Documentazione Progetto Ingegneria del Software

Lanza Matteo ()

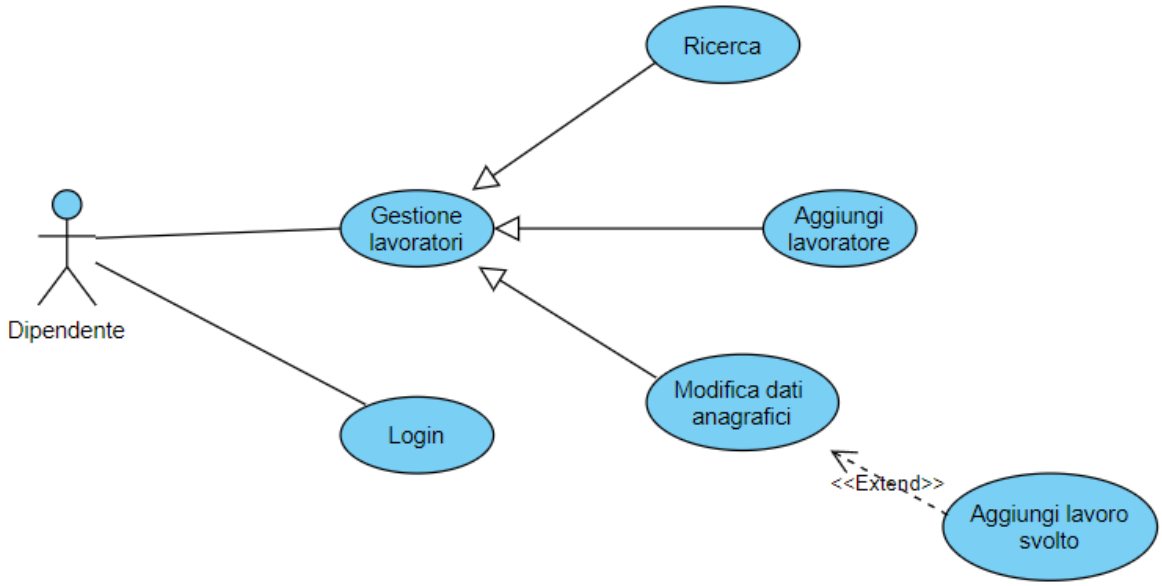
Faedo Giacomo ()

Zanetti Alex ()

INDICE

USE CASE E SCHEDE DI SPECIFICA

Il software viene utilizzato dai dipendenti o responsabili dell'agenzia. I dati dei dipendenti vengono inseriti nel database dagli amministratori, i quali forniscono anche l'username e la password ai dipendenti per effettuare il login. Successivamente se i dati inseriti per il login sono corretti i dipendenti si ritrovano davanti una schermata con il menù principale dal quale possono scegliere quello che vogliono fare: ricercare i lavoratori, aggiungere i lavoratori e modificare i dati.



DIPENDENTE: CASI D'USO E SCHEDE DI SPECIFICA

Il dipendente o responsabile si deve autenticare effettuando il login (i dati del login vengono assegnati dagli amministratori del sistema), successivamente deve scegliere quello che vuole fare scegliendo una delle voci del menù principale.

RICERCA

Il dipendente o responsabile può effettuare una ricerca sui lavoratori presenti nel database in base a determinati parametri relativi ai lavoratori. I paramenti possono essere legati tra loro tramite anche ricerche complesse con AND/OR. È stato aggiunto anche il comando SIMILAR che, se attivo, permette di cercare nei campi di tipo testo se è presente una singola lettera all'interno di quel testo.

Attori: Dipendente

Precondizioni: aver effettuato con successo il login, conoscere i parametri del lavoratore da ricercare

Passi:

1. Il responsabile del servizio preme sull'apposito pulsante "ricerca lavoratore" e accede alla pagina
2. Il responsabile inserisce le condizioni che la ricerca deve rispettare, eventualmente utilizzando anche gli appositi pulsanti and o or
3. Il responsabile preme il pulsante di ricerca

Postcondizioni: il sistema deve aver correttamente restituito a schermo i lavoratori rispettanti i filtri

INSERIMENTO LAVORATORE

In questa voce possiamo inserire tutti i dati di un nuovo lavoratore. L'identificativo univoco del lavoratore è gestito dal sistema. Ad un singolo lavoratore posso associare più zone dove può lavorare, più lingue conosciute, più esperienza effettuate, più tipi di patenti e almeno una persona da chiamare in caso di emergenza. Prima di confermare l'inserimento, viene aperta una pagine per ricontrollare i dati inseriti.

Attori: Dipendente

Precondizioni: aver effettuato con successo il login, avere ricevuto i dati anagrafici dal lavoratore

Passi:

1. Il responsabile del servizio preme sull'apposito pulsante "inserisci lavoratore" e accede alla pagina
2. Il responsabile riempie le opportune caselle con i dati del lavoratore da aggiungere
3. Il responsabile conferma l'immissione dei dati tramite apposito pulsante

Postcondizioni: i dati del nuovo lavoratore devono risultare inseriti correttamente nel database

MODIFICA ANAGRAFICA

Il dipendente o responsabile può modificare i dati anagrafici dei lavoratori. La pagina permette di trovare il lavoratore che si desidera modificare attraverso una semplice ricerca, lo si seleziona e con il pulsante che si trova in fianco e si possono modificare tutti i dati del lavoratore andando a cambiare direttamente il campo. Con inserisci extra è possibile andare ad inserire o togliere altri dati dei lavoratori come esperienza, tipi di patenti, lingue conosciute, e infine comuni dove può lavorare.

Attori: Dipendente

Precondizioni: aver effettuato con successo il login, aver ricevuto i nuovi dati da sostituire

Passi:

1. Il responsabile del servizio preme sull'apposito pulsante "modifica anagrafica"
2. Il responsabile tramite gli opportuni parametri ricerca il lavoratore a cui deve modificare i dati
3. Il responsabile individua i campi da modificare e sostituisce i vecchi dati con quelli nuovi
4. Il responsabile preme il pulsante di conferma

Postcondizioni: il sistema deve aver correttamente registrato le modifiche sul database

INSERIMENTO LAVORO SVOLTO

Per trovare questa voce si accede prima a modifica anagrafica. Nell'inserimento del lavoro svolto possiamo inserire il lavoro svolto da un determinato lavoratore inserito nel database dall'agenzia. Il lavoro svolto è sempre associato ad un lavoratore già inserito nel database.

Attori: Dipendente

Precondizioni: aver effettuato con successo il login, aver ricevuto i dati del lavoro svolto da inserire

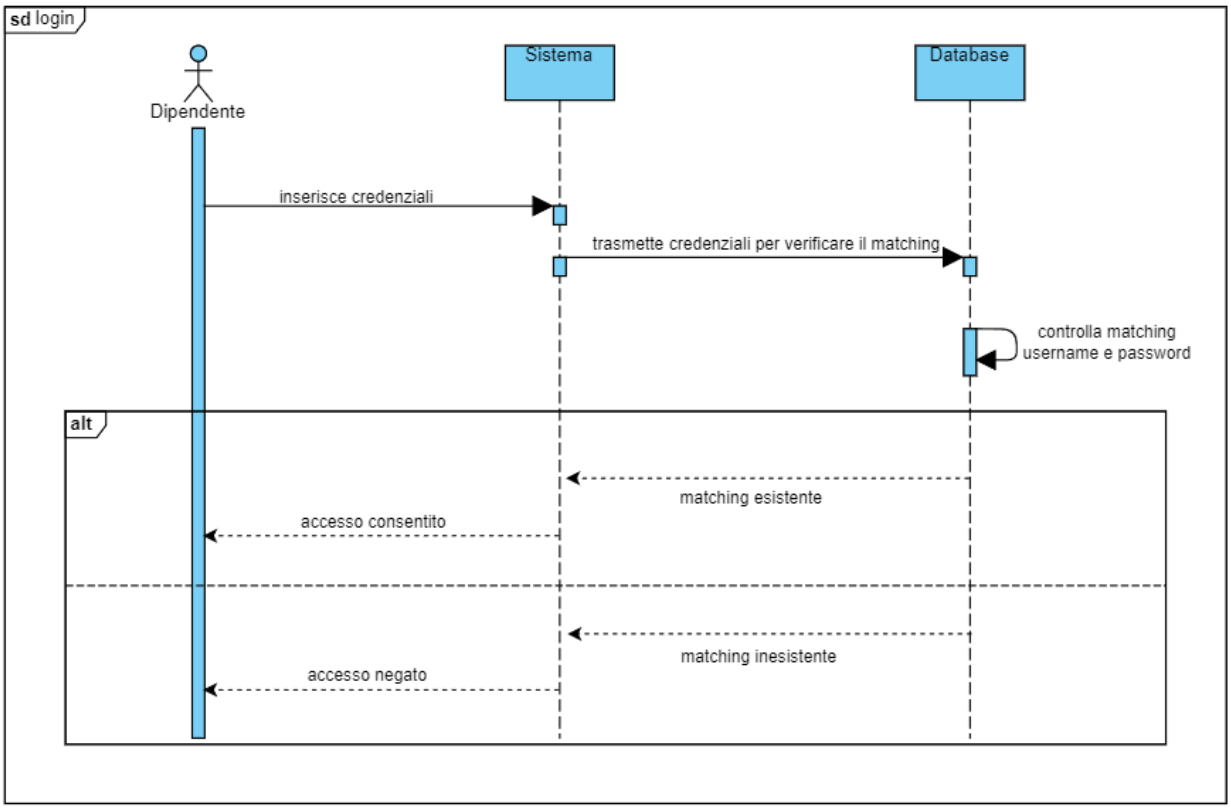
Passi:

