e restituisce un valore booleano pari a *true* se l'operazione è andata a buon fine, *false* altrimenti. Il flusso principale inizia definendo le seguenti variabili:

- viene inizializzata una variabile di tipo boolean nominata flag, a *false*;
- si crea un'istanza di `HibernateUtil` util;
- si crea un'istanza di `Session` session;
- si crea un'istanza di `SessionFactory` factory;
- si crea un'istanza di `Transaction` transaction;

Quindi nel metodo deve essere definito un blocco try catch finally:

- nel blocco try:
  - util deve essere inizializzato mediante la chiamata `getInstance()` di `HibernateUtil`;
  - quindi a partire da *util* si inizializza *factory* mediante una chiamata `getFactory()`;
  - analogamente a quanto fatto in precedenza, si inizializza *session* tramite una chiamata `openSession()` eseguita a partire da *factory*;
  - transaction viene inizializzata con una chiamata a metodo `beginTransaction()` eseguita da session;
  - quindi si è pronti per eseguire l'istruzione di *update* a partire da *session* e passando come parametro *message*;
  - si esegue il *commit*;
  - si pone *flag* a *true*;
- nel blocco catch: si effettua un *rollback* a partire da *transaction*;
- nel blocco finally: se session è diverso da null, si chiude mediante una chiamata di tipo `close()`;

Infine il metodo termina ritornando il valore di *flag*;



+ `insert(message: IMessage): boolean`

Metodo usato per inserire una chiamata nel database, riceve come parametro d'ingresso un oggetto `message.IMessage` e restituisce un valore booleano pari a *true* se l'operazione è andata a buon fine, *false* altrimenti. Il flusso principale inizia definendo le seguenti variabili:

- viene inizializzata una variabile di tipo *boolean* nominata *flag*, a *false*;
- si crea un'istanza di `HibernateUtil` *util*;
- si crea un'istanza di `Session` *session*;
- si crea un'istanza di `SessionFactory` *factory*;
- si crea un'istanza di `Transaction` *transaction*;

Quindi nel metodo deve essere definito un blocco *try-catch finally*:

- nel blocco *try*:
  - *util* deve essere inizializzato mediante la chiamata `getInstance()` di `HibernateUtil`;
  - quindi a partire da *util* si inizializza *factory* mediante una chiamata `getFactory()`;
  - analogamente a quanto fatto in precedenza, si inizializza *session* tramite una chiamata `openSession()` eseguita a partire da *factory*;
  - *transaction* viene inizializzata con una chiamata al metodo `beginTransaction()` eseguita da *session*;
  - quindi si è pronti per eseguire l'istruzione di *save* a partire da *session* e passando come parametro *message*;
  - si esegue il *commit*;
  - si pone *flag* a *true*;
- nel blocco *catch*: si effettua un *rollback* a partire da *transaction*;
- nel blocco *finally*: se *session* è diverso da *null*, si chiude mediante una chiamata di tipo `close()`;

Infine il metodo termina ritornando il valore di *flag*;

+ `getByID(identifier: long): List<IMessage>`

Metodo usato per interrogare il database ed ottenere una lista d'istanze di `IMessage` aventi come identificativo dell'utente proprietario del messaggio, l'identificativo ricevuto come parametro d'ingresso. Il flusso principale inizia definendo le seguenti variabili:

- si crea un'istanza di `HibernateUtil` *util*;
- si crea un'istanza di `List<IMessage>` *messages*;
- si crea un'istanza di `Query` *query*;
- si crea una stringa *hqlQuery* impostata per default a:  
"from message as m where m.owner = :id"
- si crea un'istanza di `Session` *session*;
- si crea un'istanza di `SessionFactory` *factory*;
- si crea un'istanza di `Transaction` *transaction*;

Quindi nel metodo deve essere definito un blocco *try catch finally*:

- nel blocco *try*:
  - *util* deve essere inizializzato mediante la chiamata `getInstance()` di `HibernateUtil`;
  - quindi a partire da *util* si inizializza *factory* mediante una chiamata `getFactory()`;

- analogamente a quanto fatto in precedenza, si inizializza *session* tramite una chiamata `openSession()` eseguita a partire da *factory*;
- *transaction* viene inizializzata con una chiamata a metodo `beginTransaction()` eseguita da *session*;
- *query* viene inizializzata mediante una chiamata `createQuery`, chiamata da *session* e a cui deve essere passata come parametro, la stringa `hqlQuery`;
- quindi si è pronti per eseguire *query* ed ottenere la lista di `IMessage` desiderata. Tale operazione si effettua richiamando `query.list()`;
- si esegue il *commit*;
- nel blocco *catch*: si effettua un *rollback* a partire da *transaction*;
- nel blocco *finally*: se *session* è diverso da *null*, si chiude mediante una chiamata di tipo `close()`;

Il programma termina controllando se *messages* è diversa da *null* e contiene almeno un elemento. Nel caso il programma termina ritornando la lista così ottenuta. Altrimenti termina ritornando *null*.

## 4.2 Package `org.softwaresynthesis.mytalk.server.abook`

### 4.2.1 `IUserData`

#### Funzione

Interfaccia rappresentante il comportamento di un generico utente del sistema. L'interfaccia dovrà definire dei metodi di tipo *get* e *set* per i dati d'interesse.

#### Relazioni d'uso

- `org.softwaresynthesis.mytalk.server.IMyTalkObject`: interfaccia da estendere. Ogni oggetto che implementerà l'interfaccia `IUserData` dovrà essere in grado di convertire il proprio contenuto informativo in formato *Json*.
- `AddressBookEntry`: l'interfaccia definisce dei metodi per la manipolazione di dati `AddressBookEntry`.

#### Metodi

- + `getId(): Long`  
Restituisce l'identificatore univoco di uno `IUserData`.
- + `getEmail(): String`  
Restituisce l'indirizzo e-mail con cui uno `IUserData` si è registrato nel sistema `MyTalk`.
- + `setEmail(mail: String): void`  
Imposta l'indirizzo e-mail con cui si registra nel sistema `MyTalk` uno `IUserData`.
- + `getPassword(): String`  
Restituisce la password di accesso al sistema `MyTalk` di uno `IUserData`.
- + `setPassword(password: String): void`  
Imposta la password di accesso al sistema di uno `IUserData`.
- + `getQuestion(): String`  
Restituisce la domanda segreta, scelta da uno `IUserData`, per il recupero della *password* smarrita di accesso al sistema `MyTalk`.

- + `setQuestion(question: String): void`  
Imposta la domanda segreta, scelta da uno `IUserData`, per il recupero della *password* smarrita di accesso al sistema MyTalk.
- + `getAnswer(): String`  
Restituisce la risposta alla domanda per il recupero della *password* smarrita di accesso al sistema MyTalk.
- + `setAnswer(answer: String): void`  
Imposta la risposta alla domanda segreta per il recupero della *password* di accesso al sistema MyTalk.
- + `getName(): String`  
Restituisce il nome di uno `IUserData`.
- + `setName(name: String): void`  
Imposta il nome di uno `IUserData`.
- + `getSurname(): String`  
Restituisce il cognome di uno `IUserData`.
- + `setSurname(surname: String): void`  
Imposta il cognome di uno `IUserData`.
- + `getPicturePath(): String`  
Restituisce una stringa con il percorso dell'immagine del profilo di uno `IUserData`.
- + `setPicturePath(path: String): void`  
Imposta il percorso dell'immagine profilo di uno `IUserData`.
- + `getAddressBook(): Set<AddressBookEntry>`  
Metodo che ritorna la rubrica dell'utente sotto forma d'insieme di `AddressBookEntry`.
- + `addAddressBookEntry(entry: AddressBookEntry): void`  
Metodo usato per aggiungere una nuova `AddressBookEntry` all'insieme di *entry* che costituisce la rubrica utente.

#### 4.2.2 IGroup

##### Funzione

Interfaccia rappresentante un gruppo di una rubrica utente del sistema MyTalk.

##### Relazioni d'uso

- `org.softwaresynthesis.mytalk.server.IMyTalkObject`: interfaccia da estendere. Ogni oggetto che implementerà l'interfaccia `IGroup` dovrà essere in grado di convertire il proprio contenuto informativo in formato *Json*.

##### Metodi

- + `getId(): Long`  
Restituisce l'identificativo univoco di uno gruppo di una rubrica utente.
- + `getName(): String`  
Restituisce il nome di un gruppo di una rubrica utente.
- + `setName(name: String): void`  
Imposta il nome di un gruppo di una rubrica utente.

### 4.2.3 IAddressBookEntry

#### Funzione

Interfaccia rappresentante una *entry* di una rubrica utente del sistema mytalk.

#### Relazioni d'uso

- `org.softwaresynthesis.mytalk.server.IMyTalkObject`: interfaccia da estendere. Ogni oggetto che implementerà l'interfaccia `IAddressBookEntry` dovrà essere in grado di convertire il proprio contenuto informativo in formato *Json*.
- `IUserData`: l'interfaccia `IAddressBookEntry` definisce più metodi che restituiscono oggetti aventi tipo di ritorno `IUserData`, essi sono i metodi *get* per ottenere il "proprietario" della rubrica e per ottenere l'utente registrato nella rubrica. Analogamente `IUserData` viene usato come parametro d'ingresso per i metodi *set* collegati ai metodi già citati.

#### Metodi

- + `getId(): Long`  
Restituisce l'identificativo univoco di una *entry* di una rubrica utente del sistema MyTalk.
- + `getEntry(): IUserData`  
Restituisce un'istanza di un oggetto avente tipo `IUserData` rappresentante un contatto della rubrica.
- + `setEntry(contact: IUserData): void`  
Imposta l'utente `IUserData` (passato come parametro d'ingresso) come contatto della rubrica.
- + `getGroup(): IGroup`  
Restituisce il gruppo a cui appartiene lo `IUserData` registrato nella rubrica.
- + `setGroup(group: IGroup): void`  
Imposta il gruppo di appartenenza dello `IUserData` registrato nella rubrica.
- + `getOwner(): IUserData`  
Restituisce lo `IUserData` possessore dell'*entry* corrente della rubrica.
- + `setOwner(owner: IUserData ): void`  
Imposta l'utente `IUserData` possessore della *entry* della rubrica.

### 4.2.4 UserData

#### Funzione

Implementazione dell'interfaccia `IUserData`. Un'istanza della classe dovrà rappresentare un generico utente del sistema definendone gli attributi e i metodi per impostare ed ottenere il contenuto dei medesimi.

#### Relazioni d'uso

- `IUserData`: interfaccia da implementare.

## Attributi

- - **id**: **long** Attributo che definisce il codice identificativo con il quale l'utente è registrato nel database del sistema.
- - **mail**: **String** Attributo che definisce l'indirizzo e-mail con il quale l'utente si è registrato nel sistema.
- - **password**: **String** Attributo che definisce la password per il *login* dell'utente nel sistema.
- - **question**: **String** Attributo che definisce la domanda segreta usata dall'utente in caso di smarrimento della password.
- - **answer**: **String** Attributo che definisce la risposta alla domanda segreta definita nell'attributo **question**.
- - **name**: **String** Attributo che definisce il nome dell'utente.
- - **surname**: **String** Attributo che definisce il cognome dell'utente.
- - **path**: **String** Attributo che definisce il percorso (sul server) in cui è memorizzata l'immagine del profilo dell'utente.
- - **addressBook**: **Set<AddressBookEntry>** Attributo che definisce l'insieme di **AddressBookEntry** che costituiscono la rubrica dell'utente.

## Metodi

- + **getId(): Long**  
Restituisce l'identificatore univoco di un utente, ritornando l'attributo **id**.
- # **setId(id: long): void**  
Imposta l'indirizzo *id* con cui l'utente si registra nel sistema MyTalk. Il metodo sovrascrive il contenuto dell'attributo **id** con il valore tipo **long** ricevuto come parametro d'ingresso, ritornando l'attributo **id**.
- + **getEmail(): String**  
Restituisce l'indirizzo e-mail con cui uno l'utente si è registrato nel sistema MyTalk, ritornando il contenuto dell'attributo **mail**.
- + **setEmail(mail: String): void**  
Imposta l'indirizzo e-mail con cui l'utente si registra nel sistema MyTalk. Il metodo non fa altro che sovrascrivere il contenuto dell'attributo **mail** con il valore tipo **String** ricevuto come parametro d'ingresso.
- + **getPassword(): String**  
Restituisce la password dell'utente, ritornando il valore contenuto nell'attributo **password**.
- + **setPassword(password: String): void**  
Imposta la password di accesso al sistema, sovrascrivendo il contenuto dell'attributo **password** con il valore di tipo **String** ricevuto come parametro d'ingresso.
- + **getQuestion(): String**  
Restituisce la domanda segreta, scelta dall'utente, per il recupero della password di accesso al sistema MyTalk. Nello specifico il metodo restituisce il contenuto dell'attributo **question**.

- + `setQuestion(question: String): void`  
Imposta la domanda segreta da inserire in caso di smarrimento della password. Il metodo sovrascrive il contenuto dell'attributo `question` con il valore tipo `String` ricevuto come parametro d'ingresso
- + `getAnswer(): String`  
Restituisce la risposta alla domanda per il recupero della password (smarrita) di accesso al sistema MyTalk. Il metodo ritorna il contenuto dell'attributo `answer`.
- + `setAnswer(answer: String): void`  
Imposta la risposta alla domanda segreta per il recupero della password. Il metodo sovrascrive il contenuto dell'attributo `answer` con il valore tipo `String` passato come parametro d'ingresso.
- + `getName(): String`  
Restituisce il nome dell'utente ritornando il contenuto dell'attributo `name`.
- + `setName(name: String): void`  
Imposta il nome dell'utente sovrascrivendo il contenuto dell'attributo `name` con il valore tipo `String` passato al metodo come parametro d'ingresso.
- + `getSurname(): String`  
Restituisce il cognome dell'utente restituendo il contenuto dell'attributo `surname`.
- + `setSurname(surname: String): void`  
Imposta il cognome dell'utente sovrascrivendo il contenuto dell'attributo `surname` con il valore tipo `String` passato al metodo come parametro d'ingresso.
- + `getPicturePath(): String`  
Restituisce una stringa con il percorso dell'immagine del profilo dell'utente, restituendo il contenuto dell'attributo `path`.
- + `setPicturePath(path: String): void`  
Imposta il percorso dell'immagine profilo di un utente, sovrascrivendo il contenuto dell'attributo `path` con il valore tipo `String` passato al metodo come parametro d'ingresso.
- + `getState(): State`  
Restituisce lo stato in cui si trova l'utente, ritornando il contenuto dell'attributo `state`.
- + `setState(state: State): void`  
Imposta lo stato in cui si trova l'utente, sovrascrivendo il contenuto dell'attributo `state` con il valore ricevuto come parametro d'ingresso.
- + `getAddressBook(): Set<AddressBookEntry>`  
Metodo che ritorna il contenuto di `addressBook`.
- + `addAddressBookEntry(entry: AddressBookEntry): void`  
Metodo usato per aggiungere ad `addressBook` una nuova `AddressBookEntry` passata come parametro d'ingresso.
- + `toJson(): String`  
Metodo usato per ritornare il contenuto di un'istanza di `UserData` sotto forma di stringa formattata in *Json*. La stringa ritornata deve corrispondere al seguente formato:

```
{name:"mio_nome",surname:"mio_cognome",email:"mia_mail",  
picturePath:"mia_immagine",id:"mio_id"}
```

dove i valori tra virgolette rappresentano il contenuto dei rispettivi campi dati contenuti nella classe.

#### 4.2.5 Group

##### Funzione

Implementazione dell'interfaccia IGroup.

##### Relazioni d'uso

- IGroup: interfaccia d'implementazione.

##### Attributi

- - **id**: **long** Attributo del codice identificativo del gruppo.
- - **name**: **String**: Attributo del nome del gruppo.

##### Metodi

- + **getId(): Long**  
Restituisce l'identificativo univoco di uno gruppo di una rubrica utente.
- + **getName(): String**  
Restituisce il nome di un gruppo di una rubrica utente.
- + **setName(name: String): void**  
Imposta il nome di un gruppo di una rubrica utente.
- + **toJson(): String**  
Metodo usato per ritornare il contenuto di un istanza di **Group** sotto forma di stringa formattata in *Json*. La stringa ritornata deve corrispondere al seguente formato:

```
{id:"mio_id",name:"mio_nome"}
```

dove i valori tra virgolette rappresentano il contenuto dei rispettivi campi dati contenuti nella classe.

#### 4.2.6 AddressBookEntry

##### Funzione

Implementazione dell'interfaccia IAddressBookEntry.

##### Relazioni d'uso

- IAddressBookEntry: interfaccia d'implementazione della classe.
- IUserData: usata per definire gli attributi destinati a identificare il possessore dell'istanza di AddressBookEntry e il relativo contatto in essa registrato.

##### Attributi

- - **id**: **long** Attributo del codice identificativo della classe.
- - **group**: **IGroup** Attributo destinato a identificare il gruppo a cui appartiene il contatto IUserName registrato nella classe. Si ricorda che il contatto può anche non appartenere ad alcun gruppo.

- - **contact**: **IUserData** Attributo destinato ad identificare il contatto registrato nell'istanza di **AddressBookEntry**.
- - **owner**: **IUserData**: Attributo destinato ad identificare il possessore dell'istanza di **AddressBookEntry**.
- - **blocked**: **boolean** Attributo booleano necessario per bloccare il contatto. Tale blocco avviene impostando l'attributo a *true*.

## Metodi

- + **getId(): Long**  
Restituisce l'identificativo univoco della *entry* di una rubrica utente del sistema MyTalk, nello specifico il contenuto dell'attributo **id**.
- + **getEntry(): IUserData**  
Restituisce il contenuto dell'attributo **contact**.
- + **setEntry(contact: IUserData): void**  
Imposta l'utente **IUserData** (passato come parametro d'ingresso) come contatto della rubrica, nello specifico il contenuto dell'attributo **contact**.
- + **getGroup(): IGroup**  
Restituisce il gruppo a cui appartiene lo **IUserData** registrato nella rubrica, nello specifico il contenuto dell'attributo **group**.
- + **setGroup(group: IGroup): void**  
Imposta il gruppo di appartenenza dello **IUserData** registrato nella rubrica, nello specifico il contenuto dell'attributo **group**.
- + **getOwner(): IUserData**  
Restituisce lo **IUserData** possessore di questa *entry* della rubrica, nello specifico il contenuto dell'attributo **owner**.
- + **setOwner(owner: IUserData ): void**  
Imposta l'utente **IUserData** possessore della *entry* della rubrica, nello specifico il contenuto dell'attributo **owner**.
- + **toJson(): String**  
Metodo usato per ritornare il contenuto di un istanza di **AddressBookEntry** sotto forma di stringa formattata in *Json*. La stringa ritornata deve corrispondere al seguente formato:  
  

```
{id:"mio_id",contact:"mio_contatto",group:"mia_gruppo",blocked:"bloccato"}
```

  
dove i valori tra virgolette rappresentano il contenuto dei rispettivi campi dati contenuti nella classe.

## 4.3 Package org.softwaresynthesis.mytalk.server.abook.servlet

### 4.3.1 AddressBookDoAddContactServlet

#### Funzione

*Servlet* che ha il compito di aggiungere alla rubrica un nuovo contatto.



## Relazioni d'uso

- `java.io.IOException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `java.io.PrintWriter`: classe istanziata all'interno del metodo `doPost()`. Usata per scrivere l'output della *servlet*.
- `javax.servlet.ServletException`: eccezione lanciabile dai metodi `doPost()` e `doGet()`.
- `javax.servlet.http.HttpServlet`: classe estesa da `LoginManager`.
- `javax.servlet.http.HttpServletRequest`: classe per usata per la comunicazione con la *servlet* (inoltra i dati in ingresso alla *servlet*).
- `javax.security.auth.login.LoginContext`: classe usata in `doPost()` per eseguire la *logout* dal sistema.
- `javax.servlet.http.HttpServletResponse`: classe per usata per la comunicazione con la *servlet* (inoltra i dati in uscita dalla chiamata a *servlet*).
- `javax.servlet.http.HttpSession`: classe usata per definire una sessione HTTP.
- `org.softwaresynthesis.mytalk.server.dao.UserDataDAO`: classe usata per comunicare tramite Hibernate con la tabella `UserData` della base di dati.

## Attributi

- `- {frozen} serialVersionUID: long` Attributo contenente l'id per la serializzazione dei dati. Il valore di creazione deve essere 10012L.

## Metodi

### + `AddressBookDoAddContactServlet()`

Costruttore pubblico della *servlet*. Richiama il costruttore della classe padre (chiamata a `super`).

### # `doGet(request HttpServletRequest, response HttpServletResponse): void`

Metodo usato per eseguire la richiesta di aggiungere un nuovo contatto all'interno della rubrica (eseguito in risposta ad una richiesta HTTP con metodo `GET`). Poiché si intende gestire in modo univoco una richiesta a tale *servlet* (indifferentemente dalla tipologia d'invio dati) il metodo reindirizza il flusso principale al metodo `doPost()`.

### # `doPost(request HttpServletRequest, response HttpServletResponse): void`

Metodo usato per eseguire la richiesta di aggiungere un nuovo contatto all'interno della rubrica (eseguito in risposta ad una richiesta HTTP con metodo `POST`). Il flusso principale inizia con la creazione di un oggetto `dao.UserDataDAO` avente nome `UserDAO`. Quindi viene aperto un blocco try catch in cui si salva in un istanza di tipo `HttpSession`, l'oggetto ritornato da una chiamata `request.getSession(false)`. I passi successivi sono (nell'ordine):

- salvare in attributo di tipo `long`, `contactID`, l'id dell'utente. Tale id dovrà essere ottenuto con una chiamata:  
`request.getParameter("contactId");`
- si crea un istanza di tipo `IUserData` a partire da una chiamata a metodo:  
`session.getAttribute("user");`

- si crea un'istanza di tipo `IUserData` che conterrà "l'utente" che dovrà essere registrato nella rubrica del chiamante. L'oggetto dovrà essere istanziato in seguito ad una chiamata:  
`userDAO.getByID(contactId);`

Quindi se l'amico (che dovrò andare a registrare) è stato correttamente istanziato (`amico != null`) allora il flusso principale procede creando e impostando un'istanza di `AddressBookEntry`. Il metodo termina "scrivendo" *true* all'interno di un'istanza di `PrintWriter` creata a partire da `response.getWriter()` ed eseguendo il metodo `addAddressBookEntry()` (passando l'entry creata in precedenza) a partire dall'istanza che rappresenta l'utente richiedente.

Se invece si è osservato che l'oggetto contenente l'amico da aggiungere, ha valore uguale a `null`, allora il metodo termina scrivendo *false* all'interno della dello stesso `PrintWriter` già citato.

#### 4.3.2 AddressBookDoRemoveContactServlet

##### Funzione

*Servlet* richiamata dal *client* per eseguire l'eliminazione di un contatto presente nella propria rubrica.

##### Relazioni d'uso

- `java.io.IOException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `java.io.PrintWriter`: classe istanziata all'interno del metodo `doPost()`. Usata per scrivere l'output della *servlet*.
- `javax.servlet.ServletException`: eccezione lanciabile dai metodi `doPost()` e `doGet()`.
- `javax.servlet.http.HttpServlet`: classe estesa da `LoginManager`.
- `javax.servlet.http.HttpServletRequest`: classe per usata per la comunicazione con la *servlet* (inoltre i dati in ingresso alla *servlet*).
- `javax.security.auth.login.LoginContext`: classe usata in `doPost()` per eseguire la logout dal sistema.
- `javax.servlet.http.HttpServletResponse`: classe per usata per la comunicazione con la *servlet* (inoltre i dati in uscita dalla chiamata a *servlet*).
- `javax.servlet.http.HttpSession`: classe usata per definire una sessione HTTP.
- `org.softwaresynthesis.mytalk.server.dao.UserDataDAO`: classe usata per comunicare tramite Hibernate con la tabella `UserData` della base di dati.