1. Il responsabile del servizio preme sull'apposito pulsante "modifica anagrafica" e accede alla pagina
2. Il responsabile preme sull'apposito pulsante "aggiungi lavoro svolto"
3. Il responsabile inserisce i dati del lavoro svolto negli appositi campi
4. Il responsabile preme il pulsante di conferma

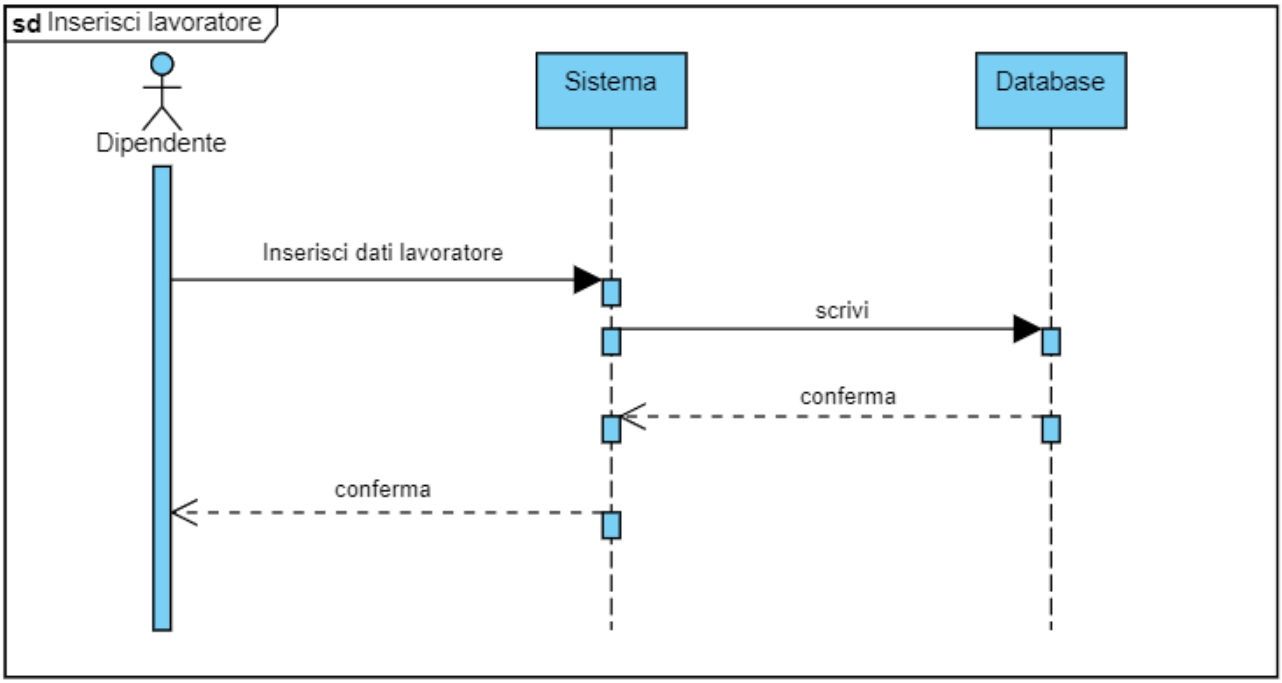
Postcondizioni: il sistema deve aver correttamente registrato il nuovo lavoro svolto sul database