##### Attributi

- `- {frozen} serialVersionUID: long` Attributo contenente l'id per la serializzazione dei dati, il valore di creazione deve essere 10013L.

## Metodi

### + AddressBookDoRemoveContactServlet()

Costruttore pubblico della *servlet*, richiama il costruttore della classe padre (chiamata a *super*).

### # doGet(request HttpServletRequest, response HttpServletResponse): void

Metodo usato per eseguire la richiesta di eliminazione di un contatto presente all'interno della rubrica (eseguito in risposta ad una richiesta HTTP con metodo **GET**). Poiché si intende gestire in modo univoco una richiesta a tale *servlet* (indifferentemente dalla tipologia d'invio dati) il metodo reindirizza il flusso principale al metodo **doPost()**.

### # doPost(request HttpServletRequest, response HttpServletResponse): void

Metodo usato per eseguire la richiesta di eliminazione di un contatto presente all'interno della rubrica (eseguito in risposta ad una richiesta HTTP con metodo **POST**). Tale metodo deve creare un istanza di **AddressBookEntry** con i dati relativi alla *entry* da eliminare, quindi richiamerà il metodo **removeAddressBookEntry()** a partire dalle due istanze di **IUserData** (quella che rappresenta l'utente richiedete e quella che rappresenta l'utente da eliminare) passando come parametro l'*entry* definita in precedenza. Più nello specifico il flusso principale inizia con la creazione di un oggetto **dao.UserDataDAO** avente nome **UserDAO**, quindi viene aperto un blocco *try-catch* in cui si salva in un istanza di tipo **HttpSession**, l'oggetto ritornato da una chiamata **request.getSession(false)**. I passi successivi sono (nell'ordine):

- salvare in attributo di tipo long, **contactID**, l'id dell'utente. Tale id dovrà essere ottenuto con una chiamata:  
`request.getParameter("contactId");`
- si crea un istanza di tipo **IUserData** a partire da una chiamata a metodo:  
`session.getAttribute("user");`
- si crea un istanza di tipo **IUserData** che conterrà "l'utente" che dovrà essere eliminato dalla rubrica del chiamante. L'oggetto dovrà essere istanziato in seguito ad una chiamata:  
`userDAO.getByID(contactId);`

Quindi se l'amico (che dovrò andare a cancellare dalla lista) è stato correttamente istanziato (**amico != null**) allora il flusso principale procede creando e impostando un istanza di **AddressBookEntry**. Quindi a partire dall'oggetto che rappresenta l'utente richiedente, viene richiamato il metodo **removeAddressBookEntry()**. Il metodo termina "scrivendo" *true* all'interno di un istanza di **PrintWriter** creata a partire da **response.getWriter()**.

Se invece si è osservato che l'oggetto contenente l'amico da rimuovere, ha valore uguale a *null*, allora il metodo termina impostando *false* all'intero della dello stesso **PrintWriter** già citato.

## 4.3.3 AddressBookDoCreateGroupServlet

### Funzione

*Servlet* richiamata dal *client* per creare un nuovo gruppo nella propria rubrica.

### Relazioni d'uso

- **java.io.IOException**: eccezione richiamabile dai metodi **doPost()** e **doGet()**.

- `java.io.PrintWriter`: classe istanziata all'interno del metodo `doPost()`. Usata per scrivere l'output della *servlet*.
- `javax.servlet.ServletException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `javax.servlet.http.HttpServlet`: classe estesa da `LoginManager`.
- `javax.servlet.http.HttpServletRequest`: classe per usata per la comunicazione con la *servlet* (inoltra i dati in ingresso alla *servlet*).
- `javax.security.auth.login.LoginContext`: classe usata in `doPost()` per eseguite la logout dal sistema.
- `javax.servlet.http.HttpServletResponse`: classe per usata per la comunicazione con la *servlet* (inoltra i dati in uscita dalla chiamata a *servlet*).
- `javax.servlet.http.HttpSession`: classe usata per definire una sessione HTTP.
- `org.softwaresynthesis.mytalk.server.dao.UserDataDAO`: classe usata per comunicare tramite Hibernate con la tabella `UserData` della base di dati.

### Attributi

- `- {frozen} serialVersionUID: long` Attributo contenente l'id per la serializzazione dei dati. Il valore di creazione deve essere 10015L.

### Metodi

#### + `AddressBookDoCreateGroupServlet()`

Costruttore pubblico della *servlet*. Richiama il costruttore della classe padre (chiamata a `super`).

#### # `doGet(request HttpServletRequest, response HttpServletResponse): void`

Metodo usato per eseguire la richiesta di creazione di un gruppo (eseguito in risposta ad una richiesta HTTP con metodo GET). Poiché si intende gestire in modo univoco una richiesta a tale *servlet* (indifferentemente dalla tipologia d'invio dati) il metodo reindirizza il flusso principale al metodo `doPost()`.

#### # `doPost(request HttpServletRequest, response HttpServletResponse): void`

Metodo usato per eseguire la richiesta di creazione di un gruppo (eseguito in risposta ad una richiesta HTTP con metodo POST). Nell'ordine proposto, devono essere eseguite le seguenti operazioni:

- creazione di un oggetto `dao.GroupDAO` avente nome `groupDAO`;
- creazione di un oggetto `HttpSession` avente nome `session`;
- creazione di un oggetto `abook.IGroup` avente nome `group`;
- creazione di un oggetto `abook.IUserData` avente nome `user`;
- creazione di un oggetto `PrinterWriter` avente nome `writer`;
- creazione di due stringhe, `name` e `result`;

A questo punto viene creato un blocco *try-catch*. Il blocco *catch* imposta la stringa *result* a *false*, il blocco *try* dovrà invece seguire il seguente iter:

- si imposta *session* con il valore ritornato da una chiamata:  
`request.getSession(false)`;

- si imposta la variabile *user* con il valore presente nell'attributo *user* di sessione (usare l'istruzione `session.getAttribute("user");`);
- impostare *name* con il contenuto del parametro *groupName* presente nell'oggetto request;
- verificare se *name* è *null* o vuoto. Nel caso procedere impostando *result* a *false* e uscendo dal blocco *try*.
- altrimenti, se *name* contiene un nome valido si crea un nuovo gruppo (salvandolo nella variabile *group*), e impostando il nome e l'utente proprietario.
- richiamare `insert()` a partire dall'istanza *groupDAO*, passando come parametro *group* (che ora conterrà in gruppo da inserire nel database). Viene impostato *result* a *true*, il metodo termina scrivendo su *write* (usando l'omonimo metodo) il valore contenuto in *result*.

#### 4.3.4 AddressBookDoDeleteGroupServlet

##### Funzione

*Servlet* richiamata dal *client* per eliminare un gruppo da una rubrica.

##### Relazioni d'uso

- `java.io.IOException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `java.io.PrintWriter`: classe istanziata all'interno del metodo `doPost()`. Usata per scrivere l'output della *servlet*.
- `javax.servlet.ServletException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `javax.servlet.http.HttpServlet`: classe estesa da `LoginManager`.
- `javax.servlet.http.HttpServletRequest`: classe per usata per la comunicazione con la *servlet* (inoltra i dati in ingresso alla *servlet*).
- `javax.security.auth.login.LoginContext`: classe usata in `doPost()` per eseguite la logout dal sistema.
- `javax.servlet.http.HttpServletResponse`: classe per usata per la comunicazione con la *servlet* (inoltra i dati in uscita dalla chiamata a *servlet*).
- `javax.servlet.http.HttpSession`: classe usata per definire una sessione HTTP.
- `org.softwaresynthesis.mytalk.server.dao.UserDataDAO`: classe usata per comunicare tramite Hibernate con la tabella *UserData* della base di dati.

##### Attributi

- `- {frozen} serialVersionUID: long` Attributo contenente l'id per la serializzazione dei dati. Il valore di creazione deve essere 10015L.

##### Metodi

###### + `AddressBookDoDeleteGroupServlet()`

Costruttore pubblico della *servlet*. Richiama il costruttore della classe padre (chiamata a `super`).

```
# doGet(request HttpServletRequest, response HttpServletResponse): void
```

Metodo usato per eseguire la richiesta di eliminazione di un gruppo (eseguito in risposta ad una richiesta HTTP con metodo GET). Poiché si intende gestire in modo univoco una richiesta a tale *servlet* (indifferentemente dalla tipologia d'invio dati) il metodo reindirizza il flusso principale al metodo `doPost()`.

```
# doPost(request HttpServletRequest, response HttpServletResponse): void
```

Metodo usato per eseguire la richiesta di eliminazione di un gruppo (eseguito in risposta ad una richiesta HTTP con metodo POST). Nell'ordine proposto, devono essere eseguite le seguenti operazioni:

- creazione di un'istanza di `dao.AddressBookEntryDAO` denominata `entryDAO`;
- creazione di un `dao.GroupDAO` denominata `groupDAO`.
- creazione di un'istanza di `abook.IGroup` denominata `group`;
- creazione di un'istanza di `abook.IAddressBookEntry` denominata `entry`;
- creazione di un'istanza di `Iterator<IAddressBookEntry>` denominata `iterator`;
- creazione di un'istanza di `PrintWriter` denominata `writer`;
- creazione di un `Set<IAddressBookEntry>` denominato `entrys`;
- creazione di una stringa denominata `result` (utilizzata per registrare il messaggio da stampare sul `writer`);

A questo punto deve essere definito un blocco *try-catch*. Dentro il blocco *catch* si imposta `result` con il valore *false*. Per quanto riguarda il blocco *try*, il programmatore dovrà definire il seguente iter:

- salvare in `group` il valore ritornato da una chiamata `getByID` a cui passo l'identificativo del gruppo (`request.getParameter(groupId)`) a partire dall'oggetto `groupDAO`.
- quindi si esegue una verifica sul contenuto di `group`. Se `group == null` allora si imposta `result` a *false*. Altrimenti si carica in `entry` l'oggetto ritornato da una chiamata `getAddressBook()` a partire da `group`.
- a tal punto (sempre dentro al costrutto condizionale `if(group != null)`) si crea un iteratore tramite `entrys.iterator()` e ciclando con tale iteratore si va a modificare la voce "gruppo" di tutti i contatti che nella mia rubrica appartengono a tale gruppo (così facendo l'eliminazione del gruppo non porterà all'eliminazione dei contatti presenti in tale gruppo).
- dunque si esegue la chiamata: `groupDAO.delete(group)` e si imposta `result` a *true*.

Il metodo termina scrivendo su `writer` il contenuto di `result`.

#### 4.3.5 AddressBookDoInsertInGroupServlet

##### Funzione

*Servlet* richiamata dal *client* per inserire un utente nella propria rubrica, all'interno di un gruppo ben determinato.

##### Relazioni d'uso

- `java.io.IOException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `java.io.PrintWriter`: classe istanziata all'interno del metodo `doPost()`. Usata per scrivere l'output della *servlet*.

- `javax.servlet.ServletException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `javax.servlet.http.HttpServlet`: classe estesa da `LoginManager`.
- `javax.servlet.http.HttpServletRequest`: classe per usata per la comunicazione con la *servlet* (inoltra i dati in ingresso alla *servlet*).
- `javax.servlet.http.HttpServletResponse`: classe per usata per la comunicazione con la *servlet* (inoltra i dati in uscita dalla chiamata a *servlet*).
- `javax.servlet.http.HttpSession`: classe usata per definire una sessione HTTP.
- `abook.IAddressBookEntry`: Interfaccia che definisce il comportamento di una generica *entry* della rubrica utente. La classe che verrà descritta crea istanze di tipo `abook.IAddressBookEntry`.
- `abook.IGroup`: Interfaccia che definisce il comportamento di un gruppo generico. La classe che qui si descrive crea istanze di tipo `abook.IGroup`.
- `abook.IUserData`: Interfaccia che definisce il comportamento di un utente generico. La classe che qui si descrive crea istanze di tipo `abook.IUserData`.
- `abook.AddressBookEntry`: classe usata per comunicare tramite *Hibernate* con la tabella `AddressBookEntry` della base di dati.
- `dao.GroupDAO`: classe usata per comunicare tramite *Hibernate* con la tabella `Group` della base di dati.
- `dao.UserDataDAO`: classe usata per comunicare tramite *Hibernate* con la tabella `UserData` della base di dati.

### Attributi

- `- {frozen} serialVersionUID: long` Attributo contenente l'id per la serializzazione dei dati, il valore di creazione deve essere 10015L.

### Metodi

#### + `AddressBookDoInsertInGroupServlet()`

Costruttore pubblico della *servlet*, richiama il costruttore della classe padre (chiamata a `super`).

#### # `doGet(request HttpServletRequest, response HttpServletResponse): void`

Metodo usato per eseguire la richiesta di aggiungere un contatto in un gruppo (eseguito in risposta ad una richiesta HTTP con metodo `GET`). Poiché si intende gestire in modo univoco una richiesta a tale *servlet* (indifferentemente dalla tipologia d'invio dati) il metodo reindirizza il flusso principale al metodo `doPost()`.

#### # `doPost(request HttpServletRequest, response HttpServletResponse): void`

Metodo usato per eseguire la richiesta di aggiungere un contatto in un gruppo (eseguito in risposta ad una richiesta HTTP con metodo `POST`). Nell'ordine proposto, devono essere eseguite le seguenti operazioni:

- creazione di un istanza di `GroupDAO` denominata `groupDAO`;
- creazione di una sessione `HttpSession` avente nome `session`;
- creazione degli oggetti rappresentanti la realtà della rubrica su cui opero: un `IAddressBookEntry` `entry`, `IGroup` `group`, `IUserData` `user` e `IUserData` `friend`;
- creazione di due identificativi di tipo `long`: `contactId` e `groupId`.



- creazione di un `PrintWriter` denominato *writer* e di una stringa *result* usata con lo scopo di memorizzare il contenuto testuale da scrivere sul writer come messaggio di notifica della *servlet*;
- creazione di un istanza `UserDataDAO` *userDAO*.

Dopo questa fase di creazione delle variabili il metodo deve definire un costrutto *try-catch*. All'interno del blocco *catch* dovrà essere predisposta la memorizzazione della parola "*false*". Passando invece alla definizione del blocco *try*, al suo interno si dovranno predisporre le seguenti istruzioni:

- inizializzazione di session al valore ottenuto da una chiamata:  
`request.getSession(false)`
- istanziazione di user a partire dal valore contenuto nella *request*. Per la precisione si dovrà usare un istruzione del tipo:  
`session.getAttribute("user")`
- con una procedura analoga alla precedente definisco il contenuto di *contactID*. Usare `getParameter("contactId")` a partire dall'oggetto *request*;
- istruzione analoga alla precedente usata per istanziare il contenuto di *groupId* (nome del parametro da ottenere: *groupId*);
- initialize friend con una chiamata `getIdByContactId` passando come parametro *contactId*.
- initialize group con una chiamata `getIdByContactId` passando come parametro *groupId*.
- il metodo controlla se `group != null`. Nel caso il flusso principale prosegue come segue: inizializzazione di *entry* e modifica dei dati stessi di *entry* mediante le chiamate a metodo dei vari "set" che la costituiscono. Nello specifico si intende impostare l'istanza in modo che definisca un contatto in rubrica non bloccato e registrato nel gruppo *group*. Il possessore sarà user e il contatto registrato friend. Quindi viene eseguito l'*update* tramite una chiamata:  
`userDAO.update(user)` *result* viene impostato a *true* e il programma esce dal costrutto condizionale.
- se *group* non era diverso da *null* allora si entra nel ramo *else* del costrutto condizionale già citato. *Result* viene impostato a *false* e il metodo esce dal ramo *else*.

Il metodo termina scrivendo su *writer* il contenuto di *result*.

#### 4.3.6 AddressBookDoRemoveFromGroupServlet

##### Funzione

*Servlet* richiamata dal *client* per inserire un utente nella propria rubrica, all'interno di un gruppo ben determinato.

##### Relazioni d'uso

- `java.io.IOException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `java.io.PrintWriter`: classe istanziata all'interno del metodo `doPost()`. Usata per scrivere l'output della *servlet*.
- `javax.servlet.ServletException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `javax.servlet.http.HttpServlet`: classe estesa da `LoginManager`.
- `javax.servlet.http.HttpServletRequest`: classe per usata per la comunicazione con la *servlet* (inoltre i dati in ingresso alla *servlet*).



- `javax.servlet.http.HttpServletResponse`: classe per usata per la comunicazione con la *servlet* (inoltre i dati in uscita dalla chiamata a *servlet*).
- `javax.servlet.http.HttpSession`: classe usata per definire una sessione HTTP.
- `abook.IAddressBookEntry`: Interfaccia che definisce il comportamento di una generica *entry* della rubrica utente. La classe che verrà descritta crea istanze di tipo `abook.IAddressBookEntry`.
- `abook.IGroup`: Interfaccia che definisce il comportamento di un gruppo generico. La classe che verrà descritta crea istanze di tipo `abook.IGroup`.
- `abook.IUserData`: Interfaccia che definisce il comportamento di un utente generico. La classe che verrà descritta crea istanze di tipo `abook.IUserData`.
- `abook.AddressBookEntry`: classe usata per comunicare tramite *Hibernate* con la tabella `AddressBookEntry` della base di dati.
- `dao.GroupDAO`: classe usata per comunicare tramite *Hibernate* con la tabella `Group` della base di dati.
- `dao.UserDataDAO`: classe usata per comunicare tramite *Hibernate* con la tabella `UserData` della base di dati.

### Attributi

- - `{frozen} serialVersionUID: long` Attributo contenente l'id per la serializzazione dei dati. Il valore di creazione deve essere 10014L.

### Metodi

#### + `AddressBookDoRemoveFromGroupServlet()`

Costruttore pubblico della *servlet*. Richiama il costruttore della classe padre (chiamata a *super*).

#### # `doGet(request HttpServletRequest, response HttpServletResponse): void`

Metodo usato per eseguire la richiesta di rimuovere un contatto da un gruppo (eseguito in risposta ad una richiesta HTTP con metodo GET). Poiché si intende gestire in modo univoco una richiesta a tale *servlet* (indifferentemente dalla tipologia d'invio dati) il metodo reindirizza il flusso principale al metodo `doPost()`.

#### # `doPost(request HttpServletRequest, response HttpServletResponse): void`

Metodo usato per eseguire la richiesta di rimuovere un contatto da un gruppo (eseguito in risposta ad una richiesta HTTP con metodo POST). Nell'ordine proposto, devono essere eseguite le seguenti operazioni:

- creazione degli oggetti necessari al completamento dell'operazione:
  - `GroupDAO groupDAO;`
  - `HttpSession session;`
  - `IAddressBookEntry entry;`
  - `IGroup group;`
  - `IUserData friend;`
  - `IUserData user; Long contactId; Long groupId;`
  - `PrintWriter writer; String result;`
  - `UserDataDAO userDAO;`

- quindi il metodo procede definendo un blocco *try-catch*. Nel ramo *catch* si imposta *result* a *false*. Passando invece alla definizione del ramo *try*, in esso devono essere definiti i seguenti punti:
  - session viene impostato mediante una chiamata a metodo: `request.getSession(false);`
  - user e groupID vengono impostate ottenendo l'omonimo attributo da *request* (usare il metodo `getParameter(nome_parametro);`
  - impostare friend e group a partire dai relativi tipi di istanze DAO e facendosi restituire i dati presenti nel database tramite una chiamata a metodo `getByID` a cui passa i relativi id (`contactId` e `groupid`);
  - avvia un costrutto condizionale dotato di ramo *else*, se `group!=null` allora il metodo procede nel ramo *if* andando ad impostare *entry* con i relativi parametri, in modo da ricreare l'istanza `IAddressBookEntry` da rimuovere dal database per mezzo del metodo `removeAddressBookEntry()` a cui passa *entry*.
  - altrimenti, se `group == null` allora entra nel ramo *else* del costrutto condizionale e procede impostando *result* a *false*.

Il metodo termina scrivendo su *writer* il contenuto di *result*.

#### 4.3.7 AddressBookDoBlockServlet

##### Funzione

*Servlet* richiamata dal *client* per bloccare un contatto presente nella propria rubrica.

##### Relazioni d'uso

- `java.io.IOException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `java.io.PrintWriter`: classe istanziata all'interno del metodo `doPost()`. Usata per scrivere l'output della *servlet*.
- `javax.servlet.ServletException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `javax.servlet.http.HttpServlet`: classe estesa da `LoginManager`.
- `javax.servlet.http.HttpServletRequest`: classe per usata per la comunicazione con la *servlet* (inoltre i dati in ingresso alla *servlet*).
- `javax.servlet.http.HttpServletResponse`: classe per usata per la comunicazione con la *servlet* (inoltre i dati in uscita dalla chiamata a *servlet*).
- `javax.servlet.http.HttpSession`: classe usata per definire una sessione HTTP.
- `abook.IAddressBookEntry`: interfaccia che definisce il comportamento di una generica *entry* della rubrica utente. La classe che verrà descritta crea istanze di tipo `abook.IAddressBookEntry`.
- `abook.IUserData`: interfaccia che definisce il comportamento di un utente generico. La classe che verrà descritta crea istanze di tipo `abook.IUserData`.
- `dao.UserDataDAO`: classe usata per comunicare tramite *Hibernate* con la tabella `UserData` della base di dati.

##### Attributi

- `- {frozen} serialVersionUID: long` Attributo contenente l'id per la serializzazione dei dati, il valore di creazione deve essere 10014L.

## Metodi

### + AddressBookDoRemoveFromGroupServlet()

Costruttore pubblico della *servlet*. Richiama il costruttore della classe padre (chiamata a *super*).

### # doGet(request HttpServletRequest, response HttpServletResponse): void

Metodo usato per eseguire la richiesta di bloccare un contatto presente in una rubrica utente (eseguito in risposta ad una richiesta HTTP con metodo GET). Poiché si intende gestire in modo univoco una richiesta a tale *servlet* (indifferentemente dalla tipologia d'invio dati) il metodo reindirizza il flusso principale al metodo `doPost()`.

### # doPost(request HttpServletRequest, response HttpServletResponse): void

Metodo usato per eseguire la richiesta di bloccare un contatto presente in una rubrica utente (eseguito in risposta ad una richiesta HTTP con metodo POST). Nell'ordine proposto, devono essere eseguite le seguenti operazioni:

- creazione degli oggetti necessari al completamento dell'operazione:
  - GroupDAO groupDAO;
  - HttpSession session;
  - IAddressBookEntry entry;
  - IUserData friend;
  - IUserData user; Long contactId;
  - Iterator<IAddressBookEntry> iterator;
  - Set<IAddressBookEntry> entrys;
  - PrintWriter writer; String result;
  - UserDataDAO userDAO;
- quindi il metodo procede definendo un blocco *try-catch*. Nel ramo *catch* si imposta *result* a *false*. Passando invece alla definizione del ramo *try* devono essere definiti i seguenti punti:
  - *session* viene impostato mediante una chiamata a metodo: `request.getSession(false)`;
  - *user* viene impostato ottenendo l'omonimo attributo da *request* (usare il metodo `getParameter("user")`);
  - impostare *friend* a partire dal relativo tipo di istanza DAO e facendo restituire i dati presenti nel database tramite una chiamata a metodo `getByID` a cui passa il parametro *contactId*;
  - avvia un costrutto condizionale dotato di ramo *else*. Se *friend* != null il metodo procede nel ramo *if* andando a scaricare l'elenco delle *entry* a partire da *user*. Quindi utilizzando l'iteratore *iterator* va a scorrere tale set di *entry* e quando verifica la presenza del contatto *friend* all'interno, modifica l'*entry* attualmente selezionata richiamando il metodo `setBlocked(true)`. Quindi prima di uscire dal costrutto *if*, viene eseguita un'operazione di *update* a partire da *user* e si memorizza nella variabile *result* il valore *true*.
  - altrimenti, se *friend* == null allora entra nel ramo *else* del costrutto condizionale e procede impostando *result* a *false*.