SEQUENCE DIAGRAM DI DETTAGLIO PER GLI USE CASE

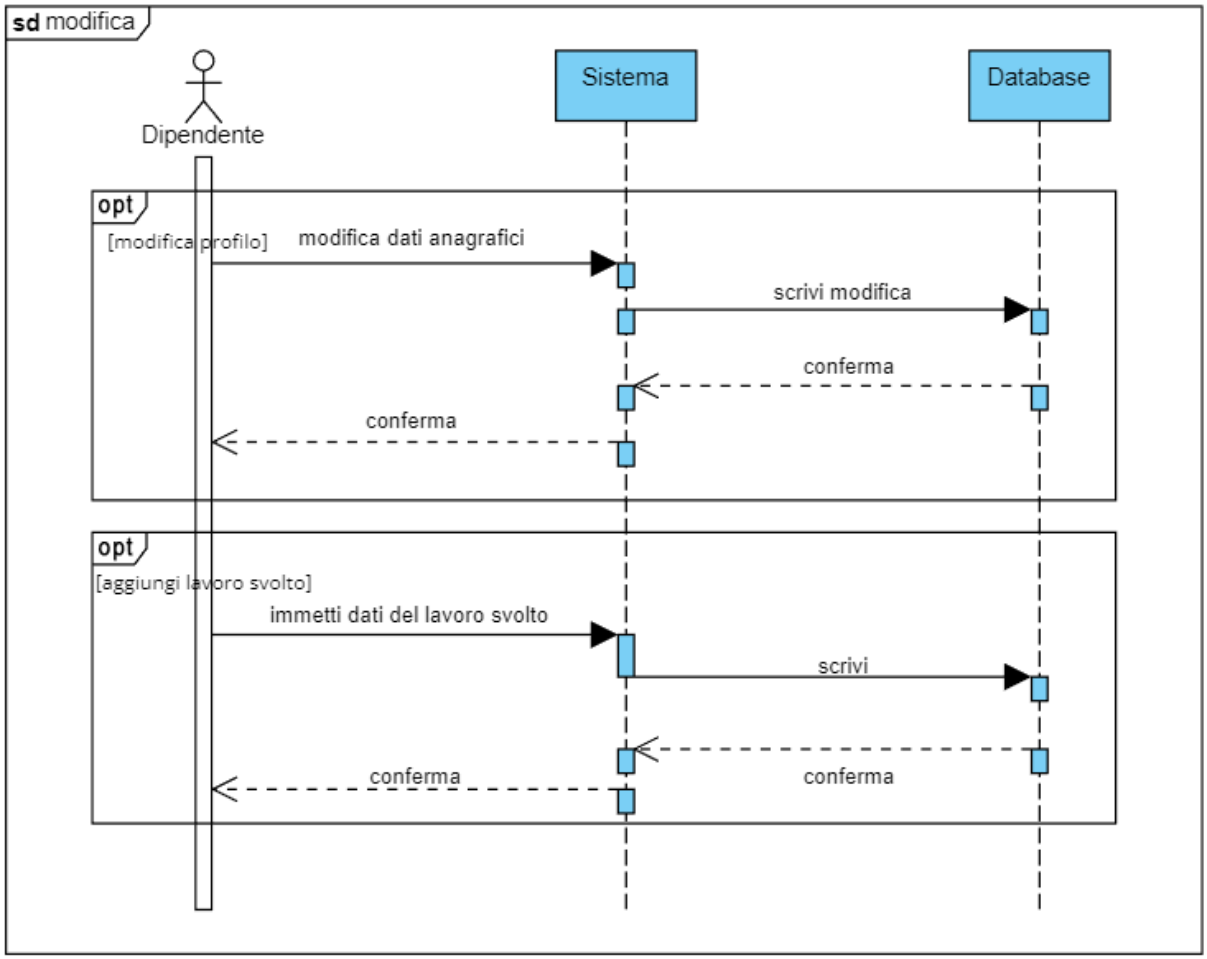
Login



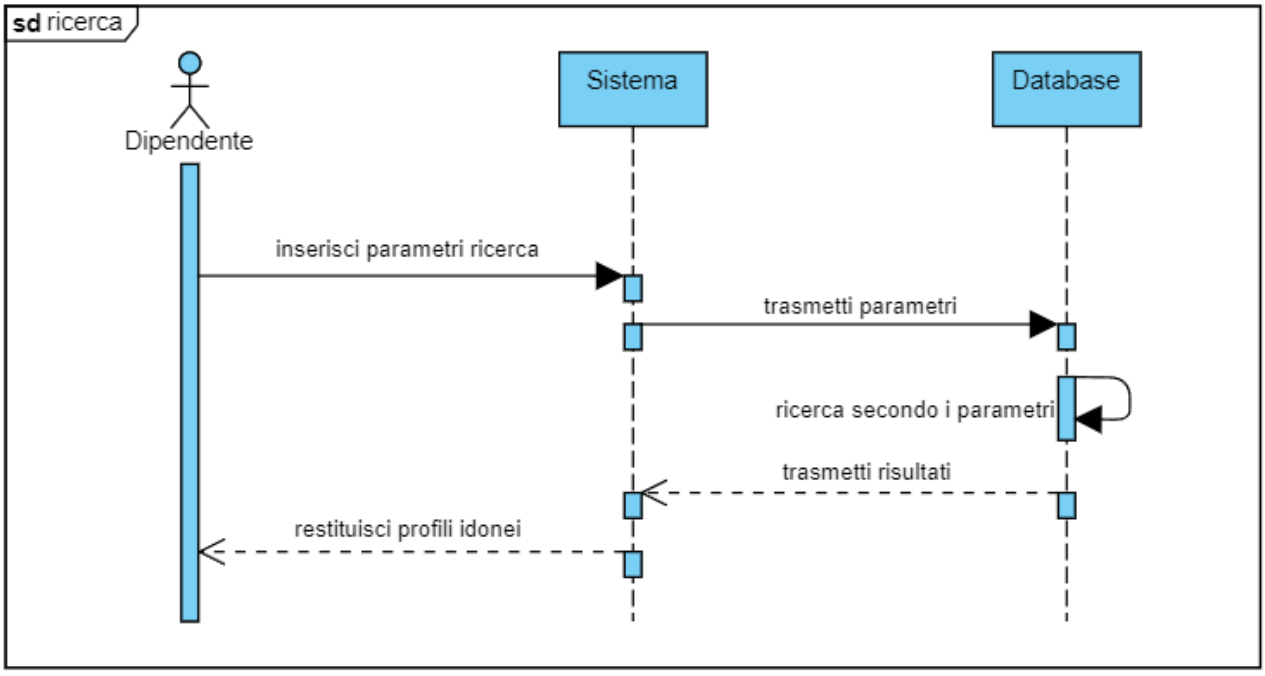
Inserisci lavoratore



Modifica dei dati anagrafici e inserisci il lavoro svolto

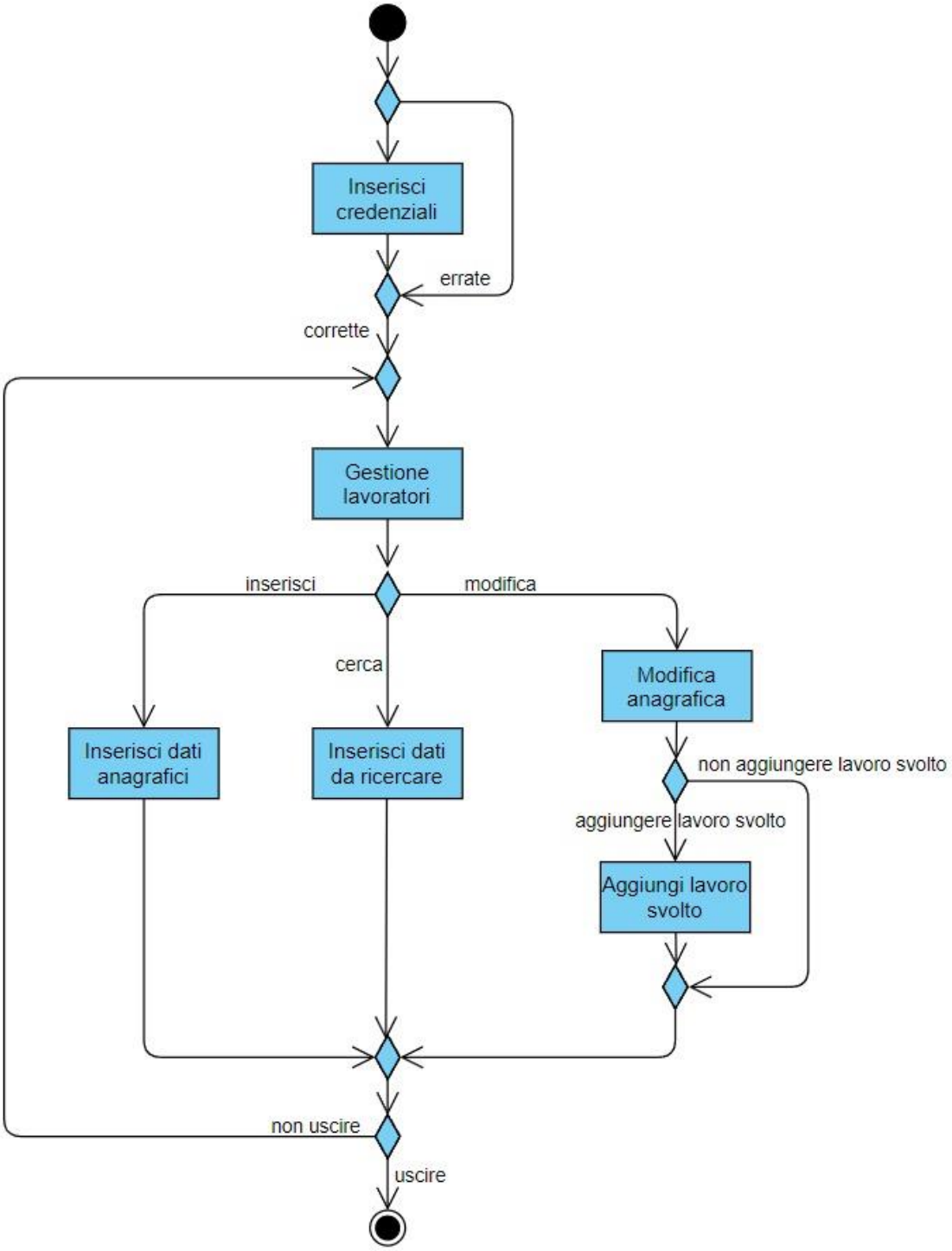


Ricerca dei lavoratori



ACTIVITY DIAGRAM RELATIVO ALLE MODALITÀ DI INTERAZIONE DEL SOFTWARE

Autenticazione e attività del responsabile/dipendente.