Il metodo termina scrivendo su *writer* il contenuto di *result*.

## 4.3.8 AddressBookDoUnblockServlet

### Funzione

*Servlet* richiamata dal *client* per sbloccare un contatto presente nella propria rubrica.

## Relazioni d'uso

- `java.io.IOException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `java.io.PrintWriter`: classe istanziata all'interno del metodo `doPost()`. Usata per scrivere l'*output* della *servlet*.
- `javax.servlet.ServletException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `javax.servlet.http.HttpServlet`: classe estesa da `LoginManager`.
- `javax.servlet.http.HttpServletRequest`: classe per usata per la comunicazione con la *servlet* (inoltra i dati in ingresso alla *servlet*).
- `javax.servlet.http.HttpServletResponse`: classe per usata per la comunicazione con la *ingreseservlet* (inoltra i dati in uscita dalla chiamata a *servlet*).
- `javax.servlet.http.HttpSession`: classe usata per definire una sessione HTTP.
- `abook.IAddressBookEntry`: Interfaccia che definisce il comportamento di una generica *entry* della rubrica utente. La classe che verrà descritta crea istanze di tipo `abook.IAddressBookEntry`.
- `abook.IUserData`: Interfaccia che definisce il comportamento di un utente generico. La classe che verrà descritta crea istanze di tipo `abook.IUserData`.
- `dao.UserDataDAO`: classe usata per comunicare tramite *Hibernate* con la tabella `UserData` della base di dati.

## Attributi

- - `{frozen} serialVersionUID: long` Attributo contenente l'id per la serializzazione dei dati, il valore di creazione deve essere 10019L.

## Metodi

### + `AddressBookDoUnblockServlet()`

Costruttore pubblico della *servlet*, richiama il costruttore della classe padre (chiamata a `super`).

### # `doGet(request HttpServletRequest, response HttpServletResponse): void`

Metodo usato per eseguire la richiesta di sbloccare un contatto presente in una rubrica utente (eseguito in risposta ad una richiesta HTTP con metodo `GET`). Poiché si intende gestire in modo univoco una richiesta a tale *servlet* (indifferentemente dalla tipologia d'invio dati) il metodo reindirizza il flusso principale al metodo `doPost()`.

### # `doPost(request HttpServletRequest, response HttpServletResponse): void`

Metodo usato per eseguire la richiesta di sbloccare un contatto presente in una rubrica utente (eseguito in risposta ad una richiesta HTTP con metodo `POST`). Nell'ordine proposto, devono essere eseguite le seguenti operazioni:

- creazione degli oggetti necessari al completamento dell'operazione:
  - `GroupDAO groupDAO;`
  - `HttpSession session;`
  - `IAddressBookEntry entry;`
  - `IUserData friend;`
  - `IUserData user; Long contactIdl;`
  - `Iterator<IAddressBookEntry> iterator;`

- `Set<IAddressBookEntry> entrys;`
- `PrintWriter writer;` `String result;`
- `UserDataDAO userDAO;`
- quindi il metodo procede definendo un blocco *try-catch*. Nel ramo *catch* si imposta *result* a *false*. Passando invece alla definizione del ramo *try*, in esso devono essere definiti i seguenti punti:
  - *session* viene impostato mediante una chiamata a metodo: `request.getSession(false);`
  - *user* viene impostata ottenendo l'omonimo attributo da *request* (usare il metodo `getParameter("user")`);
  - impostare *friend* a partire dal relativo tipo di istanza DAO e facendo restituire i dati presenti nel database tramite una chiamata a metodo `getByID` a cui passa il parametro *contactId*;
  - avvia un costrutto condizionale dotato di ramo *else*. Se `friend!=null` il metodo procede nel ramo *if* andando a scaricare l'elenco delle *entry* a partire da *user*. Quindi utilizzando l'iteratore *iterator* va a scorrere tale set di *entry* e quando verifica la presenza del contatto *friend* all'interno, modifica l'*entry* attualmente selezionata richiamando il metodo `setBlocked(false)`. Quindi prima di uscire dal costrutto *if*, viene eseguita un'operazione di *update* a partire da *user* e si memorizza nella variabile *result* il valore *true*.
  - altrimenti, se `friend == null` allora entra nel ramo *else* del costrutto condizionale e procede impostando *result* a *false*.

Il metodo termina scrivendo su *writer* il contenuto di *result*.

#### 4.3.9 AddressBookDoSearchServlet

##### Funzione

*Servlet* richiamata dal *client* per effettuare una ricerca sulla propria rubrica. La ricerca consiste nell'individuare tutti i contatti che contengono nei campi *name*, *surname* o *email* la parola ricercata. Per eseguire questa procedura di ricerca ci si avvale del metodo `searchGeneric(nome_parametro)` della classe `UserDataDAO`.

##### Relazioni d'uso

- `java.io.IOException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `java.io.PrintWriter`: classe istanziata all'interno del metodo `doPost()`. Usata per scrivere l'output della *servlet*.
- `java.util.List`: usata per memorizzare la lista di `IUserData` da restituire in seguito alla richiesta di ricerca.
- `javax.servlet.ServletException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `javax.servlet.http.HttpServlet`: classe estesa da `LoginManager`.
- `javax.servlet.http.HttpServletRequest`: classe per usata per la comunicazione con la *servlet* (inoltre i dati in ingresso alla *servlet*).
- `javax.servlet.http.HttpServletResponse`: classe per usata per la comunicazione con la *servlet* (inoltre i dati in uscita dalla chiamata a *servlet*).
- `javax.servlet.http.HttpSession`: classe usata per definire una sessione HTTP.

- `abook.IUserData`: Interfaccia che definisce il comportamento di un utente generico. La classe che verrà descritta crea istanze di tipo `abook.IUserData`.
- `dao.UserDataDAO`: classe usata per comunicare tramite *Hibernate* con la tabella `UserData` della base di dati.

### Attributi

- `- {frozen} serialVersionUID: long` Attributo contenente l'id per la serializzazione dei dati, il valore di creazione deve essere 100110L.

### Metodi

#### + `AddressBookDoSearchServlet()`

Costruttore pubblico della *servlet*, richiama il costruttore della classe padre (chiamata a *super*).

#### # `doGet(request HttpServletRequest, response HttpServletResponse): void`

Metodo usato per eseguire una ricerca generica sui contatti presenti nella rubrica del *client* (eseguito in risposta ad una richiesta HTTP con metodo `GET`). Poiché si intende gestire in modo univoco una richiesta a tale *servlet* (indifferentemente dalla tipologia d'invio dati) il metodo reindirizza il flusso principale al metodo `doPost()`.

#### # `doPost(request HttpServletRequest, response HttpServletResponse): void`

Metodo usato per eseguire una ricerca generica sui contatti presenti nella rubrica del *client* (eseguito in risposta ad una richiesta HTTP con metodo `POST`). Nell'ordine proposto, devono essere eseguite le seguenti operazioni:

- creazione degli oggetti necessari al completamento dell'operazione:
  - `IUserData` entry;
  - `List<IUserData>` users;
  - `Iterator<IUserData>` iterator;
  - `Set<IAddressBookEntry>` entrys;
  - `PrintWriter` writer;
  - `String` result e `String` parameter;
  - `UserDataDAO` userDAO;
- quindi il metodo procede definendo un blocco *try-catch*. Nel ramo *catch* si imposta *result* a *false*. Passando invece alla definizione del ramo *try*, in esso devono essere definiti i seguenti punti:
  - *parameter* viene impostato con il valore del parametro da ricercare. Tale operazione è da eseguirsi con l'istruzione:  
`request.getParameter("param");`
  - quindi il metodo procede effettuando la ricerca del parametro. Tale operazione si svolge sfruttando le specifiche della classe `UserDataDAO`. Nello specifico deve essere eseguita l'istruzione:  
`users = userDAO.searchGeneric(parameter);`
  - a questo punto si dovrà predisporre un iteratore per scorrere la lista *users*;
  - si imposta *result* al valore `""` e si entra in un ciclo *while* che potrà terminare solo quando l'iteratore già citato avrà raggiunto l'ultimo elemento ispezionabile;
  - dentro al ciclo *while* si procederà con la formattazione della stringa *result* al fine di restituire al *client* un formato sul quale possa operare. Nello specifico la

stringa *result* dovrà contenere (per ogni istanza di *IUserData* presente in *users*):

```
\ "ID_della_entry": {"name": "NOME_UTENTE",
"surname": "COGNOME_UTENTE", "email": "EMAIL_UTENTE",
"id": "ID_UTENTE", "picturePath": "PATH_IMG",
"state": "STATO", "block": "BLOCCATO/SBLOCCATO"},
```

Si osservi che in quanto è riportato in precedenza, ciò che è scritto in maiuscolo corrisponde al valore effettivo di quel parametro, quindi per esempio la parola *NOME\_UTENTE* sarà di fatto sostituita dall'effettivo nome dell'istanza *IUserData* attualmente sotto esame. Per ottenere tali valori ricorrerà ai metodi: *getName()*, *getSurname()*, *getMail()*, *getId()*, *getPath()* richiamabili a partire dall'*entry* attualmente sotto esame.

- al termine di tale procedura la stringa *result* è pronta per essere restituita al *client*, quindi si aggiunge a *result* il valore `""`, usato come carattere di terminazione.

Il metodo termina scrivendo su *writer* il contenuto di *result*.

#### 4.3.10 AddressBookGetContactsServlet

##### Funzione

*Servlet* richiamata dal *client* per scaricare la lista dei contatti presenti nella propria rubrica.

##### Relazioni d'uso

- `java.io.IOException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `java.io.PrintWriter`: classe istanziata all'interno del metodo `doPost()`. Usata per scrivere l'output della *servlet*.
- `java.util.Set`: usata per memorizzare l'insieme di *IUserData* da restituire in seguito alla richiesta di ricerca.
- `javax.servlet.ServletException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `javax.servlet.http.HttpServlet`: classe estesa da `LoginManager`.
- `javax.servlet.http.HttpServletRequest`: classe per usata per la comunicazione con la *servlet* (inoltra i dati in ingresso alla *servlet*).
- `javax.servlet.http.HttpServletResponse`: classe per usata per la comunicazione con la *servlet* (inoltra i dati in uscita dalla chiamata a *servlet*).
- `javax.servlet.http.HttpSession`: classe usata per definire una sessione HTTP.
- `abook.IAddressBookEntry`: interfaccia che definisce il comportamento di una *entry* della rubrica utente.
- `abook.IUserData`: interfaccia che definisce il comportamento di un utente generico. La classe che verrà descritta crea istanze di tipo `abook.IUserData`.
- `dao.UserDataDAO`: classe usata per comunicare tramite Hibernate con la tabella *UserData* della base di dati.



## Attributi

- - `{frozen} serialVersionUID: long` Attributo contenente l'id per la serializzazione dei dati, il valore di creazione deve essere 10010L.

## Metodi

### + `AddressBookGetContactsServlet()`

Costruttore pubblico della *servlet*. Richiama il costruttore della classe padre (chiamata a *super*).

### # `doGet(request HttpServletRequest, response HttpServletResponse): void`

Metodo usato per eseguire la richiesta di download della lista dei contatti utente (eseguito in risposta ad una richiesta HTTP con metodo GET). Poiché si gestisce in modo univoco una richiesta a tale *servlet* (indifferentemente dalla tipologia d'invio dati) il metodo reindirizza il flusso principale al metodo `doPost()`.

### # `doPost(request HttpServletRequest, response HttpServletResponse): void`

Metodo usato per eseguire la richiesta di download della lista dei contatti (eseguito in risposta ad una richiesta HTTP con metodo POST). Nell'ordine proposto, devono essere eseguite le seguenti operazioni:

- creazione degli oggetti necessari al completamento dell'operazione:
  - `HttpSession session;`
  - `IUserData user;`
  - `IUserData friend;`
  - `Iterator<IAddressBookEntry> iterator;`
  - `Set<IAddressBookEntry> entrys;`
  - `PrintWriter writer;`
  - `String result;`
  - `IAddressBookEntry entry;`
- quindi il metodo procede definendo un blocco *try-catch*. Nel ramo *catch* si imposta *result* a *null*. Passando invece alla definizione del ramo *try*, devono essere definiti i seguenti punti:
  - session viene impostata tramite una chiamata a metodo del tipo:
 

```
session = request.getSession(false);
```
  - quindi il metodo procede impostando user a partire dalla sessione precedentemente creata. Tale operazione richiede l'uso del metodo `getAttribute("user");`
  - viene inizializzato l'insieme *contacts* tramite una chiamata `getAddressBook()` richiamata a partire dalla variabile *user*;
  - a questo punto si dovrà predisporre un iteratore per scorrere l'insieme *contacts*;
  - si imposta *result* al valore "" e si entra in un ciclo *while* che potrà terminare solo quando l'iteratore già citato avrà raggiunto l'ultimo elemento ispezionabile;
  - dentro al ciclo *while*, dopo aver impostato *entry* al valore `next()` dell'iteratore, viene salvato in friend il contatto ritornato da una chiamata a `entry.getContact()`. Si procederà poi con la formattazione della stringa result al fine di restituire al *client* un formato sul quale possa operare. Nello specifico la stringa result dovrà contenere (per ogni istanza di IUserData presente in users):

```
\ "ID_della_entry": {"name": "NOME_UTENTE",
"surname": "COGNOME_UTENTE", "email": "EMAIL_UTENTE",
"id": "ID_UTENTE", "picturePath": "PATH_IMG",
```



```
"state": "STATO", "block": "BLOCCATO/SBLOCCATO"}},
```

Si osservi che in quanto è riportato in precedenza, ciò che è scritto in maiuscolo corrisponde al valore effettivo di quel parametro. Quindi per esempio la parola NOME\_UTENTE sarà di fatto sostituita dall'effettivo nome dell'istanza `IUserData` attualmente sotto esame. Per ottenere tali valori si ricorre ai metodi: `getName()`, `getSurname()`, `getMail()`, `getId()`, `getPath()` richiamabili a partire da `friend`.

- al termine di tale procedura la stringa *result* è pronta per essere restituita al *client*. Quindi si aggiunge a *result* il valore `"`, usato come carattere di terminazione.

Il metodo termina scrivendo su *writer* il contenuto di *result*.

#### 4.3.11 AddressBookGetGroupsServlet

##### Funzione

*Servlet* richiamata dal *client* per scaricare la propria rubrica, provvista di gruppi e contatti in essi presenti.

##### Relazioni d'uso

- `java.io.IOException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `java.io.PrintWriter`: classe istanziata all'interno del metodo `doPost()`. Usata per scrivere l'output della *servlet*.
- `java.util.List`: usata per memorizzare l'insieme di `IGroup` da restituire in seguito alla richiesta di ricerca.
- `java.util.Set`: usata per memorizzare l'insieme di `IAddressBookEntry` da restituire in seguito alla richiesta di ricerca.
- `javax.servlet.ServletException`: eccezione lanciabile dai metodi `doPost()` e `doGet()`.
- `javax.servlet.http.HttpServlet`: classe estesa da `LoginManager`.
- `javax.servlet.http.HttpServletRequest`: classe per usata per la comunicazione con la *servlet* (inoltre i dati in ingresso alla *servlet*).
- `javax.servlet.http.HttpServletResponse`: classe per usata per la comunicazione con la *servlet* (inoltre i dati in uscita dalla chiamata a *servlet*).
- `javax.servlet.http.HttpSession`: classe usata per definire una sessione HTTP.
- `abook.IAddressBookEntry`: interfaccia che definisce il comportamento di una *entry* della rubrica utente.
- `abook.IUserData`: interfaccia che definisce il comportamento di un utente generico. La classe che verrà descritta crea istanze di tipo `abook.IUserData`.
- `abook.IGroup`: interfaccia che definisce il comportamento di un gruppo generico. La classe che verrà descritta crea istanze di tipo `abook.IGroup`.
- `dao.GroupDAO`: classe usata per comunicare tramite *Hibernate* con la tabella `Group` della base di dati.

## Attributi

- - `{frozen} serialVersionUID: long` Attributo contenente l'id per la serializzazione dei dati. Il valore di creazione deve essere 10010L.

## Metodi

### + `AddressBookGetContactsServlet()`

Costruttore pubblico della *servlet*. Richiama il costruttore della classe padre (chiamata a *super*).

### # `doGet(request HttpServletRequest, response HttpServletResponse): void`

Metodo usato per eseguire la richiesta di download della rubrica di un utente (eseguito in risposta ad una richiesta HTTP con metodo `GET`). Poiché si intende gestire in modo univoco una richiesta a tale *servlet* (indifferentemente dalla tipologia d'invio dati) il metodo reindirizza il flusso principale al metodo `doPost()`.

### # `doPost(request HttpServletRequest, response HttpServletResponse): void`

Metodo usato per eseguire la richiesta di download della lista dei contatti utente (eseguito in risposta ad una richiesta HTTP con metodo `POST`). Nell'ordine proposto, devono essere eseguite le seguenti operazioni:

- creazione degli oggetti necessari al completamento dell'operazione:
  - `HttpSession session;`
  - `GroupDAO groupDAO;`
  - `IAddressBookEntry entry;`
  - `IGroup group;`
  - `IUserData user;`
  - `Iterator<IAddressBookEntry> entryIter;`
  - `Iterator<IGroup> groupIter;`
  - `Set<IAddressBookEntry> addentrys;`
  - `PrintWriter writer;`
  - `String result;`
- quindi il metodo procede definendo un blocco *try-catch*. Nel ramo *catch* si imposta *result* a *false*. Passando invece alla definizione del ramo *try*, in esso devono essere definiti i seguenti punti:
  - session viene impostata tramite una chiamata a metodo del tipo:
 

```
session = request.getSession(false);
```
  - quindi il metodo procede impostando user a partire dalla sessione precedentemente creata. Tale operazione richiede l'uso del metodo `getAttribute("user")`;
  - viene inizializzato l'oggetto groupDAO;
  - si inizializza la lista groups dei gruppi. Tale operazione deve essere eseguita mediante l'istruzione:
 

```
groupDAO.getByOwner(user.getId())
```
  - il metodo definisce un costrutto condizionale *if-else* basato sulla condizione `groups != null`. Nel ramo *else* il metodo non fa altro che impostare *result* a *false*. Nel ramo *if* saranno definite le seguenti istruzioni:
    - \* predisporre un iteratore per scorrere l'insieme groups (inizializzazione di `groupIter`);
    - \* si imposta *result* al valore `"{"` e si entra in un ciclo *while* che potrà terminare solo quando l'iteratore già citato avrà raggiunto l'ultimo elemento ispezionabile;

- \* all'interno del ciclo si estrae dall'iteratore il gruppo attualmente in esame e si salva sull'oggetto `group`;
- \* a partire da `group`, tramite una chiamata `getAddressBook()`, si ottiene la lista di `entry` di quel gruppo e la si salva in `addEntry`;
- \* si definisce un nuovo iteratore `addEntry.iterator()` e lo si memorizza in `entryIter`;
- \* si "scrive" dentro a `result`, i dati inerenti il gruppo in esame (`name`, `id`) e si aggiunge la stringa `"contacts:"`;
- \* si entra in un nuovo ciclo `while` che cicla sulle `entry` dell'iteratore `entryIter`. All'interno di tale ciclo si estraggono le `entry` che costituiscono i contatti del gruppo in esame e si riporta in `result` l'id di tali contatti (usare metodo `getId()`);
- \* all'uscita del ciclo `while` più interno si concatena al contenuto di `result` la stringa `"}"`;
- \* all'uscita del ciclo `while` più esterno si concatena al contenuto di `result` la stringa `"}"`;

Il metodo termina scrivendo su `writer` il contenuto di `result`.

## 4.4 Package `org.softwaresynthesis.mytalk.server.call`

### 4.4.1 ICall

#### Funzione

Interfaccia che rappresenta una chiamata effettuata dal sistema MyTalk. Gli oggetti che implementano tale interfaccia vengono usati per rappresentare lo storico delle chiamate di un utente.

#### Relazioni d'uso

- `java.util.Date`: tipo utilizzato per definire la data d'inizio e fine di una chiamata.

#### Metodi

- + `getId(): Long`  
Restituisce l'identificativo univoco della chiamata.
- + `getStartDate(): Date`  
Restituisce un'istanza di `java.util.Date` di avvio della chiamata.
- + `setStartDate(startDate: Date): void`  
Imposta la data di avvio della chiamata.
- + `getEndDate(): Date`  
Restituisce la data in cui termina la chiamata
- + `setEndDate(endDate: Date): void`  
Imposta la data in cui termina la chiamata.

### 4.4.2 Call

#### Funzione

Classe che implementa l'interfaccia `ICall`.

### Relazioni d'uso

- `java.util.Date`: tipo utilizzato per definire la data d'inizio e fine di una chiamata.

### Attributi

- - `id`: `long` Attributo contenente il codice identificativo della chiamata.
- - `startDate`: `Date` Attributo di tipo `java.util.Date` contenente la data (compresa l'ora) d'inizio della chiamata.
- - `endDate`: `Date` Attributo di tipo `java.util.Date` contenente la data (compresa l'ora) in cui la chiamata è terminata.

### Metodi

- + `getId(): Long`  
Restituisce l'identificativo univoco della chiamata ritornando il contenuto dell'attributo `id`.
- + `getStartDate(): Date`  
Restituisce il contenuto del campo `startDate`.
- + `setStartDate(startDate: Date): void`  
Imposta la data di avvio della chiamata, sovrascrivendo il contenuto `startData`.
- + `getEndDate(): Date`  
Restituisce il contenuto del campo `endDate`.
- + `setEndDate(endDate: Date): void`  
Imposta la data di avvio della chiamata, sovrascrivendo il contenuto `endData`.

#### 4.4.3 ICallList

##### Funzione

Interfaccia rappresentante una *entry* di uno storico chiamate di un utente del sistema mytalk.

### Relazioni d'uso

- `org.softwaresynthesis.mytalk.server.IMyTalkObject`: interfaccia da estendere. Ogni oggetto che implementerà l'interfaccia `ICallList` dovrà essere in grado di convertire il proprio contenuto informativo in formato *Json*.
- `IUserData`: l'interfaccia `ICallList` definisce più metodi che restituiscono oggetti aventi tipo di ritorno `IUserData`, essi sono i metodi *get* per ottenere l'utente che ha effettuato la chiamata. Analogamente `IUserData` viene usato come parametro d'ingresso per i metodi *set* collegati ai metodi già citati.
- `ICall`: l'interfaccia `ICallList` definiscono metodi che restituiscono oggetti aventi tipo di ritorno `ICall`.

## Metodi

- + `getId(): Long`  
Restituisce l'identificativo univoco di una *entry* di uno storico chiamate.
- + `setIdCall(call: ICall): void`  
Imposta l'id della chiamata `ICall` (passato come parametro d'ingresso).
- + `getIdCall(): IUserData`  
Restituisce un id di un oggetto avente tipo `ICall` rappresentante una chiamata effettuata dall'utente.
- + `setIdUser(contact: IUserData): void`  
Imposta l'id dell'utente `IUserData` (passato come parametro d'ingresso) come l'utente che ha effettuato la chiamata.
- + `getIdUser(): IUserData`  
Restituisce l'id di oggetto avente tipo `IUserData` rappresentante l'utente che ha effettuato la chiamata.
- + `setCaller(caller: boolean): void`  
Imposta l'utente che ha effettuato la chiamata come chiamante se il parametro ricevuto ha valore true.
- + `getCaller(): boolean`  
Ritorna un valore di tipo bool se l'utente della chiamata è colui che l'ha iniziata o meno.

### 4.4.4 CallList

#### Funzione

Classe che implementa l'interfaccia `ICallList`.

#### Relazioni d'uso

- `call.ICallList`: interfaccia implementata dalla classe.

#### Attributi

- - `id: long` Attributo contenente il codice identificativo dello storico chiamata.
- - `idCall: long` Attributo contenente il codice identificativo della chiamata.
- - `idUser: long` Attributo contenente il codice identificativo dell'utente partecipante alla chiamata.
- - `caller: boolean` Attributo che identifica se l'utente è il chiamante o meno.

## Metodi

- + `getId(): Long`  
Restituisce il valore dell'attributo `id`.
- + `setIdCall(call: Long): void`  
Imposta il valore dell'attributo `idCall` con il valore ricevuto da parametro "call".
- + `getIdCall(): Long`  
Restituisce il valore dell'attributo `idCall`.

```
+ setIdUser(contact: Long): void
    Imposta il valore dell'attributo idUser con il valore ricevuto da parametro "contact".

+ getIdUser(): Long
    Restituisce il valore dell'attributo idUser.

+ setCaller(caller: boolean): void
    Imposta il valore dell'attributo caller con il valore ricevuto da parametro "caller".

+ getCaller(): boolean
    Restituisce il valore dell'attributo caller.
```

## 4.5 Package org.softwaresynthesis.mytalk.server.call.servlet

### 4.5.1 DownloadCallHistoryServlet

#### Funzione

*Servlet* da richiamare per effettuare il download della lista che rappresenta lo storico delle chiamate effettuate e ricevute dell'utente. I dati vengono ritornati sotto forma di stringa in formato JSON.

#### Relazioni d'uso

- `java.io.IOException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `java.io.PrintWriter`: classe istanziata all'interno del metodo `doPost()`. Usata per scrivere l'output della *servlet*.
- `javax.servlet.ServletException`: eccezione rilancibile dai metodi `doPost()` e `doGet()`.
- `javax.servlet.http.HttpServlet`: classe estesa da `DownloadCallHistoryServlet`.
- `javax.servlet.http.HttpServletRequest`: classe per usata per la comunicazione con la *servlet* (inoltra i dati in ingresso alla *servlet*).
- `javax.servlet.http.HttpServletResponse`: classe per usata per la comunicazione con la *servlet* (inoltra i dati in uscita dalla chiamata a *servlet*).
- `javax.servlet.http.HttpSession`: classe usata per definire una sessione HTTP.
- `call.ICall`: usata per definire un oggetto rappresentante una chiamata.
- `abook.IUserData`: usata per definire un utente generico, che ha partecipato alla chiamata.

#### Attributi

- `frozen serialVersionUID: long`  
 Utilizzato come ID univoco per identificare un oggetto serializzabile. Tale ID deve essere impostato al valore 10021L.

#### Metodi

```
+ DownloadCallHistoryServlet()
    Costruttore della classe. Richiama il costruttore della classe padre HttpServlet.

# doGet(request: HttpServletRequest, response: HttpServletResponse): void
    Metodo di delega che richiama doPost() passando gli stessi dati in input.
```

# `doPost(request: HttpServletRequest, response: HttpServletResponse): void`  
Metodo usato per accedere alla funzionalità della *servlet*. Il procedimento da definire passa attraverso 3 step essenziali:

- **Inizializzazione:** il metodo definisce ed apre una nuova sessione a partire dalla richiesta `request` passata come parametro;
- **Elaborazione dati:** il metodo interroga il database attraverso le classi DAO e per ogni oggetto `ICall` in cui si evidenzia che uno dei due partecipanti è il client (`IUserData`) che ha inoltrato la richiesta a tale *servlet*, viene aggiunta alla stringa `result` da ritornare, i dati di tale chiamata. Nello specifico la formattazione della stringa di ritorno, dovrà descrivere la seguente logica:

```
{name:"NOME_UTENTE",start="DATA_INIZIO",end="DATA_FINE"}
```

Si consideri che la dove vi è un nome in maiuscolo, vi dovrà essere il dato inerente estrapolato dall'istanza `ICall` presa in considerazione.

- **Restituzione risultato:** dopo aver elaborato i dati, la *servlet* dovrà scrivere nel `PrintWriter` associato alla `request`:
  - una stringa formattata come sopra, se l'utente ha effettuato o ricevuto almeno una chiamata;
  - null, altrimenti.

## 4.6 Package `org.softwaresynthesis.mytalk.server.message`

### 4.6.1 `IMessage`

#### Funzione

Interfaccia che rappresenta un messaggio di segreteria del sistema MyTalk.

#### Relazioni d'uso

- `java.util.Date`: tipo utilizzato per definire la data in cui è stato registrato un messaggio.
- `abook.IUserData`: usata per rappresentare il mittente e il destinatario del messaggio di segreteria.

#### Metodi

- + `getId(): Long`  
Restituisce l'identificativo univoco del messaggio.
- + `getSender(): IUserData`  
Restituisce un istanza di tipo `abook.IUserData` che rappresenta il mittente del messaggio.
- + `setSender(sender: IUserData): void`  
Imposta il mittente del messaggio.
- + `getReceiver(): IUserData`  
Restituisce un istanza di tipo `abook.IUserData` che rappresenta il destinatario del messaggio.
- + `setReceiver(receiver: IUserData): void`  
Imposta il destinatario del messaggio.

**+ isNew(): boolean**

Restituisce un valore booleano che identifica lo stato del messaggio (“già letto” se il valore ritornato è *true*, “da leggere” se il valore ritornato è *false*).

**+ setNew(status: boolean): void**

Imposta il messaggio come “già ascoltato” o come “da ascoltare”.

**+ isVideo(): boolean**

Restituisce un booleano che determina se si tratta di un messaggio audio oppure audio-video.

**+ setVideo(video: boolean): void**

Imposta il messaggio come messaggio audio-video o come messaggio contenente solamente una traccia audio.

**+ getDate(): Date**

Restituisce la data in cui il mittente ha lasciato il messaggio nella segreteria del destinatario.

**+ setDate(date: Date): void**

Imposta la data in cui il mittente ha lasciato il messaggio nella segreteria del destinatario.

#### 4.6.2 Message

##### Funzione

Classe che implementa l'interfaccia `IMessage`.

##### Relazioni d'uso

- `java.util.Date`: tipo utilizzato per definire la data in cui è stato registrato un messaggio.
- `abook.IUserData`: usata per rappresentare il mittente e il destinatario del messaggio di segreteria.

##### Attributi

- - **id**: `long` Attributo contenente il codice identificativo del messaggio.
- - **sender**: `IUserData` Attributo contenente lo `IUserData` che rappresenta il destinatario del messaggio.
- - **receiver**: `IUserData` Attributo contenente lo `IUserData` che rappresenta il destinatario del messaggio.
- - **status**: `boolean` Attributo contenente un valore booleano che identifica se il messaggio è stato visionato/ascoltato o meno. L'attributo se impostato a *true*, designa il messaggio come *ascoltato*. Invece se impostato a *false* identifica il messaggio come *ancora da ascoltare*.
- - **video**: `boolean` Attributo che stabilisce se il messaggio contiene o meno una traccia video. Si ricorda che ogni messaggio contiene (di *default*) una traccia audio. Se tale attributo è impostato a *true* allora il messaggio contiene una traccia video.
- - **date**: `Date` Attributo che definisce l'orario di invio del messaggio.



## Metodi

- + `getId(): Long`  
Restituisce l'identificativo univoco del messaggio, ritornando il contenuto di `id`.
- + `getSender(): IUserData`  
Restituisce un istanza di tipo `classnameabook.IUserData` che rappresenta il mittente del messaggio. Nello specifico il metodo ritorna l'attributo `sender`.
- + `setSender(sender: IUserData): void`  
Imposta il mittente del messaggio, sovrascrivendo il contenuto dell'attributo `sender`.
- + `getReceiver(): IUserData`  
Restituisce un istanza di tipo `classnameabook.IUserData` che rappresenta il destinatario del messaggio. Nello specifico il metodo ritorna l'attributo `receiver`.
- + `setReceiver(receiver: IUserData): void`  
Imposta il destinatario del messaggio, sovrascrivendo il contenuto dell'attributo `receiver`.
- + `isNew(): boolean`  
Metodo che ritorna il contenuto dell'attributo `status`.
- + `setNew(status: boolean): void`  
Imposta il messaggio come "già ascoltato" o come "da ascoltare", sovrascrivendo il contenuto dell'attributo `status`.
- + `isVideo(): boolean`  
Restituisce il contenuto dell'attributo `video`.
- + `setVideo(video: boolean): void`  
Metodo usato per impostare la "natura" del messaggio, impostando il contenuto dell'attributo `video` mettendolo a `true` se il messaggio contiene una traccia video, oppure a `false` se non la contiene.
- + `getDate(): Date`  
Restituisce la data in cui il mittente ha lasciato il messaggio nella segreteria del destinatario, ritornando il contenuto dell'attributo `date`.
- + `setDate(date: Date): void`  
Imposta la data in cui il mittente ha lasciato il messaggio nella segreteria del destinatario, sovrascrivendo il contenuto dell'attributo `date`.

## 4.7 Package `org.softwaresynthesis.mytalk.server.message.servlet`

### 4.7.1 `InsertMessageServlet`

#### Funzione

*Servlet* da richiamare per inserire un messaggio nella segreteria di un utente.

#### Relazioni d'uso

- `java.io.IOException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `java.io.PrintWriter`: classe istanziata all'interno del metodo `doPost()`. Usata per scrivere l'output della *servlet*.
- `javax.servlet.ServletException`: eccezione rilancibile dai metodi `doPost()` e `doGet()`.

- `javax.servlet.http.HttpServlet`: classe estesa da `DownloadCallHistoryServlet`.
- `javax.servlet.http.HttpServletRequest`: classe per usata per la comunicazione con la *servlet* (inoltra i dati in ingresso alla *servlet*).
- `javax.servlet.http.HttpServletResponse`: classe per usata per la comunicazione con la *servlet* (inoltra i dati in uscita dalla chiamata a *servlet*).
- `javax.servlet.http.HttpSession`: classe usata per definire una sessione HTTP.
- `call.IMessage`: usata per definire un oggetto rappresentante un messaggio in segreteria.
- `abook.IUserData`: usata per definire un utente generico, che ha partecipato alla chiamata.

### Attributi

- `frozen serialVersionUID: long`

Utilizzato come ID univoco per identificare un oggetto serializzabile. Tale ID deve essere impostato al valore 10022L.

### Metodi

+ `InsertMessageServlet()`

Costruttore della classe. Richiama il costruttore della classe padre `HttpServlet`.

# `doGet(request: HttpServletRequest, response: HttpServletResponse): void`

Metodo di delega che richiama `doPost()` passando gli stessi dati in input.

# `doPost(request: HttpServletRequest, response: HttpServletResponse): void`

Metodo usato per accedere alla funzionalità della *servlet*. Il procedimento da definire passa attraverso 3 step essenziali:

- **Inizializzazione**: il metodo definisce ed apre una nuova sessione a partire dalla richiesta request passata come parametro;
- **Elaborazione dati**: il metodo procede creando un `IMessage` con i dati forniti dalla request e dopo aver aperto una `transaction` verso il database, esegue l'operazione `save()` sul relativo oggetto DAO. Al termine sarà necessario procedere con un operazione di `commit`. Nel realizzare il metodo, il programmatore incaricato dovrà tenere in considerazione la possibilità di fallimento durante l'esecuzione delle operazioni mediante la `transaction`. Ciò andrà gestito mediante un operazione di `rollback` atta ad eliminare una possibile inconsistenza dei dati.
- **Restituzione risultato**: dopo aver elaborato i dati, la *servlet* dovrà scrivere nel `PrintWriter` associato alla request:
  - “*true*” se l'operazione è andata a buon fine;
  - “*false*” altrimenti.

#### 4.7.2 DeleteMessageServlet

##### Funzione

*Servlet* da richiamare per eliminare un messaggio nella segreteria dell'utente che la richiama.

## Relazioni d'uso

- `java.io.IOException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `java.io.PrintWriter`: classe istanziata all'interno del metodo `doPost()`. Usata per scrivere l'output della *servlet*.
- `javax.servlet.ServletException`: eccezione rilancibile dai metodi `doPost()` e `doGet()`.
- `javax.servlet.http.HttpServlet`: classe estesa da `DownloadCallHistoryServlet`.
- `javax.servlet.http.HttpServletRequest`: classe per usata per la comunicazione con la *servlet* (inoltre i dati in ingresso alla *servlet*).
- `javax.servlet.http.HttpServletResponse`: classe per usata per la comunicazione con la *servlet* (inoltre i dati in uscita dalla chiamata a *servlet*).
- `javax.servlet.http.HttpSession`: classe usata per definire una sessione HTTP.
- `call.IMessage`: usata per definire un oggetto rappresentante un messaggio in segreteria.
- `abook.IUserData`: usata per definire un utente generico, che ha partecipato alla chiamata.

## Attributi

- `frozen serialVersionUID: long`

Utilizzato come ID univoco per identificare un oggetto serializzabile. Tale ID deve essere impostato al valore 10022L.

## Metodi

+ `DeleteMessageServlet()`

Costruttore della classe. Richiama il costruttore della classe padre `HttpServlet`.

# `doGet(request: HttpServletRequest, response: HttpServletResponse): void`

Metodo di delega che richiama `doPost()` passando gli stessi dati in input.

# `doPost(request: HttpServletRequest, response: HttpServletResponse): void`

Metodo usato per accedere alla funzionalità della *servlet*. Il procedimento da definire passa attraverso 3 step essenziali:

- **Inizializzazione**: il metodo definisce ed apre una nuova sessione a partire dalla richiesta request passata come parametro;
- **Elaborazione dati**: il metodo procede creando un `IMessage` con i dati forniti dalla request e dopo aver aperto una `transaction` verso il database, esegue un'operazione di ricerca dell'istanza appena creata all'interno della lista di `IMessage` ottenuta dal db. Quindi se si verifica che tale istanza è effettivamente presente nel database, il metodo esegue l'operazione `delete()` sul relativo oggetto DAO. Al termine sarà necessario procedere con un'operazione di `commit`. Nel realizzare il metodo, il programmatore incaricato dovrà tenere in considerazione la possibilità di fallimento durante l'esecuzione delle operazioni mediante la `transaction`. Ciò andrà gestito mediante un'operazione di `rollback` atta ad eliminare una possibile inconsistenza dei dati.
- **Restituzione risultato**: dopo aver elaborato i dati, la *servlet* dovrà scrivere nel `PrintWriter` associato alla request:
  - “*true*” se l'operazione è andata a buon fine;

- “false” altrimenti (compreso non solo il caso di errore dovuto ad un problema di connessione verso il db, ma anche quello relativo al mancato match del elemento ricercato con quelli presenti nel database).

#### 4.7.3 UpdateStatusMessageServlet

##### Funzione

*Servlet* da richiamare per modificare lo stato di un messaggio. L'idea alla base è che un messaggio può trovarsi uno dei seguenti due stati:

- letto;
- non letto;

La *servlet* permette di effettuare una transizione di stato da “non letto” a “letto”.

##### Relazioni d'uso

- `java.io.IOException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `java.io.PrintWriter`: classe istanziata all'interno del metodo `doPost()`. Usata per scrivere l'output della *servlet*.
- `javax.servlet.ServletException`: eccezione rilancibile dai metodi `doPost()` e `doGet()`.
- `javax.servlet.http.HttpServlet`: classe estesa da `DownloadCallHistoryServlet`.
- `javax.servlet.http.HttpServletRequest`: classe per usata per la comunicazione con la *servlet* (inoltra i dati in ingresso alla *servlet*).
- `javax.servlet.http.HttpServletResponse`: classe per usata per la comunicazione con la *servlet* (inoltra i dati in uscita dalla chiamata a *servlet*).
- `javax.servlet.http.HttpSession`: classe usata per definire una sessione HTTP.
- `call.IMessage`: usata per definire un oggetto rappresentante un messaggio in segreteria.
- `abook.IUserData`: usata per definire un utente generico, che ha partecipato alla chiamata.

##### Attributi

- `frozen serialVersionUID: long`

Utilizzato come ID univoco per identificare un oggetto serializzabile. Tale ID deve essere impostato al valore 10023L.

##### Metodi

+ `UpdateStatusMessageServlet()`

Costruttore della classe. Richiama il costruttore della classe padre `HttpServlet`.

# `doGet(request: HttpServletRequest, response: HttpServletResponse): void`

Metodo di delega che richiama `doPost()` passando gli stessi dati in input.

# `doPost(request: HttpServletRequest, response: HttpServletResponse): void`

Metodo usato per accedere alla funzionalità della *servlet*. Il procedimento da definire passa attraverso 3 step essenziali:

- **Inizializzazione**: il metodo definisce ed apre una nuova sessione a partire dalla richiesta request passata come parametro;

- **Elaborazione dati:** il metodo procede creando un `IMessage` con i dati forniti dalla request e dopo aver aperto una `transaction` verso il database, esegue un'operazione di ricerca dell'istanza appena creata all'interno della lista di `IMessage` ottenuta dal db. Quindi se si verifica che tale istanza è effettivamente presente nel database, il metodo esegue l'operazione `update()` sul relativo oggetto DAO, andando a modificare lo stato in di lettura del messaggio. Al termine sarà necessario procedere con un'operazione di `commit`. Nel realizzare il metodo il programmatore incaricato dovrà tenere in considerazione la possibilità di fallimento durante l'esecuzione delle operazioni mediante la `transaction`. Ciò andrà gestito mediante un'operazione di `rollback` atta ad eliminare una possibile inconsistenza dei dati.
- **Restituzione risultato:** dopo aver elaborato i dati, la *servlet* dovrà scrivere nel `PrintWriter` associato alla request:
  - “*true*” se l'operazione è andata a buon fine;
  - “*false*” altrimenti.

#### 4.7.4 DownloadMessageListServlet

##### Funzione

*Servlet* da richiamare per scaricare la lista di messaggi attualmente presenti nella propria segreteria.

##### Relazioni d'uso

- `java.io.IOException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `java.io.PrintWriter`: classe istanziata all'interno del metodo `doPost()`. Usata per scrivere l'output della *servlet*.
- `javax.servlet.ServletException`: eccezione rilanciabile dai metodi `doPost()` e `doGet()`.
- `javax.servlet.http.HttpServlet`: classe estesa da `DownloadCallHistoryServlet`.
- `javax.servlet.http.HttpServletRequest`: classe per usata per la comunicazione con la *servlet* (inoltre i dati in ingresso alla *servlet*).
- `javax.servlet.http.HttpServletResponse`: classe per usata per la comunicazione con la *servlet* (inoltre i dati in uscita dalla chiamata a *servlet*).
- `javax.servlet.http.HttpSession`: classe usata per definire una sessione HTTP.
- `call.IMessage`: usata per definire un oggetto rappresentante un messaggio in segreteria.
- `abook.IUserData`: usata per definire un utente generico, che ha partecipato alla chiamata.

##### Attributi

– `frozen serialVersionUID: long`

Utilizzato come ID univoco per identificare un oggetto serializzabile. Tale ID deve essere impostato al valore 10024L.

## Metodi

### + DownloadMessageListServlet()

Costruttore della classe. Richiama il costruttore della classe padre `HttpServlet`.

### # doGet(request: HttpServletRequest, response: HttpServletResponse): void

Metodo di delega che richiama `doPost()` passando gli stessi dati in input.

### # doPost(request: HttpServletRequest, response: HttpServletResponse): void

Metodo usato per accedere alla funzionalità della *servlet*. Il procedimento da definire passa attraverso 3 step essenziali:

- **Inizializzazione:** il metodo definisce ed apre una nuova sessione a partire dalla richiesta request passata come parametro;
- **Elaborazione dati:** quindi esegue un'operazione di download degli oggetti `IMessage` il cui possessore è l'utente che ha fatto richiesta di *download*. in seguito a tale operazione, sarà necessario popolare e formattare la stringa di ritorno come segue: per ogni messaggio presente nella lista si dovrà riportare la dicitura:

```
{name:"NOME_UTENTE",state="LETTO/NON LETTO"  
 ,url="INDIRIZZO_MESSAGGIO",data="DATA_REGISTRAZIONE_MESSAGGIO"}
```

- **Restituzione risultato:** dopo aver elaborato i dati, la *servlet* dovrà scrivere nel `PrintWriter` associato alla request:
  - la sequenza
  - null, altrimenti.

## 4.8 Package org.softwaresynthesis.mytalk.server.authentication

### 4.8.1 ISecurityStrategy

#### Funzione

Interfaccia che identifica il comportamento di una strategia generica di crittografia dei dati.

#### Relazioni d'uso

Nessuna relazione d'uso evidenziata.

## Metodi

### + encrypt(plainText: String): String

Crypta la stringa di testo ricevuta come parametro.

Il metodo può lanciare eccezioni:

- **Exception:** il metodo lancia un'eccezione.

### + decrypt(encryptedText: String)

Decrypta la stringa di testo ricevuta come parametro.

Il metodo può lanciare eccezioni:

- **Exception:** il metodo lancia un'eccezione.

#### 4.8.2 AESAlgorithm

##### Funzione

Implementazione della strategia di codifica/decodifica con l'uso dell'algoritmo AES a 128bit.

##### Relazioni d'uso

- `ISecurityStrategy`: interfaccia d'implementazione
- `java.security.Key`: usata per creare un'istanza di una chiave durante il processo di criptazione.
- `javax.crypto.Cipher`: usata per creare un'istanza di un oggetto di criptazione che implementa l'algoritmo AES a 128 bit.
- `javax.crypto.spec.SecretKeySpec`: usata dall'algoritmo di per costruire una chiave segreta a partire da un array di byte.
- `sun.misc.BASE64Encoder`: utilizzata per eseguire una trasformazione da stringa a byte.
- `sun.misc.BASE64Decoder`: utilizzata per eseguire una trasformazione da byte in stringa.

##### Attributi

- `{frozen} keyValue: byte[]`  
Array di byte usato per definire il valore della chiave su cui si basa l'algoritmo AES di criptazione. La chiave effettiva sarà creata a partire da tale attributo, per mezzo della classe `javax.crypto.spec.SecretKeySpec`.
- `{frozen} algorithm: String`  
Stringa costante che identifica il nominativo dell'algoritmo usato, e che dovrà essere specificato durante l'uso di `javax.crypto.Cipher`. L'attributo avrà valore "AES".

##### Metodi

- `generateKey(): Key`  
Metodo che restituisce una chiave di tipo `java.security.Key`, a partire dall'array `keyValue`. Per fare ciò, il metodo usa il costruttore di `javax.crypto.spec.SecretKeySpec`. Il metodo può lanciare eccezioni:
  - **Exception**: il metodo può lanciare un'eccezione generica.
- + `encrypt(plainText: String): String`  
Metodo usato per criptare un testo di tipo `String` passato come parametro d'ingresso. Il metodo usa una chiave ottenuta a partire dal metodo `generateKey()` in associazione alla classe `javax.crypto.Cipher` per criptare il testo tramite l'algoritmo AES.
  - **Exception**: il metodo lancia un'eccezione.
- + `decrypt(encryptedText: String)`  
Decrypta la stringa di testo ricevuta come parametro, attuando una procedura inversa a quella presentata nel metodo `encrypt(plainText: String)`  
  
Il metodo può lanciare eccezioni:
  - **Exception**: il metodo può lanciare un'eccezione generica.