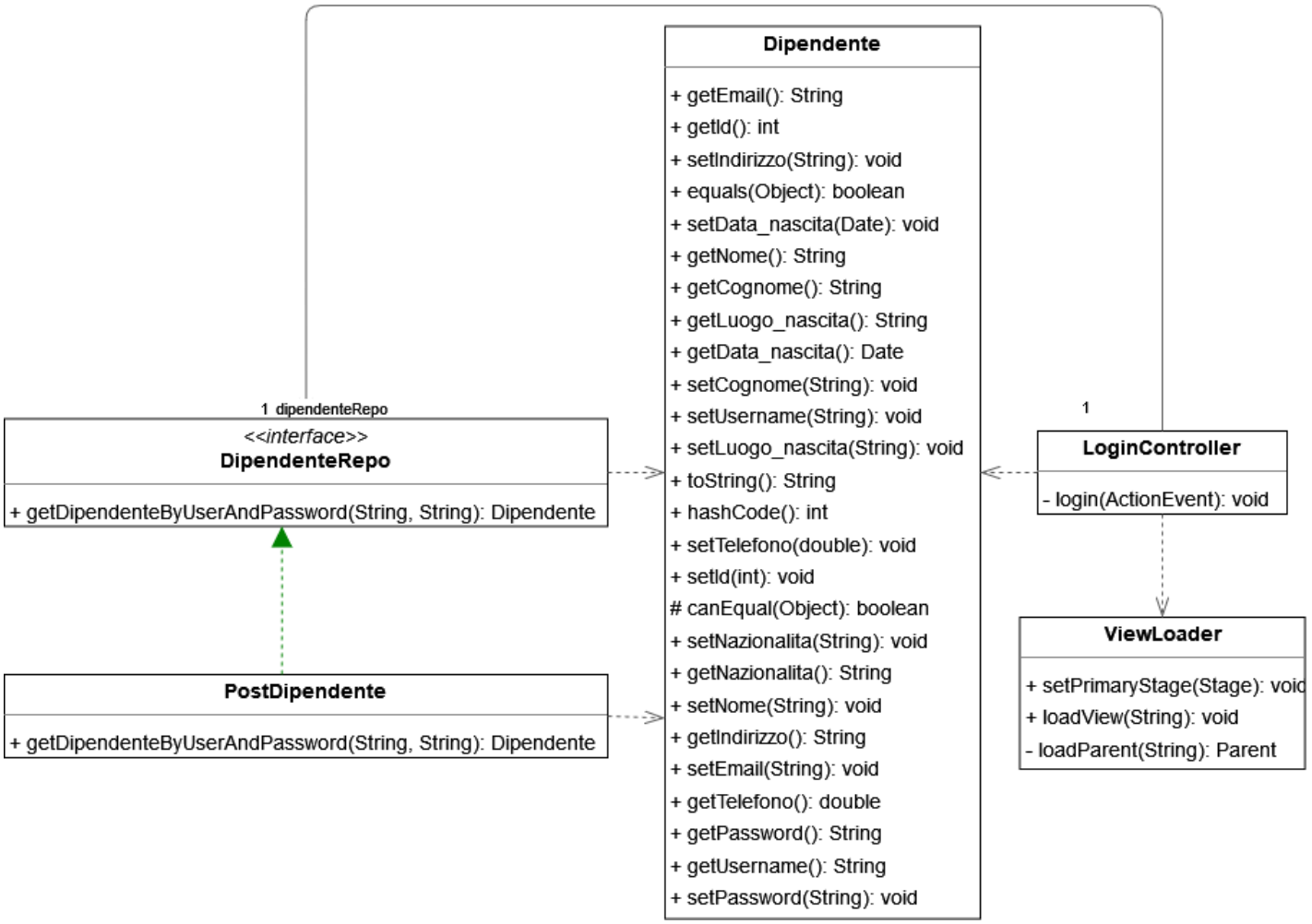


CLASS DIAGRAM E SEQUENCE DIAGRAM DEL SOFTWARE

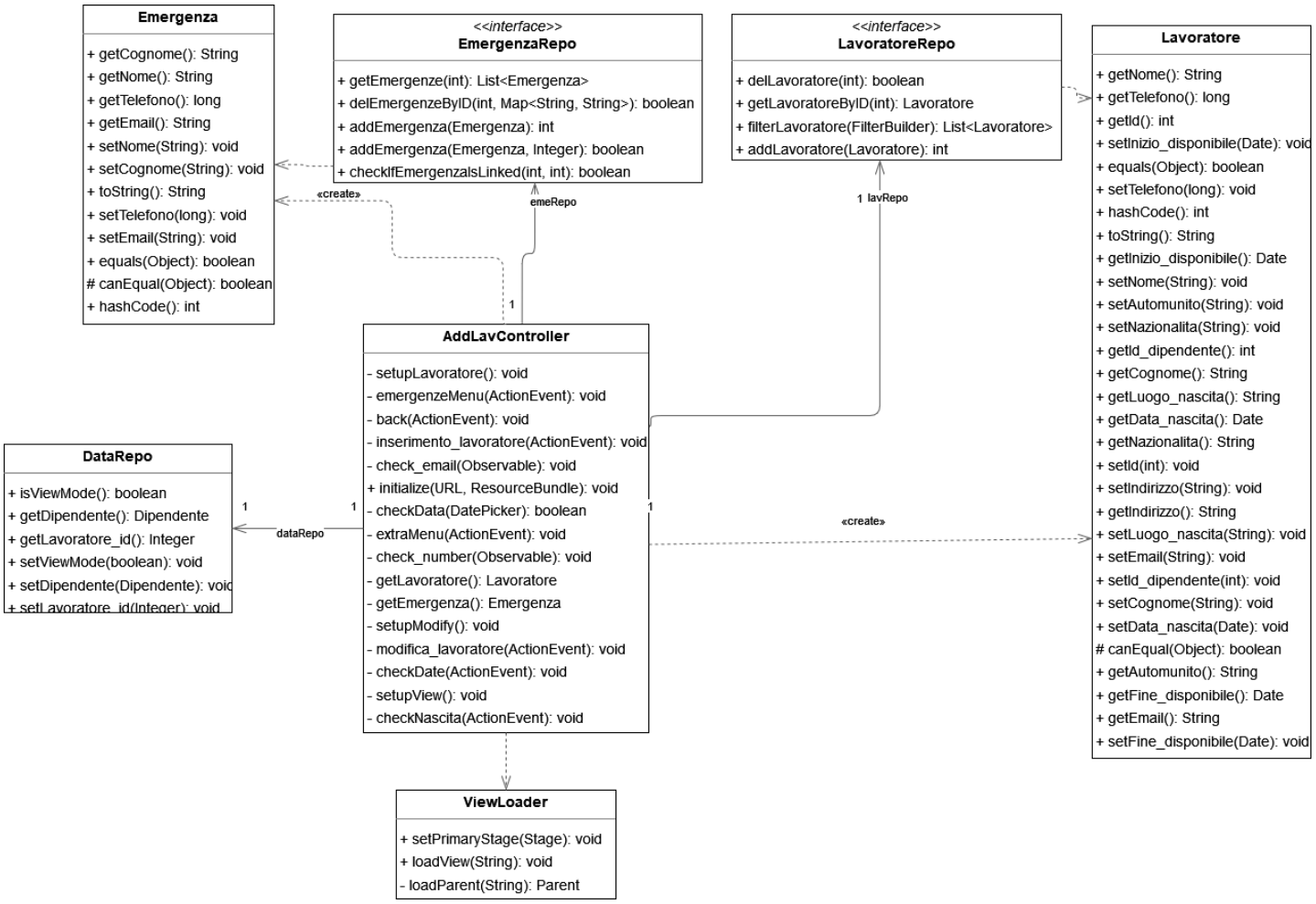
CLASS DIAGRAM DEL SOFTWARE

Per semplicità e per una maggiore chiarezza non abbiamo inserito il class diagram dell'intero software ma abbiamo inserito delle parti chiavi del nostro software.