### 4.8.3 PrincipalImpl

#### Funzione

Oggetto che permette di identificare univocamente uno `IUserData` memorizzato nel database del sistema MyTalk.

#### Relazioni d'uso

- `java.io.Serializable`: interfaccia d'implementazione usata per rendere serializzabili le istanze della classe.
- `java.security.Principal`: interfaccia d'implementazione usata per rendere caratterizzate le istanze della classe.

#### Attributi

- `{frozen} serialVersionUID: long`  
Identificativo univoco per la classe, usato al fine di rendere l'oggetto serializzabile.
- `mail: String`  
Attributo che rappresenta l'indirizzo e-mail dell'utente.

#### Metodi

- + `PrincipalImpl(mail: String)`  
Classe costruttore. crea un oggetto `PrincipalImpl` che permette di determinare univocamente lo `IUserData` che ha effettuato il login.
- + `getName(): String`  
Restituisce l'elemento identificativo (mail dell'utente) dello `IUserData` che ha effettuato la procedura di login.
- + `equals(obj: Object): boolean`  
Verifica l'uguaglianza di due oggetti `PrincipalImpl` sulla base di un confronto tra gli indirizzi mail degli utenti confrontati.
- + `hashCode(): int`  
Restituisce il codice hash dell'oggetto di invocazione.
- + `toString(): String`  
Restituisce l'istanza dell'oggetto sotto forma di stringa. In particolare evidenziando l'indirizzo e-mail dell'utente.

### 4.8.4 IAuthenticationData

#### Funzione

Interfaccia che identifica il comportamento di un oggetto adatto alla definizione dei dati di autenticazione di un utente.

#### Relazioni d'uso

Nessuna relazione d'uso evidenziata.



**Metodi**

+ `getUsername(): String`

Restituisce lo username fornito in input dall'utente durante la procedura di login.

+ `getPassword(): String`

Restituisce la password fornita in input dall'utente durante la procedura di login

**4.8.5 AuthenticationData****Funzione**

Oggetto contenente i dati di accesso utilizzati da un utente che vuole accedere al sistema MyTalk.

**Relazioni d'uso**

- `IAuthenticationData`: interfaccia da implementare.

**Attributi**

- `username: String`

Attributo contenente l'username dell'utente che richiede l'autenticazione.

- `password: String`

Attributo contenente la password dell'utente che richiede l'autenticazione.

**Metodi**

+ `AuthenticationData(username: String, password: String)`

Costruttore pubblico che per creare un istanza di `AuthenticationData`, si basa sulla password e lo username dell'utente da autenticare.

+ `getUsername(): String`

Metodo che restituisce lo username dell'oggetto di autenticazione.

+ `getPassword(): String`

Metodo che Restituisce la password dell'utente che sta effettuando la procedura di login.

+ `hashCode(): int`

Restituisce il codice hash di questa istanza.

+ `equals(Object obj): boolean`

Determina l'uguaglianza di due istanze, effettuando un confronto tra l'oggetto stesso da cui è chiamato il metodo, e un `Object obj` ricevuto come parametro d'ingresso.

+ `toString: String`

Metodo che Restituisce l'istanza sotto forma di stringa. Si osservi che il metodo non dovrà ritornare la password, ma solo lo username dell'utente da autenticare.

**4.8.6 AuthenticationModule****Funzione**

Modulo di autenticazione utilizzato dal sistema MyTalk.

## Relazioni d'uso

- `java.io.IOException`: la classe è in grado di rilanciare eccezioni relative a operazioni di IO.
- `java.security.Principal`: cinterfaccia che definisce i dati di autenticazione di un utente.
- `java.util.Map`: parametri passati ad `iniziaile()`.
- `java.util.Set`: la classe può definire istanze di Set, per memorizzare i dati trattati.
- `javax.security.auth.callback.Callback`: vettore utilizzato per contenere i dati di autenticazione.
- `javax.security.auth.callback.CallbackHandler`: parametri passati ad `iniziaile()`.
- `javax.security.auth.callback.NameCallback`: oggetto che contiene il nome dell'utente da autenticare.
- `javax.security.auth.callback.PasswordCallback`: oggetto che contiene la password dell'utente da autenticare
- `javax.security.auth.callback.UnsupportedCallbackException`: la classe è in grado di rilanciare eccezioni relative a operazioni di `UnsupportedCallback`.
- `javax.security.auth.login.FailedLoginException`: la classe è in grado di rilanciare eccezioni relative a operazioni di `FailedLoginException`.
- `javax.security.auth.login.LoginException`: la classe è in grado di rilanciare eccezioni relative a operazioni di `LoginException`.
- `javax.security.auth.spi.LoginModule`: interfaccia d'implementazione.
- `javax.security.auth.Subject`: definisce il soggetto da autenticare.
- `abook.IUserData`: la classe interagisce con istanze di oggetti identificabili come utenti.
- `dao.UserDataDAO`: utilizzata per modificare il contenuto del database. Nello psecifico viene usata per interagire con la tabella `UserData`.

## Attributi

- `login: boolean`  
Attributo che determina se il login è avvenuto oppure no.
- `commit: boolean`  
Attributo che determina se il commit è già stato eseguito.
- `handler: CallbackHandler`  
Oggetto utilizzato per il caricamento delle credenzialie.
- `password: char[]`  
Attributo che che contiene la password immessa dall'utente.
- `username: String`  
Attributo che che contiene lo username immesso dall'utente.
- `principal: Principal`  
Contiene la caratteristica autenticativa del subject. Nel sistema MyTalk tale proprietà è riservata al campo mail.

- **subject:** `Subject`

Attributo che definisce il soggetto da autenticare.

## Metodi

+ **initialize(subject: `Subject`, handler: `CallbackHandler`, sharedState: `Map`, option: `Map`): `void`**

Il metodo ha il compito di inizializzare gli attributi privati della classe con i parametri d'input.

+ **login(): `boolean` throws `LoginException`**

Metodo richiamato per verificare un riscontro tra i dati passati per l'autenticazione e i dati presenti nel database. Il metodo deve appoggiarsi alla classe `UserDataDAO` per le operazioni di estrazione dati dal database.

+ **commit(): `boolean` throws `LoginException`**

Metodo richiamato direttamente dal framework se il login va a buon fine. Il suo scopo è quello di inizializzare il subject con i relativi principle.

+ **abort(): `boolean` throws `LoginException`**

Metodo usato per bloccare la procedura di login.

+ **logout(): `boolean` throws `LoginException`**

Metodo per il logout del sistema. Il suo scopo è quello di eliminare il principle e il subject.

## 4.8.7 CredentialLoader

### Funzione

Permette di caricare le credenziali di autenticazione, fornite dall'utente, per preparare la fase di login.

### Relazioni d'uso

- `javax.security.auth.callback.Callback`: usato durante la procedura di inserimento dati.
- `javax.security.auth.callback.CallbackHandler`: interfaccia implementata dalla classe.
- `javax.security.auth.callback.NameCallback`: tipo di dato richiesto in ingresso per completare la parte di login.
- `javax.security.auth.callback.PasswordCallback`: tipo di dato richiesto in ingresso per completare la parte di login.
- `java.io.IOException`: eccezione che può essere lanciata dal metodo `handle` definito dall'interfaccia `javax.security.auth.callback.CallbackHandler`.
- `javax.security.auth.callback.UnsupportedCallbackException`: eccezione che può essere lanciata dal metodo `handle` definito dall'interfaccia `javax.security.auth.callback.CallbackHandler`.

## Attributi

- **credential: AuthenticationData**  
Attributo contenente i dati da autenticare per la login dell'utente.
- **security: ISecurityStrategy**  
Attributo che contiene un oggetto che definisce un algoritmo di criptazione per i dati. Necessario per criptare i dati di autenticazione.

## Metodi

- + **CredentialLoader(credential: AuthenticationData, security: ISecurityStrategy)**  
Costruttore pubblico. Crea un'istanza con le credenziali fornite dall'utente (fornite in fase di login).
- + **handle(callbacks: Callback[]): void**  
Effettua il caricamento e crittografia delle credenziali fornite dall'utente per la fase di login

Il metodo può lanciare eccezioni:

- **IOException**
- **UnsupportedCallbackException**

## 4.9 Package org.softwaresynthesis.mytalk.server.authentication.servlet

### 4.9.1 LogoutServlet

#### Funzione

*Servlet* da richiamare per effettuare il logout dal sistema.

#### Relazioni d'uso

- **java.io.IOException**: eccezione richiamabile dai metodi **doPost()** e **doGet()**.
- **java.io.PrintWriter**: classe istanziata all'interno del metodo **doPost()**. Usata per scrivere l'output della *servlet*.
- **javax.servlet.ServletException**: eccezione rilancibile dai metodi **doPost()** e **doGet()**.
- **javax.servlet.http.HttpServlet**: classe estesa da **LoginManager**.
- **javax.servlet.http.HttpServletRequest**: classe per usata per la comunicazione con la *servlet* (inoltra i dati in ingresso alla *servlet*).
- **javax.security.auth.login.LoginContext**: classe usata in **doPost()** per eseguire la logout dal sistema.
- **javax.servlet.http.HttpServletResponse**: classe per usata per la comunicazione con la *servlet* (inoltra i dati in uscita dalla chiamata a *servlet*).
- **javax.servlet.http.HttpSession**: classe usata per definire una sessione HTTP.
- **abook.IUserData**: usata per definire un utente.
- **dao.UserDataDAO**: usata per comunicare tramite Hibernate con la tabella **UserData** della base di dati.

## Attributi

- `frozen serialVersionUID: long`

Utilizzato come ID univoco per identificare un oggetto serializzabile.

## Metodi

+ `LogoutServlet()`

Costruttore della classe. Richiama il costruttore della classe padre `HttpServlet`.

# `doGet(request: HttpServletRequest, response: HttpServletResponse): void`

Metodo di delega che richiama `doPost()` passando gli stessi dati in input.

# `doPost(request: HttpServletRequest, response: HttpServletResponse): void`

Metodo che costituisce il kernel logico di risposta della *servlet*. Il metodo inizia caricando i parametri ricevuti mediante `HttpServletRequest`. Una chiamata a tale *servlet* corrisponde ad una richiesta di logout e si attua impostando a false la sessione contenuta nell'oggetto `HttpServletRequest request`. Il metodo procede con la creazione di un'istanza di `LoginContext` denominata `context`. Quindi viene effettuata la memorizzazione dell'oggetto ritornato da una chiamata:

```
session.getAttribute("LoginContext");
```

L'oggetto così ottenuto dovrà essere controllato, ovvero si dovrà accertare che il tipo dinamico è conforme al tipo `LoginContext` (il programmatore dovrà obbligatoriamente usare la primitiva `instanceof`). Quindi si sovrascrive il contenuto di `context` con quanto ottenuto dalla chiamata a metodo sopracitata, e si richiama il metodo `logout()` a partire dall'oggetto `context`. Il metodo termina invalidando la sessione richiamando il metodo `invalidate()` di `HttpSession`.

### 4.9.2 LoginServlet

#### Funzione

*Servlet* da richiamare per effettuare l'autenticazione dell'utente.

#### Relazioni d'uso

- `java.io.IOException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `java.io.PrintWriter`: classe istanziata all'interno del metodo `doPost()`. Usata per scrivere l'output della *servlet*.
- `javax.security.auth.login.LoginContext`: classe usata in `doPost()` per avviare la login dell'utente.
- `javax.servlet.ServletException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `javax.servlet.http.HttpServlet`: classe estesa da `LoginManager`.
- `javax.servlet.http.HttpServletRequest`: classe usata per la comunicazione con la *servlet* (inoltre i dati in ingresso alla *servlet*).
- `javax.servlet.http.HttpServletResponse`: classe usata per la comunicazione con la *servlet* (inoltre i dati in uscita dalla chiamata a *servlet*).
- `javax.servlet.http.HttpSession`: classe usata per definire una sessione HTTP.
- `abook.IUserData`: usata per definire un utente.

- `dao.UserDataDAO`: usata per comunicare tramite Hibernate con la tabella `UserData` della base di dati.

### Attributi

- `frozen serialVersionUID: long`

Utilizzato come ID univoco per identificare un oggetto serializzabile.

### Metodi

+ `LoginServlet()`

Costruttore della classe. Richiama il costruttore della classe padre `HttpServlet`.

# `doGet(request: HttpServletRequest, response: HttpServletResponse): void`

Metodo che richiama `doPost()` passando gli stessi dati in input.

# `doPost(request: HttpServletRequest, response: HttpServletResponse): void`

Metodo che costituisce il kernel logico di risposta della *servlet*. Il metodo inizia caricando i parametri ricevuti mediante `HttpServletRequest`. Una chiamata a tale metodo corrisponde ad una richiesta di login e si attua impostando due campi `String` con i dati ottenuti da una chiamata a `getParameter("username")` e `getParameter("password")`. Quindi si crea un'istanza di `HttpSession` tramite il metodo `getSession(true)` del parametro `request`. Il metodo prosegue controllando se l'username e la password precedentemente ottenute hanno valore diverso da `null`. Nel caso il flusso principale continua creando un'istanza di `AuthenticationData` (denominata *credential*) a partire dai parametri `username` e `password`. Quindi si carica la path del file di configurazione in un'apposita stringa `pathFileConfig`, tramite la chiamata a metodo:

```
System.getenv("MyTalkConfiguration") + "\\LoginConfiguration.conf".
```

Il flusso principale prosegue all'interno di un blocco `try-catch` creando:

- un `CredentialLoader` (`loader`);
- un `LoginContext` (`context`);
- un `dao.UserDataDao` (`user`);

Quindi tramite `context` si esegue il login e si impostano gli attributi di sessione come segue:

```
session.setAttribute(LoginContext, context);
```

Infine si carica in `user` un'istanza di `dao.UserDataDAO` ottenuta dalla chiamata a metodo `UserDataDAO.getByEmail()`, e restituendo `user` in un formato di formattazione `Json` (`user.toJson()`).

#### 4.9.3 RegisterServlet

##### Funzione

*Servlet* da richiamare per effettuare la registrazione al sistema MyTalk.

## Relazioni d'uso

- `java.io.IOException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `java.io.PrintWriter`: classe istanziata all'interno del metodo `doPost()`. Usata per scrivere l'output della *servlet*.
- `javax.servlet.ServletException`: eccezione richiamabile dai metodi `doPost()` e `doGet()`.
- `javax.servlet.http.HttpServlet`: classe estesa da `LoginManager`.
- `javax.servlet.http.HttpServletRequest`: classe usata per la comunicazione con la *servlet* (inoltra i dati in ingresso alla *servlet*).
- `javax.servlet.http.HttpServletResponse`: classe usata per la comunicazione con la *servlet* (inoltra i dati in uscita dalla chiamata a *servlet*).
- `javax.servlet.http.HttpSession`: classe usata per definire una sessione HTTP.
- `abook.IUserData`: usata per definire un utente.
- `dao.UserDataDAO`: usata per comunicare tramite Hibernate con la tabella `UserData` della base di dati.
- `authentication.ISecurityStrategy`: usata per criptare/decriptare i dati da inviare/ricevere
- `authentication.AESAlgorithm`: implementazione di `authentication.ISecurityStrategy` usata dalla classe descritta.

## Attributi

- `frozen serialVersionUID: long`

Utilizzato come ID univoco per identificare un oggetto serializzabile. Tale ID deve essere impostato a "10002L".

## Metodi

+ `RegisterServlet()`

Costruttore della classe. Richiama il costruttore della classe padre `HttpServlet`.

# `doGet(request: HttpServletRequest, response: HttpServletResponse): void`

Metodo che richiama `doPost()` passando gli stessi dati in input.

# `doPost(request: HttpServletRequest, response: HttpServletResponse): void`

Metodo che costituisce il kernel logico di risposta della *servlet*. Il metodo inizia caricando i parametri ricevuti mediante `HttpServletRequest`. Una chiamata a tale metodo corrisponde ad una richiesta di registrazione al sistema e si attua impostando i dati di registrazione:

- name;
- username;
- mail;
- password;
- path immagine;
- domanda segreta;

- risposta alla domanda segreta;

Per eseguire tali operazioni, il metodo predispone delle opportune variabili di tipo `String` e crea uno `abook.IUserData` `user`. Quindi all'interno di un blocco `try catch` viene impostata una strategia di criptaggio dei dati (creando un'istanza di `authentication.AESAlgorithm`). Quindi vengono caricati i dati di registrazione nelle variabili precedentemente create. Per eseguire tali operazioni è necessario usare l'istruzione:

```
request.getParameter("NOME_PARAMTERO")
```

dove i nomi dei parametri sono:

- `username` (usata per l'indirizzo mail);
- `name`;
- `surname`;
- `password`;
- `answer`;
- `question`;
- `picturePath`;

Il passo successivo (sempre dentro il blocco `try`) consiste nel creare un'istanza di `dao.UserDataDAO` `userDAO`, e nell'eseguire un'operazione di `userDAO.insert()` passando come parametro `user` (l'utente precedentemente creato). Il blocco `try` termina impostando `result` a `"true"`.

Per quanto riguarda il blocco `catch`, in esso viene impostato `result` a `"false"`.

Il metodo termina scrivendo in un `PrintWriter` il valore di `result`.

## 4.10 Package `org.softwaresynthesis.mytalk.server.connection`

### 4.10.1 `PushInbound`

#### Funzione

Classe per la definizione di un apposito canale di comunicazione client-server. La classe è un'estensione di `org.apache.catalina.websocket.MessageInbound` che verrà usata poi in `ChannelServlet`. Si osservi che l'associazione tra un'istanza di `PushInbound` e un utente del sistema è univoca: fintanto che connesso un utente ha un proprio `PushInbound` presente sul server.

#### Relazioni d'uso

- `java.io.IOException`: Eccezione che può lanciare il metodo `OnTextMessage()`.
- `java.nio.CharBuffer`: tipologia di *buffer* usata per memorizzare il messaggio ricevuto come parametro d'ingresso.
- `java.util.Iterator`: usata per scorrere il contenuto dell'insieme di `AddressBookEntry`
- `java.util.Set`: struttura dati usata per memorizzare le *entry* (`AddressBookEntry`) che costituiscono la rubrica dell'utente.
- `org.apache.catalina.websocket.MessageInbound`: classe da estendere, aggiunge alla classe `PushInbound` le funzionalità necessarie per renderla un "canale di comunicazione" utilizzabile dai client.



- `com.google.gson.*`: converte i dati interni in una stringa formato json.
- `State`: classe interna utilizzata per rappresentare lo stato dell'utente a cui è associata l'istanza di `PushInbound`. Un utente ha uno stato fintanto che è connesso (quindi fintanto che esiste sul server un'istanza di `PushInbound` ad esso associata). Gli stati possibili sono: `available` (utente connesso e libero, quindi disposto a ricevere chiamate) e `occupied` (utente connesso ma attualmente occupato in un'altra conversazione).
- `abook.AddressBookEntry`: usata per definire la rubrica dell'utente, richiamata dal metodo `onTextMessage` nel momento in cui si presenta la necessità di aggiornare lo stato dell'utente e renderlo visibile ai suoi contatti.
- `abook.IUserData`: usata per riferire un istanza della classe ad un particolare utente.
- `dao.UserDataDAO`: usata per riferire un istanza della classe ad un particolare utente.

### Attributi

- `id: Long`  
Identificativo di tipo *Long* del canale di comunicazione associato ad un client univoco.
- `state: State`  
Attributo usato per memorizzare lo stato dell'utente "proprietario" del `PushInbound`.

### Metodi

- + `setId(n: Long): void`  
Metodo per impostare il valore contenuto nell'attributo `id`.
- + `getId(): Long`  
Metodo che ritorna il contenuto dell'attributo `id`.
- + `onTextMessage(message: CharBuffer): void`  
Metodo invocato al momento della ricezione di un messaggio da parte del client. Il metodo riceve in input un oggetto di tipo `CharBuffer` contenente il messaggio inviato dal client. Inizialmente il metodo crea un istanza per ognuno dei seguenti oggetti:
  - `Gson`: tipo di *Json* definito da *Google*;
  - `JsonParser`: *parser* usato per scorrere il contenuto di una stringa *Json*;
  - `JsonArray`: array popolato a partire dal *parsing* della stringa di messaggio data in input.

Prima di procedere si voglia considerare quanto segue: il metodo dopo aver "segmentato" il messaggio ricevuto in input, si occupa di esaminarne il contenuto che può essere di 5 tipologie, ciascuna identificata tramite un valore intero positivo da 1 a 5. Tale valore deve essere salvato nell'istanza di `JsonArray`. Detto ciò, tornando a definire il flusso principale del metodo, si osservi che:

- Se la richiesta inoltrata è del tipo 1: il metodo prende in lettura il messaggio e imposta il contenuto di `id` con il valore letto mediante procedura di *parsing* analoga a quella definita al passo precedente. Per farlo utilizza il metodo `fromJson` richiamato dall'istanza `gsonObj` di tipo `Gson` creata al passo precedente. A tale metodo passa il contenuto dell'array e in particolare ciò che è salvato nella posizione 1 (utilizzo di metodo `get(int i)`).

- Se la richiesta inoltrata è del tipo 2: il metodo procede con le istruzioni necessarie a scambiare i dati per la chiamata. Nello specifico viene salvato in un attributo di tipo `Long`, l'id del client che desidero chiamare, quindi ricerco l'oggetto `PushInbound` associato al client che desidero contattare, e gli inoltro il messaggio ricevuto come parametro d'ingresso.
- Se la richiesta inoltrata è del tipo 3: il metodo comunica al client "destinatario" della chiamata, l'id del client chiamante. La procedura è analoga a quella identificata nel punto precedente, con la specifica che il messaggio inoltrato è l'identificativo del cliente che desidera avviare la chiamata.
- Se la richiesta inoltrata è del tipo 4: il metodo si occupa dell'eliminazione della *webSocket*, contattando l'istanza `ChannelServlet` associata a questo `PushInbound`.
- Se la richiesta inoltrata è del tipo 5: il metodo viene usato per modificare lo stato dell'utente, con il valore passato tramite messaggio. Dopo la ricezione del messaggio, il metodo cambia il valore del campo `state` con il valore ricevuto come parametro d'ingresso. Quindi procede ricavando la lista degli utenti nella rubrica dell'utente che ha cambiato stato e comunica loro che è avvenuto un cambiamento di stato.

#### 4.10.2 ChanelServlet

##### Funzione

*Servlet* per la gestione delle connessioni. Richiamata dai client per ottenere le informazioni necessarie a stabilire una comunicazione client-client.

##### Relazioni d'uso

- `java.util.Vector`: vettore usato per memorizzare i `PushInBound` creati all'interno della classe.
- `javax.servlet.Servlet`: interfaccia d'implementazione. Necessaria per definire la classe come *servlet*.
- `javax.servlet.ServletConfig`: classe per la configura della *servlet*.
- `javax.servlet.ServletException`: eccezione richiesta per la firma del metodo `init` (metodo richiesto per implementazione dell'interfaccia `javax.servlet.Servlet`).
- `javax.servlet.annotation.WebServlet`: necessaria per definire la *servlet* come `WebServlet`.
- `javax.servlet.http.HttpServletRequest`: necessaria per definire le richieste inoltrate alla classe (richieste a *servlet*).
- `org.apache.catalina.websocket.StreamInbound`: classe utilizzata per definire un canale di comunicazione server-client.
- `org.apache.catalina.websocket.WebSocketServlet`: classe da estendere per utilizzare i metodi di comunicazione tramite *webSocket*.

##### Attributi

- `{frozen} serialVersionUID: long`

Utilizzato come ID univoco per identificare un oggetto serializzabile.

- `{frozen} clients: Vector<PushInbound>`

Vettore contenente i canali di comunicazione server-client. Un canale di comunicazione `PushInbound` viene creato dal Server al momento dell'autenticazione di un utente (al quale poi viene associato).

## Metodi

### + `ChannelServlet()`

Costruttore della *servlet*. Richiama `super()`.

### # `createWebSocketInbound(subProtocol: String, request: HttpServletRequest): StreamInbound`

Metodo per la creazione di una *websocket*. Il metodo ha anche il compito di salvare nel vettore di connessioni attive e la ritorna al client. Nello specifico il server segue l'iter:

- 1) creazione di un *PushInbound*;
- 2) aggiungo l'oggetto creato al punto precedente, al vettore di *PushInbound* `clients`;
- 3) il metodo termina ritornando l'istanza di *PushInbound* creata al punto 1.

### + `findClient(n: Long): PushInbound`

Metodo per la ricerca di una connessione client dato l'identificativo dell'utente, esegue una ricerca (mediante ciclo *for*) all'interno del vettore `clients` e ad ogni iterazione verifica se l'istanza *i*-esima presa in esame ha un campo `id` che corrisponde a quello dato in input come parametro di ricerca. Il metodo termina ritornando il *PushInbound* associato al client, se si verifica l'effettiva esistenza di un client con `id` uguale a quello passato in input, `null` altrimenti.

### + `removeClient(c: PushInbound): void`

Metodo per la rimozione di un oggetto *PushInbound* dal vettore `clients`. Il metodo esegue una ricerca tramite ciclo *for* (analoga a quella eseguita nel metodo `findClient`). Se la ricerca ha esito positivo il metodo termina eseguendo la rimozione dell'istanza dall'oggetto `clients` tramite la chiamata a metodo `remove`.

## 5 Specifica sotto-architettura clientpresenter

La sotto-architettura `clientpresenter`, a livello di definizione di prodotto, richiede una trattazione speciale a causa del particolare dominio tecnologico coinvolto. Essa è infatti definita con il linguaggio *JavaScript* che pur essendo definito come linguaggio orientato agli oggetti non permette di definire classi, a ciò si aggiunge un'ulteriore particolarità di *JavaScript*: è debolmente tipizzato. In ragione di ciò, al fine di facilitare al programmatore la comprensione del progetto (pesando quindi alle varie entità come a classi) senza però confonderlo in fase di stesura del codice, si è pensato di adoperare la seguente terminologia:

**ATTRIBUTI:** saranno definiti con una sintassi simile a quella già usata per la parte *server*, pertanto al nome dell'attributo sarà associato il tipo "logico" che idealmente rappresenta. In particolare emergono i seguenti casi:

- Nodi DOM: come si vedrà alcuni attributi rappresentano nodi DOM già esistenti nella pagina HTML che definisce l'interfaccia utente e modificati (a *run-time*) sulla base di informazioni ottenute dal *server*. Tali attributi dovranno essere codificati come:

```
this.mio_atributo = document.getElementById('id_DOM');
```

Tali attributi saranno indicati con il formalismo:

```
(+, -, #) nome_atributo: Nodo_DOM
```

- String: se l'attributo è destinato a contenere esclusivamente valori di tipo *String* sarà segnato nel documento come:

```
(+, -, #) nome_atributo: String
```

- Array: se l'attributo è un *array*, che in *JavaScript* non chiede di essere definito per tipo di valori contenibili, nel documento sarà precisato come:

```
(+, -, #) nome_atributo: Tipo[]
```

- Contatto: se l'attributo è un contatto, nel documento sarà precisato come:

```
(+, -, #) nome_atributo: Object(contact)
```

Tale oggetto conterrà i seguenti campi dati:

- name;
- surname;
- email;
- id;
- picturePath;
- state;
- blocked.

- Gruppo: se l'attributo è un gruppo, nel documento sarà precisato come:

```
(+, -, #) nome_atributo: Object(group)
```

Tale oggetto conterrà i seguenti campi dati:

- name;
- id;
- contacts (rappresentato come un *array* di `Object(contact)`).

**METODI:** la sintassi usata dal programmatore per definire un metodo in *JavaScript* dovrà essere la seguente:

```
var nome_metodo = function(){ contenuto }
```

mentre la sintassi usata nel documento per definirne la firma, resta la stessa usata in precedenza.

## 5.1 Package `org.softwaresynthesis.mytalk.clientpresenter.guicontrol`

### 5.1.1 `AddressBookPanelPresenter`

#### Funzione

Presenter incaricato di gestire il pannello della rubrica, contiene le funzioni associate ai *widget* grafici della vista relativi a quest'ultima e ha la responsabilità di aggiornare la vista sulla base dei dati ricevuti dal *server*.

#### Relazioni d'uso

Nessuna relazione d'uso evidenziata.

#### Attributi

- **element: DOM\_node**  
Attributo che definisce il contenuto del nodo DOM inerente alla rappresentazione della rubrica. Nello specifico tale nodo corrisponde ad un `div` con id `AddressBookPanel`.
- **urlServlet: String**  
Attributo che definisce l'URL dal quale è richiamabile la *servlet* usata per interagire con l'apparato server del sistema.
- **suffix: String**  
Suffisso da concatenare all'url per richiamare le *servlet* appropriate.
- **contacts: Array()**  
Attributo che definisce la lista degli utenti presenti nella rubrica dell'utente che ha effettuato l'autenticazione.
- **groups: Array()**  
Attributo che definisce la lista dei gruppi presenti nella rubrica dell'utente che ha effettuato l'autenticazione. Però ogni gruppo sono elencati i contatti contenuti in esso.
- **operations: String[]**  
*Array* di stringhe usato per memorizzare i nomi delle istruzioni eseguibili, essi rappresentano una parte del nome della *servlet* incaricata a risolvere l'operazione descritta (e.g. `GetContacts` rappresenta l'operazione svolta dalla *servlet* avente nome: `urlServlet` concatenato a "`GetContactsServlet`"). I valori con i quali si dovrà caricare **operations** sono:
  - `GetContacts`;
  - `DoAddContact`;
  - `DoDeleteContact`;
  - `DoInsertInGroup`;
  - `DoRemoveFromGroup`;

- DoCreateGroup;
- DoDeleteGroup;
- GetGroups;
- DoBlock;
- DoUnblock;
- DoSearch;

## Metodi

### + initialize(): void

Metodo per inizializzare `AddressBookPanel` e popolarlo con i contatti della rubrica, modifica il DOM creando tre div: `divSearch`, `divSort` e `divList`. Quindi sequenzialmente ne crea il contenuto:

- `divSearch`: definisce un campo per l'*input* testuale e un bottone per ricercare nella lista un utente avente tra le parole chiave, una con valore uguale a quello inserito nel campo d'*input*.
- `divSort`: crea delle *select* per specificare le tipologie di ordinamento attuabili sulla lista degli utenti.
- `divList`: *div* che viene popolato con i nomi degli utenti presenti nella rubrica dell'utente che l'ha richiamata.

Quindi dopo aver creato questi elementi il metodo procede inserendo nel DOM il "sotto-albero" così creato.

### - getAddressBookContacts(): void

Metodo che recupera i contatti e i gruppi della propria rubrica dal server usando la tecnologia AJAX. Il metodo crea una variabile contenente un'istanza `XMLHttpRequest()`. Quindi con metodo *POST* inoltra la richiesta alla *servlet* il cui indirizzo è generato dalla concatenazione di `urlServlet` con `operation[1]`. Elabora quindi i dati ottenuti mediante il metodo `JSON.parse()` e li salva negli array `contacts` e `groups`.

### + setup(): void

Inserisce i contatti estratti dal *server* all'interno della lista `AddressBookList` all'interno di `AddressBookPanel`.

### + displayContactList(contact: String[]): void

Metodo usato per visualizzare la lista dei contatti utente, esso prende il nodo DOM (già esistente) e destinato a contenere la lista dei contatti, popolandolo richiamando il metodo `addListItem()`.

### - addListItem(list: DOM\_node, contact: Object(contact): void

Metodo richiamato da `displayContactList()` per visualizzare i contatti presenti della rubrica, crea le variabili contenenti i dati del contatto da attribuire al *tag* di *markup* `li`. Tali variabili sono quelle tipiche di un oggetto `Object(contact)` (nome, cognome, email, status e immagine).

### - addOptionToSelect(select: DOM\_element, value: String, text: String): void

Metodo utilizzato per aggiungere un'opzione all'elemento `select` ricevuto come parametro dal metodo. Ha il compito di creare un `TextNode` ed appenderlo alla `select`.

**+ addContact(contact: Object(contact)): boolean**

Metodo per l'aggiunta di un contatto alla rubrica, riceve come parametro un oggetto di tipo contatto da inserire, appunto, nella propria rubrica. Per assicurarsi di non introdurre ridondanza ed aggiungere un contatto già presente il metodo esegue due controlli: uno lato *client* e uno lato *server*. Il controllo lato *client*, si scorre la lista **contact** alla ricerca di un **contact** che cerca un *match* con il nuovo contatto. Nel caso sia già presente il metodo termina ritornando *false*. Per eseguire il controllo lato *server* il metodo contatta il *server* mediante la *servlet* **AddressBookDoAddContactServlet**, questa verificherà se l'utente avente *id contact* è già presente nella rubrica, nel caso non lo sia si occuperà di creare la nuova istanza di **AddressBookEntry** e la registrerà nel database, in caso contrario non esegue operazioni. La chiamata a *servlet* ritorna un valore booleano che sarà *true* se l'inserimento ha avuto successo, *false* altrimenti. Se l'inserimento ha avuto successo il flusso principale prosegue eseguendo l'aggiornamento della rubrica in locale tramite la chiamata a metodo **setup()**. Il metodo termina infine ritornando *true* se è stato aggiunto l'utente, altrimenti lancia un'eccezione.

**+ removeContact(contact: Obj): boolean**

Metodo il cui funzionamento è analogo a quello precedentemente definito, si occupa di eliminare un contatto conoscendone l'*id* passato per parametro d'ingresso. Il flusso principale esegue due controlli per verificare la presenza dell'utente nella rubrica (analogamente a quanto visto nel caso precedente). Dopo tale verifica, se il contatto è realmente presente nella rubrica, la *servlet* **AddressBookDoRemoveContactServlet** avrà eseguito la cancellazione come conseguenza al *match* effettuato in fase di verifica, nel caso il metodo richiama **setup()** per rieseguire l'aggiornamento della rubrica. Il metodo ritorna infine *true* se effettivamente il contatto era presente ed è stato eliminato, altrimenti lancia un'eccezione.

**+ addGroup(name: String): boolean**

Metodo per l'aggiunta di un nuovo gruppo nella rubrica, permette l'aggiunta di gruppi con nomi duplicati (ossia nomi di gruppi già presenti nella rubrica). Il flusso principale richiama la *servlet* **AddressBookDoCreateGroupServlet** con il nome del gruppo da creare. Successivamente, se la *servlet* ha dato **setup()** esito positivo, viene richiamata il metodo *setup* che aggiorna la rubrica locale, altrimenti lancia un'eccezione.

**+ deleteGroup(idGroup: int): boolean**

Metodo per l'eliminazione di un nuovo gruppo presente nella rubrica, riceve come parametro d'ingresso l'*id* del gruppo da eliminare. Il flusso principale richiama la *servlet* **AddressBookDoRemoveGroupServlet** passandogli l'*id* del gruppo candidato all'eliminazione. Quindi, se la *servlet* notifica l'effettiva esistenza di un gruppo con tale *id*, essa si occupa dell'eliminazione e il flusso principale prosegue richiamando il metodo *setup* se la eliminazione ha avuto buon fine, altrimenti lancia un'eccezione.

**+ addContactInGroup(contact: Object(contact), group: Object(contact)): boolean**

Metodo per l'aggiunta di un contatto in un gruppo ben definito, riceve come parametro d'ingresso il contatto e il gruppo. Il metodo richiama da prima **contactExistInGroup**, successivamente il flusso principale richiama la *servlet* **AddressBookDoInsertInGroupServlet** passandogli i parametri d'ingresso, quindi se la *servlet* notifica l'effettiva esistenza di un gruppo con il nome ricevuto, essa si occupa di aggiungervi il contatto e il flusso principale ritorna al metodo **setup()** per aggiornare la rubrica in locale, ritorna infine un *feedback* booleano a *true* per dare notifica se l'operazione ha avuto buon fine, altrimenti rilancia un'eccezione.

**+ deleteContactFromGroup(contact: Object(contact), group: Object(group)): boolean**

Metodo per l'eliminazione di un contatto presente in un gruppo, riceve come parametro

d'ingresso il gruppo su cui effettuare la ricerca del contatto da eliminare e il contatto. Il metodo richiama da prima `contactExistInGroup`, poi il flusso principale richiama la *servlet* `AddressBookDoRemoveFromGroupServlet` passandogli i parametri d'ingresso, la *servlet* notifica l'effettiva esistenza di un gruppo con il nome ricevuto che contiene il contatto candidato all'eliminazione, essa si occupa dell'eliminazione e il flusso principale ritorna al metodo `setup()` per aggiornare la rubrica in locale, ritorna infine un *feedback* booleano per dare notifica sull'esito dell'operazione: *true* se è stato creato il nuovo gruppo, altrimenti lancia un'eccezione.

- `contactExistInGroup(contact: Object(contact), group: Object(group)): boolean`

Metodo che verifica l'esistenza di un contatto in un determinato gruppo, restituisce un *feedback* booleano per confermare o meno l'esistenza dell'utente ricercato nel gruppo designato.

+ `applyFilterByString(param: String): array()`

Metodo che data una stringa come parametro d'ingresso filtra la lista di contatti presente in locale. Non richiede interazione con il server, ovvero non vengono richiamate *servlet*.

In un primo step il programmatore dovrà creare un *array* atto a contenere l'elenco dei contatti da restituire, quindi il metodo procede con la definizione di un'espressione regolare basata sul parametro ricevuto. Per tale operazione la sintassi obbligatoria è:

```
var pattern = new RegExp(param)
```

Quindi viene avviato un ciclo *for* che analizza ogni contatto presente in `contacts`. Successivamente si effettua un filtraggio sui campi *name*, *surname*, *email* del contatto attualmente in esame. Per fare tali operazioni si dovrà usare:

```
pattern.test(\*elemento su cui ricercare param*)
```

Se tale filtraggio dà esito positivo, il contatto viene aggiunto all'*array* dei contatti da restituire. Il metodo termina quando, dopo aver eseguito il ciclo *for*, il metodo ritorna l'*array* sopracitato.

+ `applyFilterByGroup(idGroup: int): array()`

Metodo che dato un intero come parametro d'ingresso filtra la lista di contatti presente in locale, restituendo un elenco dei contatti appartenenti al gruppo avente id uguale a `idGroup`, inoltre non deve interagire con il *server*, ovvero non vengono richiamate *servlet*.

Come per `applyFilterByString()` anche in questo caso il metodo deve creare un *array* da popolare con i contatti appartenenti al gruppo `idGroup`. Quindi viene eseguito un ciclo *for* che ispeziona tutti i contatti presenti in `contacts` e per ogni ciclo il metodo controlla i gruppi in cui compare il contatto *i*-esimo (tale azione si attua per mezzo di un secondo ciclo innestato), se si verifica una corrispondenza tra l'id del gruppo in esame con l'identificativo `idGroup`, si aggiunge il contatto all'*array* creato in precedenza. Il metodo termina restituendo l'*array* sopra-citato.

+ `showFilter(filteredContacts: Array()): void`

Metodo usato per l'eliminare il contenuto preesistente nella rubrica e visualizza il nuovo contenuto filtrato. Il metodo riceve come parametro un *array* che rappresenta la lista filtrata dei contatti da visualizzare. Per eseguire tali operazioni viene avviato un ciclo *for* per ogni contatto presente in `contacts`. All'interno di tale ciclo deve essere eseguita la seguente istruzione:



```
addItem(/*nome della variabile contenente il nodo dom AddressBookList*/,
filtredContacts[contact])
```

+ `contactAlredyPresent(contact: Object(contact))`: `boolean`

Metodo che dato un contatto verifica se appartiene alla rubrica. Per compiere tale azione viene da prima ricaricata la rubrica, mediante una chiamata a metodo `getAddressBookContacts()`. Quindi si usa un ciclo *for* per scorrere l'intera lista precedentemente ottenuta e ad ogni ciclo si controlla se il contatto *i*-esimo ha id uguale a quello del contatto ricevuto come parametro d'ingresso (`contact.id`). Se tale confronto ha esito positivo, il metodo termina ritornando *true*, altrimenti il metodo termina al termine del ciclo *for* ritornando *false*.

+ `blockUser(contact: Object(contact))`: `boolean`

Metodo utilizzato per bloccare (all'interno della rubrica utente) il contatto `contact` passato come parametro d'ingresso. Il primo step consiste nell'eseguire il controllo sul parametro ricevuto. Per farlo si osserva il valore ritornato da una chiamata a metodo `contactAlredyPresent()` passando come parametro d'ingresso `contact`. Se il valore di ritorno è *false* il metodo termina con l'eccezione "Contatto non presente nella rubrica". Altrimenti il controllo passa allo step successivo, in cui si verifica se l'attributo `blocked` di `contact` è impostato a *true*. Nel caso il metodo termina con l'eccezione "Contatto già bloccato". Se il flusso principale ha superato i test precedenti, allora viene creata un'istanza di request `XMLHttpRequest()` usata per contattare la *servlet* avente nome:

```
urlServlet + operations[8] + suffix
```

Per tale richiesta deve essere associata, all'evento `onreadystatechange`, una funzione che salva in un array il contenuto di una chiamata:

```
JSON.parse(request.responseText)
```

Il flusso principale prosegue aprendo la request così definita (uso del metodo `open(POST, urlServlet + operations[8] + suffix, true)`) e inviando il contatto mediante una chiamata `request.send(contactId= + contact.id)`.

Il metodo termina eseguendo un controllo. Se il risultato `result` della chiamata a *servlet* ha valore *true* allora il metodo termina aggiornando la rubrica mediante una chiamata `setup()` e ritornando *true*. In caso contrario il metodo termina con un'eccezione.

+ `unlockUser(contact: Object(contact))`: `boolean`

Metodo usato per sbloccare un utente all'interno di una rubrica. Il metodo ha un funzionamento analogo al precedente, nello specifico vengono attuati gli stessi controlli prima tramite `contactAlredyPresent()` e poi tramite la verifica del contenuto del `contact.block`. Successivamente si crea una `XMLHttpRequest` che si appoggia alla *servlet* definita dalla stringa:

```
urlServlet + operations[9] + suffix
```

Tale server ha il compito di sbloccare il contatto `contact`. Quindi il metodo termina eseguendo un controllo sul tipo di risultato. Se l'operazione precedente è avvenuta con successo allora si procede aggiornando la rubrica (chiamata a metodo `setup()`), altrimenti il metodo termina con un'eccezione.

```
+ getGroupsWhereContactsIs(contact: Object(contact)): Object(group)
```

Tale metodo ha il compito, dato un utente, di ritornare una lista di gruppi in cui esso è presente. Riceve come unico parametro il contatto da ricercare e successivamente vengono ciclati tutti i gruppi e per ogni gruppo si controlla se è inserito il contatto, in caso di match viene aggiunto il gruppo alla lista di gruppi da ritornare.

### 5.1.2 LoginPanelPresenter

#### Funzione

Presenter incaricato di gestire il pannello di *login*, prende in carica i dati inseriti dall'utente (username e password) passandoli al server per eseguire l'autenticazione. La *servlet* usata a tale scopo è *LoginManager*.

Il presenter si occupa anche di fornire delle funzionalità per il recupero della password.

#### Relazioni d'uso

#### Attributi

```
- element: DOM_node
```

Attributo che definisce il contenuto del nodo DOM inerente alla schermata di *login*, nello specifico tale nodo corrisponde ad un `<div>` con id `LoginPanel`.

```
- urServlet: String
```

Attributo contenente l'url della *servlet* incaricata di gestire il *login*.

#### Metodi

```
- testCredentials(data: String): void
```

Metodo che testa quanto ricevuto dal server e, in caso di login avvenuto correttamente, reindirizza il browser nel presenter finale dopo aver salvato i dati dell'utente. Il metodo inizia salvando in una variabile avente nome *user* il contenuto di data trattato attraverso una chiamata `JSON.parse(data)`, quindi se il contenuto di *user* non è nullo si procede richiamando il `communicationcenter` e impostando il valore di `communicationcenter.my` a *user*. Il metodo termina richiamando il metodo `hide()` e avviando la costruzione della UI mediante una chiamata `mediator.buildUI()`.

```
+ sendAnswer(username: String, answer: String): void
```

Metodo per inviare la risposta alla domanda segreta al server, riceve due parametri d'ingresso: lo *username* dell'utente e la risposta (*answer*) da inviare al *server*. Come primo passo viene creata un'istanza di `XMLHttpRequest` salvata in una variabile avente nome *request*, successivamente si associa all'evento `onclick` di *request* una funzione avente il compito di inviare la richiesta *AJAX* al server, e di richiamare in risposta uno dei seguenti metodi:

- `correctAnswer()` nel caso la risposta inviata sia corretta;
- `incorrectAnswer()` se invece la risposta inviata dall'utente è errata.

```
+ correctAnswer(): void
```

Metodo usato per modificare il nodo DOM salvato in `element` per comunicare all'utente che la risposta da lui inserita (per il recupero password) è corretta. Il metodo deve rimuovere il figlio *passwordretrieval* da `element` quindi salva in una variabile avente nome *message* il contenuto di una chiamata `document.createElement("p")`. Successivamente procede caricando su *message* (usando il metodo `appendChild(var_figlio)`) il messaggio

da visualizzare per notificare all'utente il successo dell'operazione. Il messaggio visualizzato sarà:

```
Recupero password avvenuto correttamente.  
Ti è stata inviata un'email contenente i dati richiesti.
```

il messaggio sopracitato è temporaneo e dovrà essere rimosso allo scadere di un *timeout* di durata 2000 millisecondi (usare la funzione `window.setTimeout(nome_funzione)`).

+ `incorrectAnswer(): void`

Metodo che In caso di inserimento della risposta non corretta alla domanda segreta visualizza un messaggio di avvertimento all'utente per 2 secondi quindi lascia il controllo al *form* di inserimento della risposta alla domanda segreta.

Il metodo deve cercare un elemento (`document.createElement(p)`) per la visualizzazione del messaggio:

Dati non corretti. Inserire nuovamente la risposta.

Quindi si visualizza il messaggio con una procedura identica a quella usata in `correctAnswer()`.

+ `getSecretQuestion(): void`

Metodo che recupera la domanda segreta con una richiesta asincrona al server, esso imposta una variabile `userID` con il valore ritornato da una chiamata `getUsername()`, quindi avvia una richiesta *AJAX* iniziando un oggetto `XMLHttpRequest`. La variabile `request` contenente tale istanza dovrà associare all'evento `onreadystatechange` una funzione che si occupa di catturare la stringa ritornata dal server (uso obbligatorio di `question = this.responseText`). Il metodo termina infine restituendo il valore contenuto in `question`.