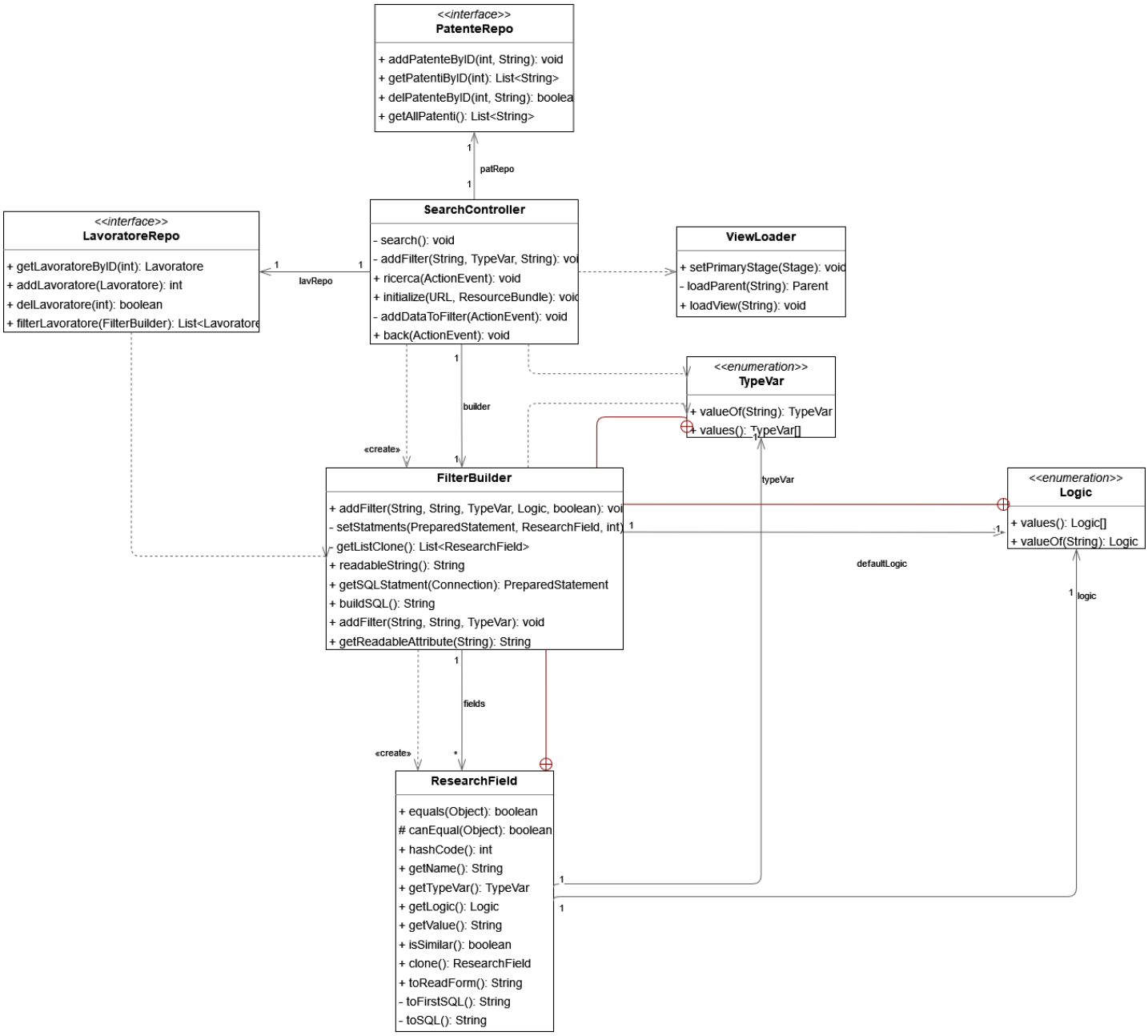
Login



Aggiungi lavoratore

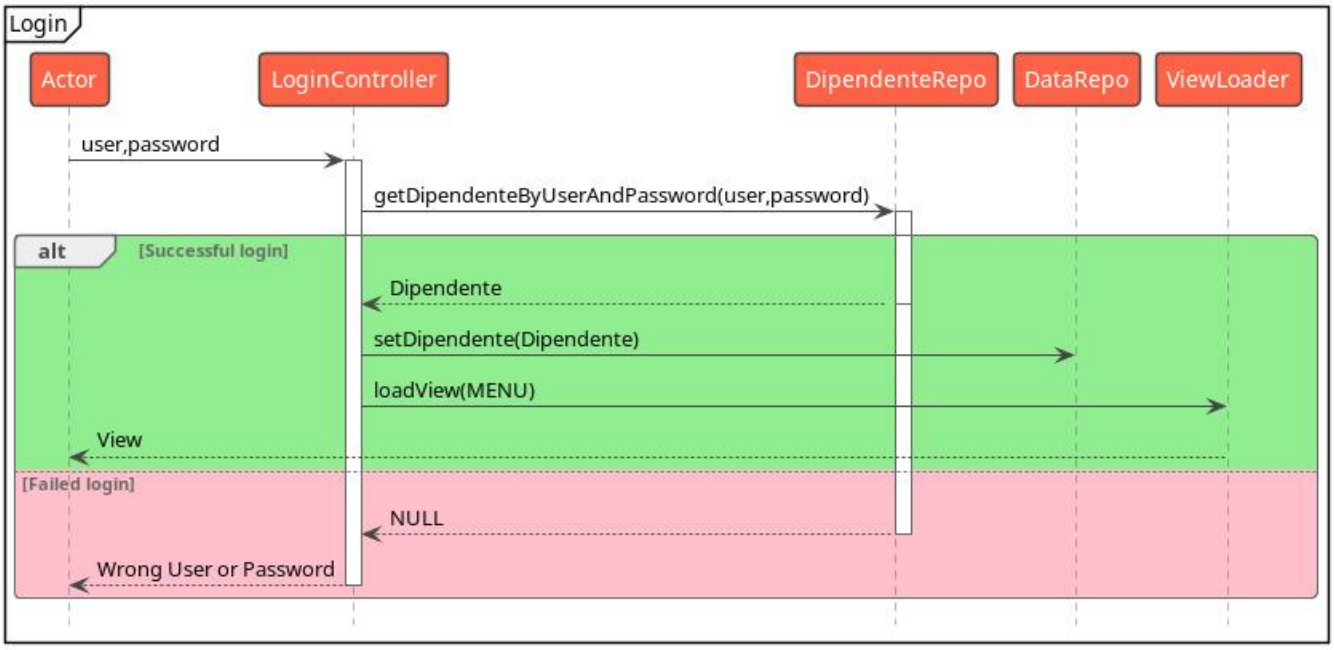


Ricerca

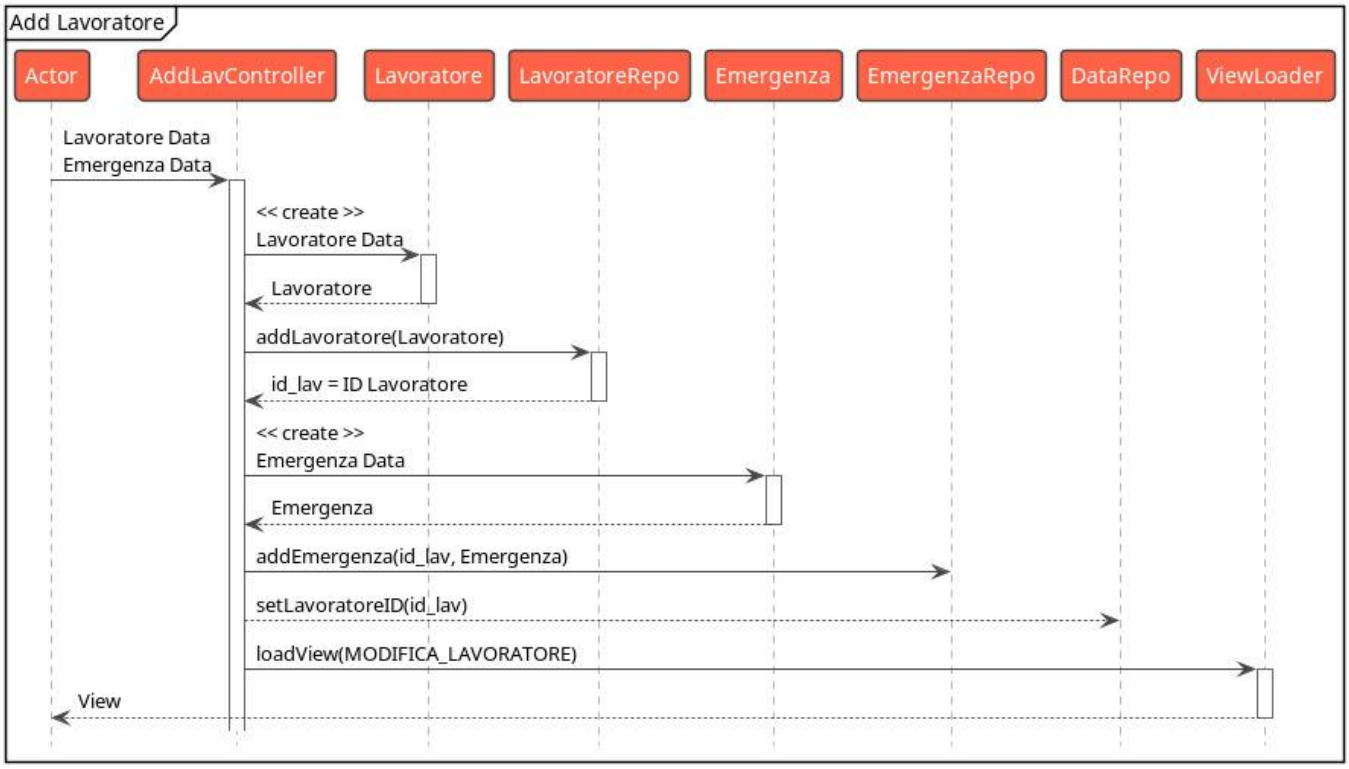

<<interface>>
PatenteRepo
+ addPatenteByID(int, String): void
+ getPatentiByID(int): List<String>
+ delPatenteByID(int, String): boolea
+ getAllPatenti(): List<String>
1
patRepo
1
SearchController
- search(): void
- addFilter(String, TypeVar, String): voi
+ ricerca(ActionEvent): void
+ initialize(URL, ResourceBundle): voic
- addDataToFilter(ActionEvent): void
+ back(ActionEvent): void
<<interface>>
LavoratoreRepo
+ getLavoratoreByID(int): Lavoratore
+ addLavoratore(Lavoratore): int
+ delLavoratore(int): boolean
+ filterLavoratore(FilterBuilder): List<Lavoratore
1
1
lavRepo
ViewLoader
+ setPrimaryStage(Stage): void
- loadParent(String): Parent
+ loadView(String): void
1
builder
<<enumeration>>
TypeVar
+ valueOf(String): TypeVar
+ values(): TypeVar[]
«create»
1
1
typeVar
FilterBuilder
+ addFilter(String, String, TypeVar, Logic, boolean): voi
- setStatments(PreparedStatement, ResearchField, int)
- getListClone(): List<ResearchField>
+ readableString(): String
+ getSQLStatment(Connection): PreparedStatement
+ buildSQL(): String
+ addFilter(String, String, TypeVar): void
+ getReadableAttribute(String): String
1
1
-1
<<enumeration>>
Logic
+ values(): Logic[]
+ valueOf(String): Logic
defaultLogic
1
logic
1
fields
«create»
*
ResearchField
+ equals(Object): boolean
# canEqual(Object): boolean
+ hashCode(): int
+ getName(): String
+ getTypeVar(): TypeVar
+ getLogic(): Logic
+ getValue(): String
+ isSimilar(): boolean
+ clone(): ResearchField
+ toReadForm(): String
- toFirstSQL(): String
- toSQL(): String
1
1

SEQUENCE DIAGRAM DEL SOFTWARE

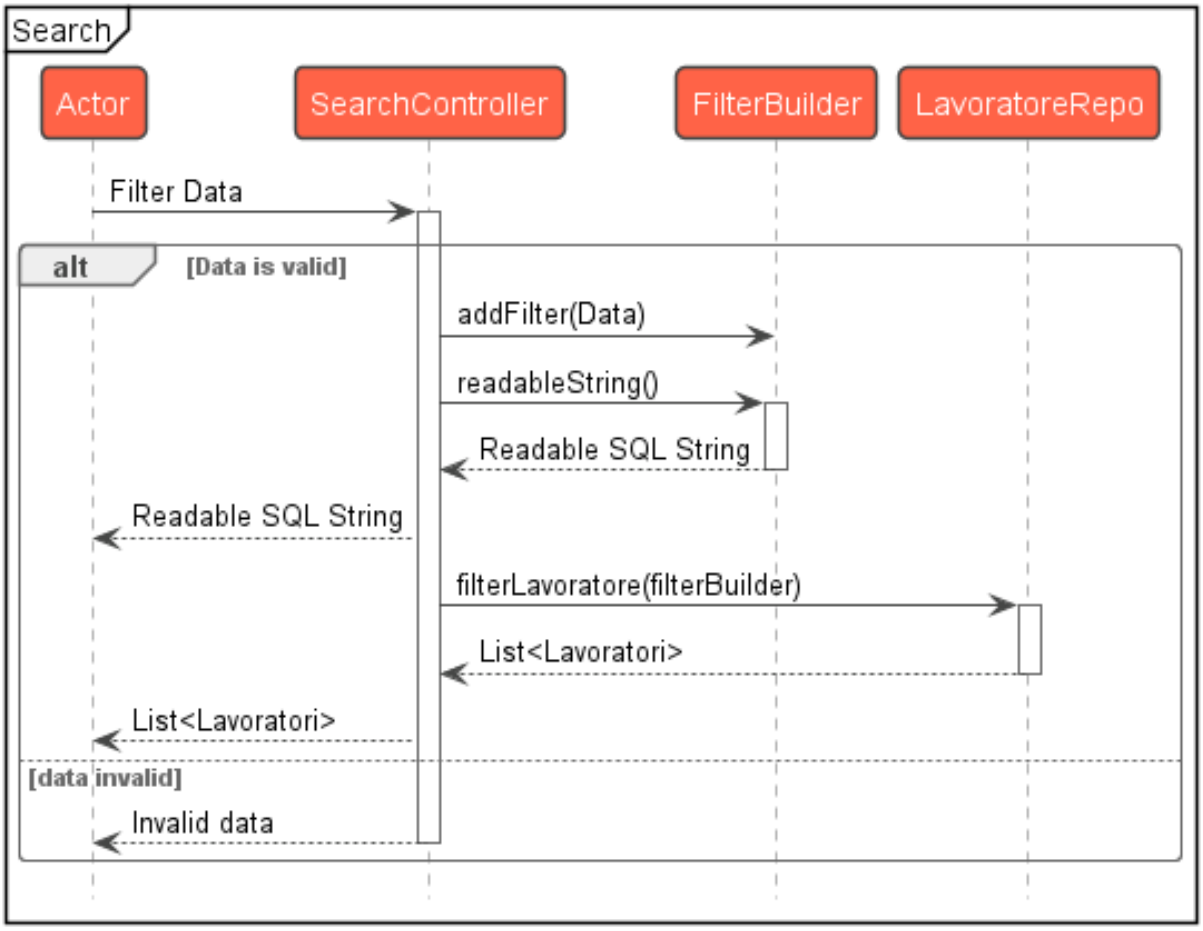
Login



Inserimento del lavoratore



Ricerca


Search
Actor
SearchController
FilterBuilder
LavoratoreRepo
Filter Data
alt
[Data is valid]
addFilter(Data)
readableString()
Readable SQL String
Readable SQL String
filterLavoratore(filterBuilder)
List<Lavoratori>
List<Lavoratori>
[data invalid]
Invalid data

ATTIVITÀ DI TEST

L'attività di test è stata sviluppata nelle seguenti fasi:

1. Test da parte degli sviluppatori
2. Test con JUnit
3. Test da parte di utenti generici

TEST DA PARTE DEGLI SVILUPPATORI

Gli sviluppatori hanno immesso nel sistema degli input (corretti e non) per vedere se la reazione del software fosse quella attesa.

Login

- Corretto funzionamento dell'autenticazione: se i dati inseriti (username e password) sono errati viene dato il messaggio di errore, al contrario a fronte di dati corretti inseriti dall'utente viene effettuato il corretto accesso. Username e password sono inseriti dagli amministratori nella base di dati (marco01, 12345),(errore: marco18,45789).

Aggiungi lavoratori

- Verificato che viene mostrato un errore se la data di nascita è successiva al giorno attuale (20/08/2022).
- Numero di telefono: inserimento di soli numeri, non accetta altri caratteri.
- Nell'inserimento della e-mail ci deve essere un testo prima della @ (che deve essere presente) e deve essere presente anche un punto, altrimenti viene dato un errore (il controllo viene fatto attraverso un regex).
- La data di inizio della disponibilità non può essere dopo della data di fine della disponibilità, altrimenti viene dato un errore (01/08/2022, 25/08/2022).
- Emergenze telefono ed e-mail, controlli uguali a quelli precedenti (marco@gmail.com, errore: marcogmail.com/marco@gmail).
- Pulsante "Inserisci lavoratore": il lavoratore viene inserito correttamente nel database se i dati inseriti sono corretti.
- Almeno una persona per le emergenze deve essere inserita per un lavoratore, altrimenti c'è un messaggio di errore.

Modifica anagrafica

- Non è possibile inserire delle lettere per cercare l'id del dipendente.
- Verificato che sia possibile fare la ricerca con uno o più attributi.
- Verificato che la data di inizio non possa essere dopo la data di fine, altrimenti viene dato un errore (01/09/2022, 20/09/2022).
- Verificato che l'inserimento degli attributi extra del lavoratore (esperienze, tipi di patenti, lingue conosciute e infine i comuni dove poter lavorare) possano essere inseriti e tolti in modo corretto.
- Verificato che si possa aggiungere in modo corretto nuove persone da chiamare in caso di emergenza e che ce ne debba essere almeno una (non è possibile cancellare l'ultima persona da chiamare in caso di emergenza).
- Verificato i controlli, già descritti prima, sull'inserimento del telefono e della e-mail.
- Verificato che tutti i campi devono essere riempiti, altrimenti viene dato un errore.

Aggiungi lavoro svolto

- Verificato che premendo su "AGGIUNGI LAVORO" il lavoro viene aggiunto al primo lavoratore della lista ottenuta dalla ricerca, oppure al lavoratore che viene selezionato dalla lista.
- Verificato che tutti i campi devono essere riempiti per poter aggiungere correttamente il lavoro.
- Verificato che la data di inizio non può essere dopo la data di fine, altrimenti viene dato un errore (01/09/2022, 20/09/2022).
- Verificato che il lavoro inserito deve essere stato svolto negli ultimi cinque anni (01/09/2021, 20/11/2006).

Ricerca

- Verificato che la data di inizio della disponibilità non può essere dopo della data di fine della disponibilità, altrimenti viene dato un errore.
- Verificato che premendo su "ADD" su un campo vuoto non dia errore o un'eccezione ed è stato verificato che aggiunge correttamente il filtro corrispondente nella ricerca.
- Verificato che premendo su "UNDO" cancelli sempre l'ultima azione effettuata (inserimento di un parametro).

- Verificato che la ricerca funzioni correttamente con il parametro nome (Silvia, Marco).
- Verificato che la ricerca funzioni correttamente con il parametro cognome (Bruni, Rossi).
- Verificato che la ricerca funzioni correttamente con il parametro lingua (italiano, inglese).
- Verificato che la ricerca funzioni correttamente con il parametro mansione (Bagnino, Programmatore).
- Verificato che la ricerca funzioni correttamente con il parametro comune (Verona, Mantova).
- Verificato che la ricerca funzioni correttamente con i parametri fine ed inizio disponibilità.
- Verificato che la ricerca funzioni correttamente con il parametro automunito (SI/NO).
- Verificato che la ricerca funzioni correttamente con il parametro patente(A2,A1,B1).
- Verificato che i pulsanti “AND”,”OR” e ”Similar” funzionino in modo corretto nella ricerca dei lavoratori.
- Verificato che premendo su “CLEAR” cancelli sempre il contenuto della tabella ed i risultati della ricerca e la stringa di ricerca.

TEST CON JUNIT

È stato fatto il test automatizzato con JUnit di alcune classi del codice ritenute più significative e problematiche.

“I Test JUNIT si focalizzano sul testing dei componenti che modificano i dati”

Classe PostLavoratore

```
@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
class PostLavoratoreTest {

   static Lavoratore lavoratore;
   static MainRepo repo;
   static LavoratoreRepo repoLav;
   static EmbeddedPostgres db;

   @BeforeAll
   static void setUp() {
      repo = setupTempDB();
      repoLav = repo.getLavoratoreRepo();
   }
   @AfterAll
```

```
static void tearDown() throws IOException {
   db.close();
}

@SneakyThrows
private static MainRepo setupTempDB() {
   db = EmbeddedPostgres.builder()
         .start();
   URL url = PostLavoratoreTest.class.getResource("/schema.sql");
   assert url != null;
   String schema = new String(Files.readAllBytes(Path.of(url.toURI())));
   db.getPostgresDatabase().getConnection().createStatement().execute(schema);
   return new PostDriver(db.getPostgresDatabase());
}

@Test
@Order(1)
void addLavoratore() {
   lavoratore = new Lavoratore();
   lavoratore.setId_dipendente(1);
   lavoratore.setNome("Test1");
   lavoratore.setCognome("Testing");
   lavoratore.setLuogo_nascita("LAst");
   lavoratore.setData_nascita(Date.valueOf(LocalDate.EPOCH));
   lavoratore.setNazionalita("IT");
   lavoratore.setIndirizzo("DIDID");
   lavoratore.setTelefono(2022233455);
   lavoratore.setEmail("te@te.com");
   lavoratore.setAutomunito("SI");
   lavoratore.setInizio_disponibile(Date.valueOf(LocalDate.EPOCH));
   lavoratore.setFine_disponibile(Date.valueOf(LocalDate.now()));
   int id = 0;
   try {
      id = repoLav.addLavoratore(lavoratore);
   } catch (SQLException e) {
      Assertions.fail();
   }
   lavoratore.setId(id);
}

@Test
@Order(2)
void getLavoratoreByID() {
   Lavoratore tmpLav = repoLav.getLavoratoreByID(lavoratore.getId());
   Assertions.assertEquals(tmpLav, lavoratore);
}

@Test
@Order(3)
void updateLavoratore() {
   Lavoratore tempLav = lavoratore.clone();
   String name = "AL";
```

```
        tempLav.setNome(name);
        try {
            repoLav.updateLavoratore(tempLav);
        } catch (SQLException e) {
            Assertions.fail();
        }
        Assertions.assertEquals(repoLav.getLavoratoreByID(lavoratore.getId()).getNome(), name);
    }

    @Test
    @Order(4)
    void filterWrongLavoratore() {
        FilterBuilder builder = repo.getFilterBuilderInstance();
        builder.addFilter("nome", "Test1", FilterBuilder.TypeVar.STRING, FilterBuilder.Logic.AND, false);
        Assertions.assertEquals(repoLav.filterLavoratore(builder).size(), 0);
    }

    @Test
    @Order(5)
    void filterLavoratore() {
        FilterBuilder builder = repo.getFilterBuilderInstance();
        builder.addFilter("nome", "AL", FilterBuilder.TypeVar.STRING, FilterBuilder.Logic.AND, false);
        String name = repoLav.filterLavoratore(builder).get(0).getNome();
        Assertions.assertEquals(name, "AL");
    }

    @Test
    @Order(6)
    void delLavoratore() {
        repoLav.delLavoratore(lavoratore.getId());
        Assertions.assertNull(repoLav.getLavoratoreByID(lavoratore.getId()));
    }
}
```