+ `buildRetrievePasswordForm(): nodo_DOM`

Metodo usato per costruire la *form* per il recupero della password utente, inizia costruendo la citata *form* con il valore ottenuto mediante una chiamata `document.createElement("fieldset")`, quindi a partire dalla variabile in cui ha memorizzato il dato ritornato con il metodo precedente, imposta l'attributo *id* con *passwordretrieval*. Definisce quindi una *label* con la domanda (utilizzare `document.createElement("label")` e imposta l'attributo *for* con *inputanswer*). Il metodo procede impostando il campo d'immissione della domanda e successivamente si impostano i seguenti attributi:

- attributo *id* con "inputanswer";
- attributo *name* con "inputanswer";
- attributo *placeholder* con "risposta";
- attributo *required* con "required";

Successivamente si crea il bottone di *submit* impostando gli attributi:

- attributo *type* con "submit";
- attributo *value* con "OK";

Associare all'evento *click* del bottone una funzione che richiami `sendAnswer()` passando come attributi `username` e il valore contenuto nell'elemento `inputAnswer`.

In ultima si usa `appendChild` per agganciare al nodo DOM la *label* e `inputAnswer`. Il metodo termina ritornando il nodo DOM.

**+ getUsername(): String**

Metodo per il recupero dello *username* dall'interfaccia grafica. Per tale operazione il metodo deve usare l'istruzione:

```
var username = document.getElementById("username").value;
```

quindi verifica se il contenuto di *username* è vuoto o no: nel caso lo sia, termina, altrimenti procede nel verificare se l'email inserita rispetta l'espressione regolare:

```
^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$
```

il metodo termina ritornando *username*.

**+ getPassword(data: array()): String**

Metodo che esegue il *login* inviando al server i dati di autenticazione, esso riceve come parametro d'ingresso un *array* associativo contenente *username* e *password*.

Il flusso principale inizia verificando il che il contenuto di `data.username` e di `data.password` non sia vuoto, quindi invia una richiesta *AJAX* al *server* (come sempre in questo caso è d'obbligo usare `var request = new XMLHttpRequest();`). Il programmatore dovrà associare all'evento `onreadystatechange` una funzione che si occupa di richiamare il metodo `testCredentials()`. Il passo successivo consiste "nell'aprire" la richiesta *request* con un'istruzione `open()` che riceve come parametro l'url della *servlet* associata (valore presente in `servletURL`), successivamente viene usata un'istruzione `setRequestHeader()` passando i seguenti parametri:

- "content-type";
- "application/x-www-form-urlencoded";

infine va creata la *querystring* da inviare (tramite istruzione `request.send()`), deve essere obbligatoriamente impostata a:

```
"username=" + encodeURIComponent(data.username) +  
"&password=" + encodeURIComponent(data.password) + "&operation=1"
```

Il metodo termina ritornando la *querystring*.

**+ hide(): void**

Metodo usato per nascondere il *form* di autenticazione, per lasciare spazio nella finestra ad altri elementi grafici come la schermata principale o il pannello di registrazione.

**+ initialize(): void**

Metodo usato per inizializzazione del pannello di login con la creazione di tutti i *widget* grafici che sono contenuti al suo interno. Nello specifico il metodo esegue la creazione (nell'ordine proposto) dei seguenti elementi:

- form, con attributi:
  - "name" impostato a "login";
  - "action" impostato a "°";
  - "method" impostato a "°";
  - "accept-charset" impostato a "utf-8";
- ul contennete il form;

- li che conterrà lo username, con attributi;
- label che riporterà la frase “Nome utente: ”. Label con attributi:
  - “for” impostato a “username”;
- input per l’inserimento della e-mail, con attributi:
  - “type” impostato a “email”;
  - “id” impostato a “username”;
  - “name” impostato a “username”;
  - “placeholder” impostato a “yourname@email.com”;
  - “required” impostato a “required”;
- input per l’inserimento della password, con attributi:
  - “type” impostato a “password”;
  - “id” impostato a “password”;
  - “name” impostato a “password”;
  - “placeholder” impostato a “password”;
  - “required” impostato a “required”;
- input submit pulsante di login, con attributi:
  - “type” impostato a “submit”;
  - “value” impostato a “Login”;
- input submit pulsante di registrazione, con attributi:
  - “type” impostato a “submit”;
  - “value” impostato a “Registrazione”;
- input submit pulsante di recupero password, con attributi:
  - “type” impostato a “submit”;
  - “value” impostato a “Recupera password”;

### 5.1.3 RegisterPanelPresenter

#### Funzione

Presenter incaricato di gestire la *form* di registrazione

#### Relazioni d’uso

Nessuna relazione d’uso evidenziata.

#### Attributi

- **servletURL: String**  
Url della *servlet* che deve gestire la registrazione.
- **element: DOM\_node**  
Elemento (inteso come nodo DOM) controllato da questo Presenter.

## Metodi

+ `getSurname(): String`

Metodo che estrae dalla *form* il valore del cognome del nuovo utente e lo restituisce.

+ `getName(): String`

Metodo che estrae dalla *form* il valore del nome del nuovo utente e lo restituisce.

+ `getAnswer(): String`

Metodo che estrae dal *form* la risposta alla domanda segreta associata al nuovo utente e la restituisce.

+ `getQuestion(): String`

Metodo che estrae dalla *form* la domanda segreta associata al nuovo utente e la restituisce.

+ `getPassword(): String`

Metodo che estrae dalla *form* la *password* associata al nuovo utente e la restituisce.

+ `getUsername(): String`

Metodo che estrae dalla *form* lo *username* del nuovo utente (per *username* si intende l'indirizzo email dell'utente). Si osservi che il metodo dovrà verificare che la mail restituita sia ben formattata secondo l'espressione regolare:

```
^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$
```

Se la mail è valida il metodo termina ritornandone il valore.

+ `getPicturePath(): String`

Metodo che estrae dalla *form* il percorso dell'immagine del nuovo profilo utente e lo restituisce.

+ `register(userData: Arrey()): String`

Metodo che Invia i dati ricevuti alla *servlet* per la creazione di un nuovo account utente. Tale metodo inizia creando un istanza (*request*) di `XMLHttpRequest` usata per comunicare con la *servlet*. Quindi il metodo abilita *request* ad inviare un richiesta sincrona al *server*, usando l'istruzione:

```
request.open("POST", servletURL, false)
```

Il metodo procede definendo la stringa da usare come *query* per da inviare alla *servlet*. Tale *querystring* contiene:

```
"username=" + encodeURIComponent(userData.username) +  
"&password=" + encodeURIComponent(userData.password) +  
"&question=" + encodeURIComponent(userData.question) +  
"&answer=" + encodeURIComponent(userData.answer)
```

Il metodo termina restituendo il contenuto di *querystring*.

+ `initialize(): void`

Metodo che si occupa di inizializzare la *form* di registrazione con la creazione di tutti i *widget* grafici che sono contenuti al suo interno. Nello specifico il metodo esegue i seguenti passi:

- creazione dell'elemento *form*;
- creazione dell'elemento `<ul>` contenuto nel *form*;
- creazione dell'item per lo username, definito mediante una *label* riportante la frase "Indirizzo email: ", e di un campo *input* di tipo email per l'inserimento della email dell'utente;
- costruisce il *list item* con la *label* e l'*input* definiti al punto precedente, usando il metodo `appendChild()`;
- crea l'item per la password, definendo una *label* riportante la frase "Password: " e un oggetto *input* di tipi *password*;
- costruisce il *list item* con la *label* e l'*input* definiti al punto precedente, usando il metodo `appendChild()`;
- creazione dell'item per la domanda segreta definito mediante una *label* avente valore "Domanda segreta: " e un oggetto *input* di tipo *text*;
- costruisce il *list item* con la *label* e l'*input* definiti al punto precedente, usando il metodo `appendChild()`;
- creazione dell'item per la risposta alla domanda segreta, definito mediante una *label* avente valore "Risposta: " e un oggetto input di tipo *text*;
- costruisce il *list item* con la *label* e l'*input* definiti al punto precedente, usando il metodo `appendChild()`;
- creazione dell'item per il nome, definito mediante una *label* avente valore "Nome: " e un oggetto input di tipo *text*;
- costruisce il *list item* con la *label* e l'*input* definiti al punto precedente, usando il metodo `appendChild()`;
- creazione dell'item per il cognome, definito mediante una *label* avente valore "Cognome: " e un oggetto input di tipo *text*;
- costruisce il *list item* con la *label* e l'*input* definiti al punto precedente, usando il metodo `appendChild()`;
- creazione dell'item per inserire l'immagine utente, definito mediante una *label* avente valore "Immagine del profilo: " e un oggetto input di tipo *text*;
- costruisce il *list item* con la *label* e l'*input* definiti al punto precedente, usando il metodo `appendChild()`;
- creazione di un pulsante di registrazione (oggetto input di tipo submit e value "Registrati").
- operazione finale per "appendere" tutti gli oggetti creati al `registerForm`.

+ `hide(): void`

Nasconde il *form* di registrazione per lasciare spazio alla schermata principale dell'applicativo (che deve essere costruita dal `PresenterMediator`).

#### 5.1.4 CommunicationPanelPresenter

##### Funzione

Questo presenter ha il compito di gestire tutte le comunicazioni che possono avvenire tra persone, quindi sia di natura testuale che di tipo audio - audio/video.

##### Relazioni d'uso

Nessuna relazione d'uso evidenziata.

## Attributi

- `chatElements: HTMLDivElement[]`  
array associativo contenente tutte le chat aperte in un dato momento.

## Metodi

- `createLabel(user: Object(contact)): String`  
Metodo che dato un utente (parametro del metodo), restituisce una stringa identificativa per il contatto in questione. Più precisamente, se il contatto contiene sia il nome che il cognome, la stringa restituita sarà composta da questi due elementi, altrimenti in base ai casi solo il nome o il cognome. Nell'ulteriore caso non siano presenti né il nome né il cognome, restituisce l'email, campo dati obbligatorio in fase di registrazione e quindi sempre presente.
- `createChatItem(user: Object(contact)): DOM_node`  
Tale metodo restituisce un `DOM_element` di tipo `<li>` contenente l'identificativo dell'utente con cui si sta comunicando (parametro del metodo). All'oggetto restituito viene assegnato un "id" corrispondente all'"id" dell'utente con cui si vuole dialogare. All'evento `onClick` viene collegato l'invocazione del metodo `displayChat(user)` che fa visualizzare la chat, viene inoltre creato il bottone per chiudere la chat che con l'evento "onclick" richiama il metodo `removeChat(user)`.
- `createChatElement(user: Object(contact)): DOM_node`  
Tale metodo restituisce un `DOM_element` contenente un `div` che rappresenta l'area di chat. A tale elemento viene assegnato un "id" impostato a "divContainerChat". Deve essere creato anche una `form` assegnando l'"id" dell'utente come valore all'attributo "id" a tale `form` va aggiunto una `testarea` con `id` impostato a `chatText` dove andrà visualizzata la cronologia della chat; una casella di input con `id` uguale `input`; un bottone per inviare il messaggio assegnandogli all'evento `onClick` l'invocazione del metodo `send(user, text)` del `CommunicationPanelPresenter`. Infine viene aggiunta la `form` all'elemento `div` e ritornato.
- + `displayChat(user: Object(contact)): void`  
Provoca la visualizzazione della chat con l'utente (ricevuto da parametro) nel `div` contenitore pertinente.
- + `addChat(user: Object(contact)): void`  
Provoca l'aggiunta di una chat, con l'utente ricevuto come parametro, andando ad aggiornare il campo privato `chatElements` e visualizzandola nell'elemento `ulOpenChat`.
- + `removeChat(user: Object(contact)): void`  
Comportamento simile al metodo scritto precedentemente. Viene chiusa una chat aperta con l'utente ricevuto da parametro andando ad eliminare l'elemento corrispondente da `chatElements` e rimuovendo anche il `DOM_element` dall'`ulOpenChat`.
- + `appendToChat(user: Object(contact), text: String): void`  
Il metodo ha il compito di aggiunge una stringa `text` all'interno dell'area di testo che è associata alla chat con l'utente `user` (parametro del metodo).
- + `createPanel(): DOM_node`  
Con l'invocazione di tale metodo, viene inizializzato il pannello costruendo i `widget` grafici interni e lo restituisce in modo che possa essere inserito all'interno del pannello principale. Più dettagliatamente, deve costruire:
  - `div` relativo alla chiamata assegnandogli come "id" `CommunicationPanel`;



- div relativo alla gestione della chiamata assegnandogli come “id” divCall;
- div relativo alla chat testuale assegnandogli come “id” divChat;
- elemento video per la visualizzazione del proprio stream ricavato dalla videocamera con “id” myVideo;
- elemento video per la visualizzazione dello *stream* dell’altro utente con *id* otherVideo;
- div dove inserire la visualizzazione delle statistiche della chiamata tra cui:
  - elemento “span” per byte Ricevuti con *id*= statReceived;
  - elemento “span” per byte Inviati con *id*= statSend;
  - elemento “span” per il tempo con *id*= timerSpan;
- bottone per la terminazione della chiamata che all’evento *onClick* richiama il metodo `endCall()` di `communicationcenter`;
- elemento div contenente le chat aperte.

Tutti gli elementi sopra descritti andranno poi aggiunti al parametro di ritorno *element* tramite la funzione `appendChild()` fornita dai `DOM_element`.

+ `updateTimer(text: String): void`

Il metodo aggiorna il tempo di comunicazione durante una chiamata. Imposta il campo *value* dello *span* relativo alla visualizzazione del tempo chiamata con il valore del parametro ricevuto in input (*text*).

+ `updateStats(text: String, isReceivedData: boolean): void`

Il metodo controlla tramite il parametro booleano ricevuto in input (*isReceivedData*) se bisogna aggiornare lo *span* relativo ai dati inviati oppure lo *span* relativo ai dati ricevuti e lo imposta con il valore *text* ricevuto per parametro.

+ `getMyVideo(): DOM_element`

Il metodo ritorna l’elemento DOM con *id* uguale ad “myVideo”.

+ `getOtherVideo(): DOM_element`

Il metodo ritorna l’elemento DOM con *id* uguale ad “otherVideo”.

### 5.1.5 ContactPanelPresenter

#### Funzione

Presenter incaricato di gestire il pannello che visualizza le informazioni di un singolo contatto.

#### Relazioni d’uso

Nessuna relazione d’uso evidenziata.

#### Attributi

- `element: DOM_element`

elemento controllato dal presenter in questione.

#### Metodi

- `adjustBlockButtonDisplay(contact: Object(contact)): void`

Il metodo riceve come parametro un contatto (*contact*), successivamente viene controllato se tale utente è bloccato o meno e imposta correttamente la proprietà *style* agli elementi della vista. più dettagliatamente:

se il contatto è bloccato:

- nascondo il bottone per bloccare un utente;
- mostro il bottone per sbloccare un utente

altrimenti

- mostro il bottone per bloccare un utente;
- nascondo il bottone per sbloccare un utente

- `buildGroupsDiv(contact Object(contact))`: `void`

Il metodo, che riceve come parametro un contatto (*contact*), ha il compito di recuperare tutti i gruppi a cui il contatto appartiene e visualizzare nel *div* opportuno una *label* per ogni gruppo trovato indicandone il nome.

+ `createPanel()`: `DOM_element`

Metodo richiamato quando viene selezionato un contatto dalla rubrica. Inizializza il pannello che mostra le informazioni del contatto selezionato. Più precisamente, verranno creati tanti `listItem` (`<li>`) quanti i seguenti elementi:

- nome;
- cognome;
- email;
- avatar raffigurante l'immagine del contatto;
- div contenente i gruppi a cui il contatto appartiene;
- pulsante per la chiamata audio;
- pulsante per la chiamata video;
- pulsante per avviare una chat;
- pulsante per bloccare il contatto;
- pulsante per sbloccare il contatto.

Infine ritorno l'elemento creato.

+ `display(contact: Object(contact))`: `void`

Metodo che popola nel corretto modo tutti gli elementi necessari per la visualizzazione di un utente. Prima di tutto quindi si recupereranno tutti gli elementi necessari, ossia:

- elemento per nome;
- elemento per cognome;
- elemento per email;
- elemento per avatar;
- pulsante per aggiungere un contatto alla rubrica;
- pulsante per la chiamata audio;
- pulsante per la video-chiamata;
- pulsante per avviare una chat;
- pulsante per bloccare il contatto;
- pulsante per sbloccare il contatto.

Successivamente si andranno a popolare tali elementi utilizzando le informazioni del contatto passato come parametro (*contact*), infine si imposterà correttamente il comportamento dei bottoni associando le giuste chiamate a metodi del *mediator*, ossia:

- aggiungi contatto -> `onContactAdded(contact.id);`
- blocca contatto -> `onBlockContact();`
- sblocca contatto -> `onUnlockContact();`
- chiamata audio -> `onCall(contact, false);`
- chiamata video -> `onCall(contact, true);`
- avvia chat -> `onChatStarted(contact);`

### 5.1.6 MainPanelPresenter

#### Funzione

Presenter incaricato di gestire il pannello principale, ossia quello centrale.

#### Relazioni d'uso

Nessuna relazione d'uso evidenziata.

#### Attributi

- **element:** `DOM_element`  
elemento controllato dal presenter in questione.

#### Metodi

- + **initialize():** `void`  
Metodo utilizzato per costruire il pannello principale che occupa la parte centrale della finestra, esso imposta inoltre l'immagine del logo "MyTalk" come sfondo quando il pannello è vuoto.
- + **displayChildPanel(node DOM\_node):** `void`  
Metodo che visualizza il `DOM_node` (*node*) ricevuto come parametro nel pannello in questione.
- + **hide():** `void`  
Metodo che rende invisibile il pannello impostando la proprietà *display* a "none".

### 5.1.7 PresenterMediator

#### Funzione

Presenter incaricato di gestire la collaborazione tra i vari sotto-presenter.

#### Relazioni d'uso

Nessuna relazione d'uso evidenziata.

#### Attributi

- **presenters:** `Array associativo`  
*array* contenente tutti i sotto-presenter.  
Presenter di primo livello:
  - login -> `LoginPanelPresenter;`
  - register -> `RegisterPanelPresenter;`

- addressbook -> AddressBookPanelPresenter;
- tools -> ToolsPanelPresenter;
- main -> MainPanelPresenter.

Presenter di secondo livello:

- accountsettingspp -> AccountSettingsPanelPresenter;
- communicationpp -> CommunicationPanelPresenter;
- contactpp -> ContactPanelPresenter;
- callhistorypp -> CallHistoryPanelPresenter;
- messagepp -> MessagePanelPresenter;;
- searchresultpp -> SearchResultPanelPresenter;
- grouppp -> GroupPanelPresenter.

## Metodi

- `getCommunicationPP(): CommunicationPanelPresenter` Il metodo restituisce l'istanza dell'attributo di tipo `CommunicationPanelPresenter`.

+ `buildUI(): void`

Tale metodo ha il compito di inizializzare l'interfaccia grafica delegando ai presenter il compito di disegnare gli elementi principali dell'interfaccia, incaricando i presenter di primo livello di creare e popolare i rispettivi pannelli. In particolare, si nasconderanno i seguenti pannelli:

- register;
- login;
- addressbook;
- main;
- tools.

+ `buildLoginUI(): void`

Visualizza l'interfaccia di autenticazione al sistema, che comprende il *form* di *login*. Viene chiamato quindi il metodo `initialize()` del `LoginPanelPresenter` per la sua costruzione.

+ `buildRegistrationUI(): void`

Visualizza il form di registrazione al sistema, utilizzato dagli utenti che vogliono creare un nuovo account, viene chiamato quindi il metodo `initialize()` del `RegisterPanelPresenter` per la sua costruzione.

+ `onContactSelected(contact: Object(contact)): void`

Metodo invocato nel momento in cui viene selezionato un contatto, tale evento invoca il metodo `display` del `ContactPanelPresenter` prendendo come parametro l'oggetto *contact* ricevuto inizialmente.

+ `onContactAdded(contact: Object(contact)): void`

Funzione di *callback* richiamata dai pulsanti di `SearchResultPanel` che comunica all'`AddressBookPanelPresenter` di aggiungere un contatto. Riceve come parametro l'utente che si vuole aggiungere (*contact*), questo parametro viene a suo volta passato alla funzione `addContact()` di `addressBookPanelPresenter`. In caso di errore, viene segnalato con un *alert*.

+ `onContactRemoved(userID: String): void`

Funzione di *callback* richiamata dai pulsanti di SearchResultPanel che comunica all'AddressBookPanelPresenter di rimuovere un contatto. Il metodo riceve come parametro l'utente che si vuole rimuovere (*contact*). Questo parametro viene a suo volta passato alla funzione `removeContact()` di addressBookPanelPresenter. In caso di errore, viene segnalato con un *alert*.

+ `onGroupAdded(name: String): void`

Funzione di *callback* richiamata dai pulsanti di SearchResultPanel che comunica all'AddressBookPanelPresenter di aggiungere un gruppo. Il metodo riceve come parametro il nome del nuovo gruppo che si vuole aggiungere (*name*). Questo parametro viene a suo volta passato alla funzione `addGroup()` di addressBookPanelPresenter.

+ `onGroupRemoved(group: object(group)): void`

Funzione di *callback* richiamata dai pulsanti di SearchResultPanel che comunica all'AddressBookPanelPresenter di rimuovere un gruppo. Il metodo riceve come parametro il gruppo che si vuole rimuovere (*group*). Questo parametro viene a suo volta passato alla funzione `removeGroup()` di addressBookPanelPresenter.

+ `onContactAddedInGroup(contact: Object(contact), group: Object(group)): void`

Funzione di *callback* richiamata dai pulsanti di SearchResultPanel che comunica all'AddressBookPanelPresenter di aggiungere un contatto ad un gruppo. Il metodo riceve come parametro il contatto che si vuole aggiungere (*contact*) e il gruppo a cui aggiungere il contatto (*group*). Questi parametri vengono a suo volta passati alla funzione `addContactInGroup()` di addressBookPanelPresenter.

+ `onContactRemovedInGroup(contact: Object(contact), group: Object(group)): void`

Funzione di *callback* richiamata dai pulsanti di SearchResultPanel che comunica all'AddressBookPanelPresenter di rimuovere un contatto da un gruppo. Il metodo riceve come parametro il contatto che si vuole rimuovere (*contact*) e il gruppo a cui rimuovere il contatto (*group*). Questi parametri vengono a suo volta passati alla funzione `removeContactFromGroup()` di addressBookPanelPresenter.

+ `onBlockedContact(contact: Object(contact)): void`

Funzione di *callback* che comunica all'AddressBookPanelPresenter di bloccare un contatto. Il metodo riceve come parametro il contatto che si vuole bloccare (*contact*), questo parametro viene a suo volta passato alla funzione `blockUser()` di addressBookPanelPresenter. In caso di errore, viene notificato con un *alert* riportando l'errore relativo.

+ `onUnlockContact(contact: Object(contact)): void`

Funzione di *callback* che comunica all'AddressBookPanelPresenter di sbloccare un contatto. Il metodo riceve come parametro il contatto che si vuole sbloccare (*contact*), questo parametro viene a suo volta passato alla funzione `unlockUser()` di addressBookPanelPresenter. In caso di errore, viene notificato con un *alert* riportando l'errore relativo.

+ `displayMessagePanel(): void`

Provoca la creazione del pannello della segreteria e la sua visualizzazione all'interno del MainPanel come elemento figlio. La costruzione del pannello è affidata al metodo *createPanel* che viene reso disponibile da tutti i presenter di secondo livello, viene quindi creato dal metodo `createPanel()` del presenter messagePanelPresenter un elemento di tipo DOM\_node che viene poi passato al presenter *main* come parametro del metodo `displayChildPanel`.

- + `displayAccountSettingsPanel(): void`  
Provoca la creazione del pannello delle impostazioni dell'utente e la sua visualizzazione all'interno del MainPanel. Come il metodo sopra citato, viene creato dal metodo `createPanel()` del presenter CallHistoryPanelPresenter un elemento di tipo `DOM_node` che viene poi passato al presenter *main* come parametro del metodo `displayChildPanel`.
- + `displayCallHistoryPanel(): void`  
Provoca la creazione del pannello dello storico delle chiamate e la sua visualizzazione all'interno del MainPanel. Come il metodo sopra citato, viene creato dal metodo `createPanel()` del presenter CallHistoryPanelPresenter un elemento di tipo `DOM_node` che viene poi passato al presenter *main* come parametro del metodo `displayChildPanel`.
- + `onFiltredAppliedByParam(param: String): void`  
Funzione di *callback* richiamata dai pulsanti di SearchResultPanel che comunica all'AddressBookPanelPresenter di filtrare la lista dei contatti secondo il parametro ricevuto (*param*). Viene scatenata l'invocazione del metodo `applyFilterByString` del presenter AddressBookPresenter passandogli come parametro *param*.
- + `getGroupsWhereContactsIs(contact: Object(contact)): void`  
Funzione di *callback* che comunica all'AddressBookPanelPresenter di cercare i gruppi a cui appartiene un utente nella propria rubrica ricevuto come parametro (*contact*). Viene quindi invocato il metodo `getGroupsWhereContactIs()` del presenter AddressBookPanelPresenter.
- + `displaySearchResultPanel(): void`  
Provoca la creazione del pannello delle impostazioni del proprio account e la sua visualizzazione all'interno del MainPanel. Viene creato dal metodo `createPanel()` del presenter AccountSettingsPanelPresenter un elemento di tipo `DOM_node` che viene poi passato al presenter *main* come parametro del metodo `displayChildPanel`.
- + `displayCommunicationPanel(): void`  
Provoca la creazione del pannello delle comunicazioni e la sua visualizzazione all'interno del MainPanel. Viene creato dal metodo `createPanel()` del presenter CommunicationPanelPresenter un elemento di tipo `DOM_node` che viene poi passato al presenter *main* come parametro del metodo `displayChildPanel`.
- + `displayContact(contact: Object(contact)): void`  
L'invocazione di tale metodo provoca la visualizzazione della scheda di un contatto (*contact*) nel CommunicationpanelPresenter. Viene richiamato il metodo `displayCommunicationPanel()` e successivamente il metodo `display` passando come contatto il parametro ricevuto precedentemente.
- + `contactAlreadyPresent(contact: Object(contact)): boolean`  
Metodo che controlla se l'utente ricevuto come parametro del metodo (*contact*) è già presente nella rubrica. Viene invocato il metodo `contactAlreadyPresent()` di AddressBookPanelPresenter passando come parametro il contatto in questione.
- + `onChatStarted(user: Object(contact)): void`  
Coordina i presenter nel momento in cui ha inizio una nuova comunicazione testuale e incapsula la collaborazione fra ContactPanelPresenter e CommunicationPanelPresenter. Viene quindi
  - invocato il metodo `createPanel()` di CommunicationPanelPresenter;
  - invocato il metodo `displayChildPanel()` di MainPanelPresenter passandogli come parametro l'oggetto creato precedentemente;

- invocato il metodo `addChat()` di `CommunicationPanelPresenter` passando come parametro l'utente interessato;
- invocato il metodo `displayChat()` di `CommunicationPanelPresenter` passando come parametro l'utente interessato.

+ `onCall(contact: Object(contact), onlyAudio: boolean): boolean`

Metodo per gestire la chiamata, riceve come parametri l'utente che si vuole chiamare (*contact*) e un *flag* per segnalare se la chiamata è solo audio (*onlyAudio*).

#### 5.1.8 MessagePanelPresenter

##### Funzione

Presenter incaricato di gestire i messaggi in segreteria.

##### Relazioni d'uso

Nessuna relazione d'uso evidenziata.

##### Attributi

- `messages: Message[]`

Array che conterrà tutti i messaggi della segreteria

- `servlets: String[]`

Array che contiene le tre URL delle *servlets* da contattare per i messaggi

##### Metodi

- `getServletURLs(): void`

Configura gli URL delle *servlet* da interrogare leggendoli dal file di configurazione in base alle operazioni che questo presenter deve essere in grado di compiere.

Più precisamente, si dovrà creare una "XMLHttpRequest" rivolta al file XML di configurazione dove sono riportate tutte le *servlet*, ricevuta la risposta si popolerà l'*array* dichiarato nella classe (*servlet*).

- `deleteMessage(idMessage: String): void`

Elimina un messaggio dalla segreteria contattando la *servlet* responsabile dell'operazione e scaricando nuovamente i messaggi. Il metodo procede quindi nell'invocazione della *servlet* corretta passando come parametro l'"id" ricevuto come parametro del metodo (*idMessage*). Se la *servlet* ritorna il valore *true*, significa che l'eliminazione è andata a buon fine e quindi si ricaricano i messaggi. In caso di errore, apparirà all'utente un messaggio d'errore indicante il motivo.

- `addListItem(message: Object()): void`

Aggiunge un messaggio ad una lista per creare l'elenco della segreteria telefonica generale. Viene quindi creato un *list item* e successivamente aggiunto alla lista `MessageList` di tale Presenter.

- `setAsRead(idMessage: String, valueToSet: String): void`

Imposta lo stato di un messaggio come "letto" oppure "non letto" a seconda del parametro ricevuto in input. Tale metodo demanda il compito alla *servlet* opportuna passandogli entrambi i parametri (*idMessage* e *valueToSet*). In caso di messaggio d'errore da parte del server, viene notificato anche all'utente.

**- `getMessages(): void`**

Ottiene i messaggi di segreteria salvati nel server contattando la *servlet* corrispondente. Una volta ricevuta risposta con relativi dati associati, li salva all'interno dell'*array messages* dichiarato all'interno di questo presenter. Per contattare la *servlet*, va usato come in precedenza "XMLHttpRequest".

**+ `createPanel(): DOM_element`**

Costruisce il pannello della segreteria telefonica, che deve essere visualizzato all'interno del MainPanel dell'applicazione quando è selezionata la funzione corrispondente dal pannello degli strumenti. Il MessagePanel è costituito da un elemento video seguito da un *div* che a sua volta contiene una lista di messaggi creata con l'ausilio dei metodi `getMessages()` e `addListItem()`. Tutti questi elementi vanno aggiunti all'elemento *element* ritornato poi dal metodo.

**+ `setup(): void`**

Metodo che ogni sua invocazione provoca il recupero dei messaggi in segreteria. Al suo interno sarà presente la chiamata al metodo `getMessages()`.

### 5.1.9 CallHistoryPanelPresenter

#### Funzione

Presenter incaricato alla gestione dello storico chiamate.

#### Relazioni d'uso

Nessuna relazione d'uso evidenziata.

#### Attributi

**- `calls: String[]`**

Array che conterrà tutte le chiamate effettuate

**- `servlets: String[]`**

Array che contiene le tre URL delle *servlet* da contattare per i messaggi

#### Metodi

**- `getServletURLs(): void`**

Configura gli URL delle *servlet* da interrogare leggendoli dal file di configurazione in base alle operazioni che questo presenter deve essere in grado di compiere.

Più precisamente, si dovrà creare una "XMLHttpRequest" rivolta al file XML di configurazione dove sono riportate tutte le *servlet*, ricevuta la risposta si popolerà l'*array* dichiarato nella classe (*servlet*).

**- `getCalls(): String`**

Ottiene tutta la lista delle chiamate effettuate e tracciate nel *server*, una volta ricevuta risposta con relativi dati associati li utilizza come dati di ritorno. Per contattare la *servlet*, va usato come in precedenza "XMLHttpRequest".

**- `addListItem(call: String): void`**

Aggiunge alla lista delle chiamate visualizzata nel CallHistoryPanel una nuova voce che corrisponde alla chiamata ricevuta da parametro. Ogni voce della lista contiene i dati relativi alla chiamata e non consente di effettuare alcuna azione su di essa.



**+ createPanel(): DOM\_element**

Costruisce il pannello dello storico delle chiamate, che deve essere visualizzato all'interno del MainPanel dell'applicazione quando è selezionata la funzione corrispondente dal pannello degli strumenti. Il CallHistoryPanel è costituito da una semplice lista con tanti *list Item* quante sono le chiamate tracciate nel server. Tutti questi elementi vanno aggiunti all'elemento *element* ritornato poi dal metodo.

**+ setup(): void**

Metodo che ogni sua invocazione provoca il recupero dello storico chiamate, verrà quindi per prima cosa richiamato il metodo `getCalls()` e successivamente una chiamata al metodo `addListItem(call)` per ogni chiamata ritornata.

**5.1.10 SearchresultPanelPresenter****Funzione**

Presenter incaricato di gestire il pannello che visualizza i risultati di ricerca.

**Relazioni d'uso**

Nessuna relazione d'uso evidenziata.

**Attributi****- servlets: String[]**

Array che contiene le URL delle *servlet* da contattare per eseguire ricerche

**Metodi****- getServletURLs(): void**

Configura gli URL delle *servlet* da interrogare leggendoli dal file di configurazione in base alle operazioni che questo presenter deve essere in grado di compiere.

Più precisamente, si dovrà creare una "XMLHttpRequest" rivolta al file XML di configurazione dove sono riportate tutte le *servlet*. Ricevuta la risposta, si popolerà l'*array* dichiarato nella classe (*servlet*).

**- addListItem(list: DOM\_element, contact: Object(contact)): void**

Aggiunge alla lista ricevuta da parametro (*list*) il contatto anch'esso ricevuto da parametro (*contact*). L'aggiunta del contatto alla lista avviene prima di tutto effettuando un controllo sui dati da visualizzare nel *list item* che se presenti nel database saranno:

- nome;
- cognome;
- email;
- immagine profilo.

Infine, si assegna il comportamento corretto all'evento "onClick" che andrà a richiamare il metodo `onContactSelected(contact)` del MediatorPresenter.

**+ createPanel(): DOM\_element**

Costruisce il pannello dei risultati di ricerca, che deve essere visualizzato all'interno del MainPanel dell'applicazione quando viene effettuata una ricerca. Il SearchResultPanel è costituito da una semplice lista che verrà popolata con i risultati della ricerca.

```
+ displayContactList(contacts: Object(contact)): void
```

Metodo che visualizza all'interno del pannello una lista di contatti che ottenuta dal server a seguito di una ricerca. Tale metodo demanda la visualizzazione per ogni contatto richiamando il metodo `addListItem(userList, contact)`.

#### 5.1.11 GroupPanelPresenter

##### Funzione

Presenter incaricato di gestire i gruppi.

##### Relazioni d'uso

Nessuna relazione d'uso evidenziata.

##### Attributi

Nessun attributo presente.

##### Metodi

```
- addListItem(list: DOM_element, group: Object(group)): void
```

Aggiunge alla lista ricevuta come parametro (*list*) un gruppo, anch'esso ricevuto come parametro (*group*) creando un `listItem` (`<li>`). Il *list item* sarà composto da uno *span* con il nome del gruppo e vicino dovrà apparire un'immagine per l'eliminazione del gruppo stesso. Va quindi gestito il comportamento all'evento "onclick" su tale immagine che comporterà l'eliminazione del gruppo. Prima di procedere alla vera e propria eliminazione, dovrà apparire all'utente un messaggio di conferma che chiede se effettivamente vuole cancellare il gruppo: se l'utente accetta, viene richiamato il metodo del MediatorPresenter `onGroupRemoved(group)` che procedere ad eliminare il gruppo.

```
+ createPanel(): DOM_element
```

Costruisce il pannello per la gestione dei gruppi, che deve essere visualizzato all'interno del MainPanel dell'applicazione quando viene richiesta tale operazione. Il GroupPanel è costituito da una semplice lista contenente tanti *list item* quanti sono i gruppi dell'utente.

```
+ displayGroupList(groups: Object(group)): void
```

Metodo che visualizza all'interno del pannello una lista di gruppi, tale metodo demanda la visualizzazione di ogni singolo gruppo richiamando il metodo `addListItem(groupList, group)`.

#### 5.1.12 ToolsPanelPresenter

##### Funzione

Presenter incaricato di gestire il pannello degli strumenti.

##### Relazioni d'uso

Nessuna relazione d'uso evidenziata.

### Attributi

- **servlets:** `String[]`  
Array che contiene le URL delle *servlet* relative a tale presenter
- **element:** `DOM_element`  
elemento di tipo DOM che rappresenta l'intero pannello

### Metodi

- **getServletURLs():** `void`  
Configura gli URL delle *servlet* da interrogare leggendoli dal file di configurazione in base alle operazioni che questo presenter deve essere in grado di compiere.  
Più precisamente, si dovrà creare una “XMLHttpRequest” rivolta al file XML di configurazione dove sono riportate tutte le *servlet*. Ricevuta la risposta, si popolerà l'*array* dichiarato nella classe (*servlet*).
- + **initialize():** `void`  
Inizializza il pannello degli strumenti dell'applicazione rendendolo visibile. Tale pannello conterrà link che rimandano a tutte le funzioni disponibili per l'operazione scelta. In particolare:
  - sezione che rimanda alle funzionalità della segretaria;
  - sezione che rimanda alle impostazioni dell'account;
  - sezione che rimanda alle funzionalità dello storico chiamate;
  - sezione che rimanda alle funzionalità della gestione dei contatti.Tutte queste sezioni saranno espresse come list item di una lista. All'evento “onclick” su ogni sezione corrisponderà l'invocazione del metodo che rimanda al presenter relativo alla sezione.
- + **hide():** `void`  
Rende invisibile il pannello degli strumenti. Va impostata la proprietà *display* di tale elemento a false.
- + **logout():** `void`  
Effettua il *logout* dell'utente dal sistema, per far ciò viene invocata la *servlet* relativa.

#### 5.1.13 AccountSettingsPanelPresenter

##### Funzione

Presenter incaricato di gestire il pannello delle impostazioni dell'utente.

##### Relazioni d'uso

Nessuna relazione d'uso evidenziata.

### Attributi

- **servlets:** `String[]`  
Array che contiene le URL delle *servlet* relative a tale presenter

## Metodi

### + `createPanel(): DOM_element`

Inizializza il pannello costruendone i *widget* grafici interni e lo restituisce in modo che possa essere inserito all'interno del pannello principale. Più dettagliatamente, verrà costruita una lista con *list item* contenenti le informazioni dell'utente, ossia:

- nome;
- cognome;
- e-mail;
- immagine profilo;
- password;
- domanda segreta;
- risposta alla domanda segreta.

Infine, sarà presente un pulsante con *id* uguale "changeButton" il quale all'evento "on-click" richiama il metodo `onChangeButtonPressed()`, descritto successivamente, che dà la possibilità di modificare i dati.

### - `onChangeButtonPressed(): void`

Gestisce la pressione del pulsante "changeButton" che trasforma la lista di elementi testuali in un *form* da compilare per modificare i propri dati. Il *form* è completo di un pulsante *submit* che attiva "onSubmitChange", metodo descritto successivamente.

I campi modificabili, recuperati dall'oggetto presente in `CommunicationCenter` (`my`) che rappresenta il profilo utente, sono:

- nome;
- cognome;
- e-mail;
- immagine profilo;
- password;
- domanda segreta;
- risposta alla domanda segreta.

### - `onSubmitChange(): void`

Gestisce il cambiamento dei dati da parte dell'utente contattando, se necessario (controllo effettuato tramite chiamata al metodo `hasSomethingChanged`), la *servlet* incaricata di aggiornare il contenuto del database sul server. Il metodo va a recuperare i valori tramite chiamate di metodo `document.getElementById(id)` e successivamente contatta la *servlet* tramite un `XMLHttpRequest` passando i dati prelevati dalla pagina. Se la *servlet* ritorna un errore viene gestito tramite la visualizzazione dell'errore all'utente, altrimenti viene richiamato il metodo `initialize()` per aggiornare la visualizzazione dei dati più recenti.

### - `hasSomethingChanged(data: String[]): boolean`

Metodo che verifica l'effettivo cambiamento di dati rispetto a quanto contenuto in `communicationcenter.my` confrontandoli con l'oggetto `data` ricevuto come parametro. Il metodo controlla che almeno il valore di un campo dati sia effettivamente cambiato e ritorna *true* in tale caso. Nel caso opposto, ossia tutti i dati sono uguali ai precedenti, ritorna *false*.

- `buildQueryString(data: String[]): String`

Costruisce la stringa corrispondente alla *query* che deve essere spedita alla *servlet* per portare a termine la richiesta di cambiamento dei dati personali. Viene costruita partendo dai valori recuperati dall'oggetto `data` passato come parametro e utilizzando la giusta sintassi MySQL.

## 5.2 Package `org.softwaresynthesis.mytalk.clientpresenter.kernel`

### 5.2.1 CommunicationCenter

#### Funzione

Classe logica che gestisce tutta la parte della comunicazione lato *client*.

#### Relazioni d'uso

Nessuna relazione d'uso evidenziata.

#### Attributi

- + `videoCommunication: String[]`

Array che contiene i dati della video-chiamata

- + `openChat: DOM_element[]`

Array che contiene i dati della video-chiamata

- `urlServlet: String`

Contiene l'indirizzo della *servlet* che gestisce la comunicazione

- `my: String[]`

Array contenente i dati personali dell'utente

- `websocket: Object()`

Oggetto che rappresenta la *websocket* da utilizzare per comunicare con il server. Il costruttore di tale oggetto richiede l'URL della *servlet*.

#### Metodi

- `formatBytes(bytes: int): String`

Metodo che formatta i byte ricevuti ed inviati al fine di fornire una visualizzazione sensata delle statistiche.

Il metodo riceve un numero rappresentante i *bytes* (`bytes`) e li converte in KB/s o MB/s a seconda della grandezza.

- `formatTime(tempo: int): String`

Metodo che ritorna nel formato "hh:mm:ss" il tempo della comunicazione.

Il metodo riceve un numero rappresentante il tempo espresso in secondi (`tempo`), lo converte nel formalismo descritto prima e lo restituisce.

- `stopTimer(): void`

Metodo che ferma l'aggiornamento del timer, viene richiamato quando la chiamata termina.

- `stopStats(): void`

Metodo che ferma l'aggiornamento delle statistiche relative alla chiamata in corso, viene richiamato quando la chiamata termina.

- `dumpStats(obj: Object()): void`

Metodo che permette l'estrazione dei dati rappresentanti le statistiche della chiamata, esso riceve come parametro un oggetto (`obj`), lo elabora, ed estrae solo i byte ricevuti e inviati. Ricavati questi dati, li visualizza nello *span* corrispondente.

- `gotDescription(): void`

Metodo utilizzato da WebRTC che imposta la propria descrizione e la invia al *client* chiamato in modo tale da poter instaurare la comunicazione. Verrà quindi prima di tutto utilizzata la funzione proprietaria di WebRTC `setLocalDescription` per impostare la propria descrizione e successivamente, mediante la *websocket*, inviata al client peer chiamato.

+ `connect(): void`

Metodo utilizzato per la creazione della connessione con il *server*. Viene inizializzata la variabile `websocket` richiamando il costruttore con un parametro (URL della servlet). Successivamente viene specificato il comportamento della *websocket* per determinati eventi quali:

- **onopen**: istruzioni da eseguire quando la *websocket* viene aperta. In questo caso, deve essere inviato alla *websocket* il proprio *id* utente in modo tale da identificare univocamente il canale aperto con il server.
- **onclose**: istruzioni da eseguire quando la *websocket* viene chiusa per qualsiasi motivo, notificando il server di tale avvenimento.
- **onerror**: istruzioni da eseguire quando avviene un errore con la *websocket*. Tale evento verrà notificato all'utente con un messaggio d'errore che descrive l'errore e l'atteggiamento da seguire.
- **onmessage**: metodo più corposo in quanto gestisce l'unico evento richiamato mentre avviene una comunicazione dal server verso il *client*. Per diversificare i messaggi ricevuti, si è deciso di rappresentare il messaggio come un *array* nel quale il primo elemento identifica il tipo di richiesta. In questo modo, avremo tre tipi:
  - type 2: riceve e gestisce una richiesta di chiamata. Imposta quindi con il metodo nativo di WebRTC `setRemoteDescription` la descrizione del peer chiamante (secondo elemento dell'*array* ricevuto) e la comunicazione può iniziare.
  - type 3: riceve una richiesta con l'*id* del chiamante come dato. Viene memorizzato nel *client* in modo tale da conoscere il canale che il chiamante ha aperto con il server per future comunicazioni.
  - type 5: notifica il cambiamento di stato degli amici. I dati ricevuti contengono l'*id* dell'utente e relativo nuovo stato, si procede quindi a modificare lo stato dell'utente nella rubrica.

+ `disconnect(): void`

Metodo utilizzato per disconnettersi dal sistema, tale evento deve essere notificato al *server* tramite la *websocket* inviando un messaggio di tipo 4 contenente il proprio *id*.

+ `call(isCaller: boolean, contact: Object(contact)): void`

Metodo che gestisce la vera e propria chiamata tramite WebRTC.

Inizialmente, si inizializza la variabile `pc` che è di tipo `RTCPeerConnection`, oggetto reso disponibile dalle librerie di WebRTC, successivamente si definiscono i comportamenti associati agli eventi di `RTCPeerConnection`, ossia:

- **onicecandidate**: evento scatenato quando un nuovo peer si “candida” per poter chiamare, viene quindi inviata la propria “descrizione” all'altro peer.

- `onaddstream`: evento scatenato quando viene aggiunto uno *stream* nell'oggetto `RTCPeerConnection`. In questa situazione, il chiamato riceve lo *stream* del chiamante e lo visualizza nell'apposito tag video. In contemporanea, vengono richiamati i metodi che fanno partire la visualizzazione delle statistiche di chiamata.
- `onremovestream`: evento scatenato quando viene rimosso uno *stream* nell'oggetto `RTCPeerConnection`. In questa situazione, colui che effettua questa operazione vuole terminare la chiamata e deve quindi re-inviare la propria descrizione (attraverso il metodo `gotDescription()` in modo tale che l'altro `peer` possa aggiornare la descrizione remota. Vengono richiamati alla fine i metodi che fermano la visualizzazione delle statistiche di chiamata e chiudono il canale di comunicazione tramite la chiamata del metodo `close()` dell'oggetto `pc`.

+ `webkitGetUserMedia(audio: boolean, video: boolean): void`

Metodo richiamato nel momento in cui un utente vuole iniziare una chiamata. Il metodo, fornito dalle librerie native di *Google Chrome*, cattura lo *stream* della propria videocamera/microfono. Riceve due parametri che indicano se deve essere catturato audio e/o video. Procedendo, viene aggiunto lo *stream* all'oggetto `RTCPeerConnection` tramite il metodo `addStream(localstream)` fornito dalle librerie di WebRTC.

+ `endCall(): void`

Metodo richiamato per terminare una chiamata, a cascata viene richiamato il metodo `removeStream(localstream)` e inviata la nuova descrizione al `peer` remoto.

## 6 Specifica sotto-architettura clientview

Il sistema, dovendo essere contenuto in un'unica pagina web, dispone di un'unica vista che viene modificata all'occorrenza dai vari presenter. Tale vista è suddivisa in tre parti:

- parte sinistra contenente il pannello della propria rubrica;
- parte destra contenente il pannello degli strumenti;
- parte centrale contenente il MainPanel.

Quest'ultimo, il MainPanel, è designato a cambiare il suo contenuto in base alle operazioni svolte dall'utente in quel dato momento. Ad esempio, se l'utente seleziona un contatto dalla propria rubrica, il pannello centrale conterrà la visualizzazione della scheda del contatto selezionato. Se invece avvia una chiamata, vedrà visualizzato il pannello delle chiamate. La vista è quindi demandata tutta ai presenter che la modificano in base alle richieste.