Classe ValidateData

```
class ValidateDataTest {

    ValidateClass object;

    @BeforeEach
    void setUp() {
        object = new ValidateClass();
        object.setName("Test");
        object.setOptional("Optional");
        object.setPhone(1545855977);
        object.setEmail("test@test.com");
    }

    @Test
    void valid() {
        Assertions.assertTrue(object.validate());}
```

```
    @Test
    void invalidNull() {
        object.setName(null);
        Assertions.assertFalse(object.validate());
    }

    @Test
    void invalidPhone() {
        object.setPhone(555545);
        Assertions.assertFalse(object.validate());
    }

    @Test
    void invalidMail() {
        object.setEmail("test@");
        Assertions.assertFalse(object.validate());
    }

    @Test
    void validOptional() {
        object.setOptional(null);
        Assertions.assertTrue(object.validate());
    }
}
```

Classe Mappable

```
class MappableTest {

    MappableClass object;
    Map<String,String> map;

    @BeforeEach
    void setUp() {
        object = new MappableClass();
        object.setName("Test");
        object.setPhone(123456789);
        object.setDate(LocalDate.now());
        object.setPrices(List.of(3.4f, 5.6f, 7.8f));
        map = object.toMap();
    }

    @Test
    void stringMap() {
        assertEquals(map.get("name"),object.getName());
    }

    @Test
    void numberMap() {
        assertEquals(map.get("phone"),String.valueOf(object.getPhone()));
    }
```

```
    @Test
    void complexMap() {
        assertEquals(map.get("date"),object.getDate().toString());
    }

    @Test
    void listMap() {
        assertEquals(map.get("prices"),object.getPrices().toString());
    }

    @Test
    void nullTest(){
        object.setName(null);
        map = object.toMap();
        assertTrue(map.get("name").isEmpty());
    }

}
```

TEST DA PARTE DI UTENTI GENERICI

Infine, il software è stato sottoposto ad un test da parte di alcuni individui con poca conoscenza dell'informatica e senza nessuna informazione sullo sviluppo. Si è cercato di non aiutare l'utente nell'uso del software, in modo che il risultato sia il più attendibile possibile. È stata data una generale spiegazione sull'obiettivo del software e anche qualche veloce indicazione sul funzionamento della ricerca dei lavoratori, la quale è stata subito capita e ne è stato fatto un corretto utilizzo. Lo scopo di questo test era trovare degli errori che non sono stati individuati nei test precedenti degli sviluppatori, è stata fatta qualche minima modifica per la visualizzazione dei dati. In generale però non è stato trovato alcun particolare errore nello sviluppo del software.

PRINCIPALI PATTERN ADOTTATI

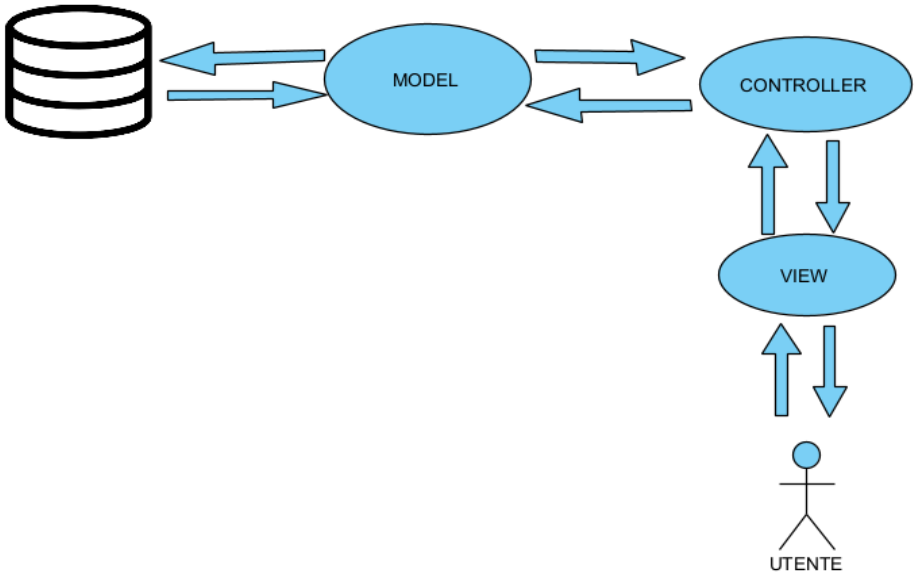
MVC

Nell'architettura del software è stato utilizzato un pattern MVC. Abbiamo scelto questo metodo perché ci è sembrato il più semplice e il più adattabile alla nostra architettura software. In questo modo abbiamo separato le tre componenti per avere un'architettura più chiara e comprensibile.

- **Model**: si trovano i metodi di accesso ai dati e la definizione degli oggetti.
- **View**: fa visualizzare l'interfaccia e i dati all'utente, gestisce l'interazione fra quest'ultimo tutta l'infrastruttura.
- **Controller**: riceve i comandi dell'utente attraverso il View ed esegue delle operazioni che possono modificare i dati nel Model, portano generalmente anche ad un cambiamento nello stato del View.

Il Model interagisce con il database per andare a inserire/modificare/eliminare i dati che sono presenti in quest'ultimo. Il database è dove vengono salvati e archiviati i dati, per la gestione del DB abbiamo utilizzato PgAdmin che è un'interfaccia grafica che consente di amministrare e gestire database di PostgreSQL, abbiamo scelto PgAdmin perché l'abbiamo già utilizzata in corsi precedenti. Mentre per la View è una collezione di oggetti di framework javaFX che vengono caricati tramite i file FXML.

L’utente interagisce con la View vedendo la pagina ed interagendo con essa (inserendo dati, premendo pulsanti ecc…), ogni pulsante è associato ad un metodo o ad una funzione che è presente nel controller, in questo modo quest’ultimo può comunicare con il Model attraverso i metodi che sono presenti nella “repo”, il model comunica con il database con stringe SQL e i driver PostgreSQL.

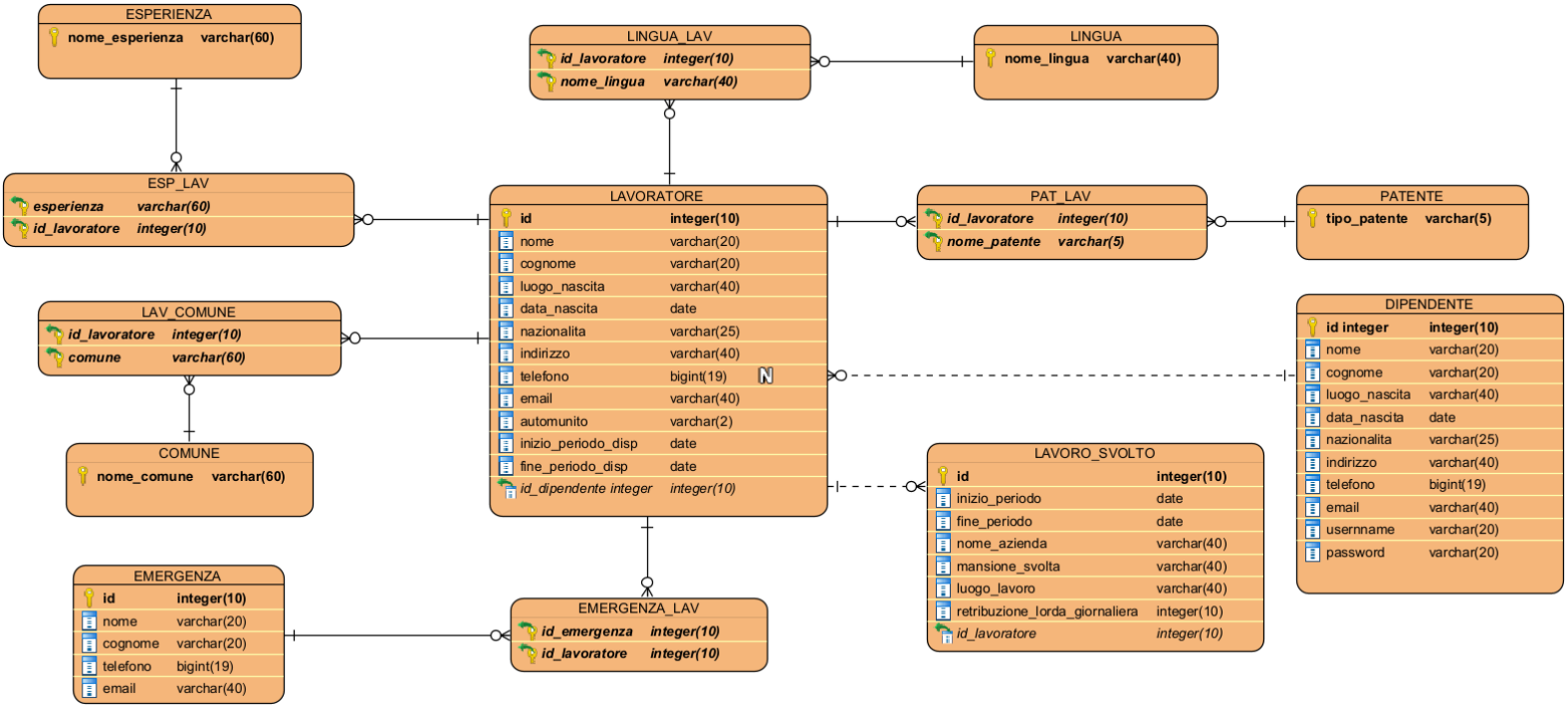


FACTORY PATTERN

Il design pattern utilizzato nella programmazione del software è stato il Factory Pattern. Il Factory Pattern è un pattern creazionale, cioè un pattern che ci permette di creare un'istanza di una classe. È possibile ottenere un oggetto senza conoscere l'esatta classe di provenienza, il Factory Pattern prevede la separazione tra la logica di creazione e l'effettiva logica di utilizzo di una classe. Chi usa la classe Factory non devono interessarsi ad eventuali gerarchie di classi legati agli oggetti di cui hanno bisogno di essere istanziati. Se la classe vuole un determinato oggetto non gli interessa e non vede se fa parte di una gerarchia o dove sia collocato nella gerarchia.
Nel nostro caso Factory Pattern viene usato nel seguente modo.
ButtonColumnFactory è una classe che viene usata per creare una colonna di pulsanti personalizzata, ogni pulsante richiama ad una callback personalizzata passata nel constructor della Factory. Ci sono tre funzioni che la estendono:

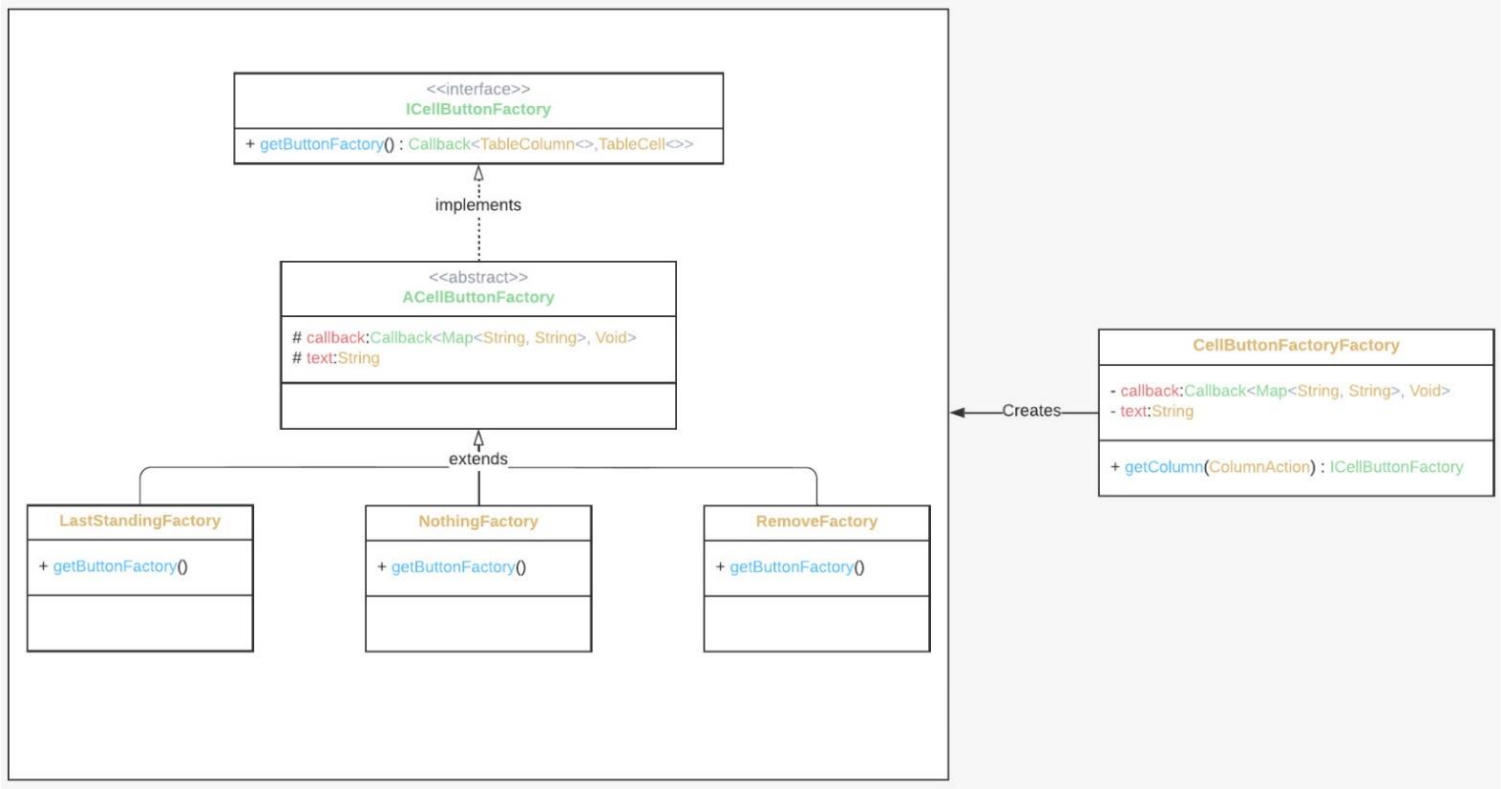
- *getLastStandingColumn().* Crea una colonna di pulsanti, nel caso in cui si provasse ad eliminare l'ultimo elemento verrà annullato e verrà emesso un messaggio di errore (per esempio nelle emergenze se serve mantenere almeno una persona da chiamare in caso di emergenze).
- *getRemoveColumn().*Crea una colonna di pulsanti, nel caso in cui il pulsante viene premuto rimuoverà l'elemento stesso.

- *getNothingColumn().* Crea una colonna di pulsanti (utilizzata per andare a visualizzare per esteso i dati dei lavoratori).



SCHEMA DELLA BASE DI DATI

chiave primaria chiave esterna



LAVORATORE
id, nome, cognome, luogo_nascita, data_nascita, nazionalita, indirizzo, telefono, email, automunito, inizio_periodo_disp, fine_periodo_disp, id_dipendente;

DIPENDENTE
id, nome, cognome, luogo_nascita, data_nascita, nazionalita, indirizzo, telefono, email, username, password;

LAVORO_SVOLTO
id, inizio_periodo, fine_periodo, nome_azienda, mansione_svolta, luogo_lavoro, retribuzione_lorda_giornaliera, id_lavoratore;

EMERGENZA
id, nome, cognome, telefono, email;

PATENTE
tipo_patente;

LINGUA
nome_lingua;

ESPERIENZA
nome_esperienza;

COMUNE
nome_comune;

LINGUA_LAV
nome_lingua, id_lavoratore;

ESP_LAV
esperienza, id_lavoratore;

LAV_COMUNE
comune, id_lavoratore;

PAT_LAV
nome_patente, id_lavoratore;

EMERGENZA_LAV
id_emergenza, id_lavoratore;

ASSOCIAZIONI TRA LE DIVERSE ENTITÀ

- Un lavoratore può aver o non aver fatto una o più esperienze pregresse (0,N), una stessa esperienza può essere fatta da uno o più lavoratori (1,N).
- Un lavoratore può richiedere o non richiedere di lavorare in uno o più comuni (0,N), in un comune possono far richiesta di lavorare nessuno o più lavoratori (0,N).
- Un lavoratore deve avere almeno una o più persone da chiamare in caso di emergenza (1,N), quest'ultima può essere associata ad uno o più lavoratori (1,N).
- Un lavoratore può aver o non aver svolto uno o più lavori (0,N), un determinato lavoro viene svolto solo da un lavoratore (1,1).
- Un lavoratore può avere o non avere una o più tipi di patente (0,N), un tipo di patente può essere o non essere associata ad uno o più lavoratori (0,N).
- Un lavoratore può conoscere o non conoscere una o più lingue (0,N), una lingua può essere o non essere conosciuta da uno o più lavoratori (1,N).
- Un lavoratore viene inserito da un solo dipendente dell'agenzia (1,1), un dipendente può inserire o non inserire uno o più lavoratori (0,N).

NOTE FINALI SUL PROCESSO DI SVILUPPO

Il processo di sviluppo è stato di tipo agile ed incrementale. Quindi le fasi di progettazione, implementazione e validazione non sono state mantenute ben separate. Nonostante questa non netta separazione tra le varie fasi abbiamo cercato prima di progettare e poi di implementare il codice vero e proprio.
Dopo aver modificato e aggiornato il codice è stata sempre fatta una breve attività di test. In parallelo a queste attività, è stata prodotta una documentazione.
Nella documentazione è stato raccolto il materiale UML (diagrammi di classe), gli UML descrittivi (sequence diagram). È stata sviluppata anche la fase di analisi dei requisiti, generando i relativi use-case e i diagrammi di attività. Il database è stato progettato e implementato durante la progettazione, anche se alcune modifiche sono state fatte durante l'implementazione del codice per avere un adattamento migliore del codice con il database.