



# Ingegneria del Software

Corso di Laurea in Ingegneria Informatica

Prof. Nicola Capuano

**Sviluppo di un'applicazione software per la gestione di una rubrica, finalizzata all'organizzazione e alla conservazione dei contatti telefonici e delle e-mail.**

## Progettazione

Davide PERNA RUGGIERO

October 18, 2025



# 1 Introduzione

## 2 Model

### 2.1 Contact.java

Classe che rappresenta l'entità *Contacto* della rubrica. Gestisce le informazioni di ogni contatto (nome, cognome, numeri di telefono, indirizzi e-mail, azienda, note e preferito).

```

1 public final class Contact {
2
3     /*-Constants-*/
4     public static final int MAX_PHONE_COUNT = 3;
5     public static final int MAX_EMAIL_COUNT = 3;
6     public static final int NAME_MAX_LEN = 50;
7
8     /*-Fields-*/
9     private UUID id;
10    private String firstName;
11    private String lastName;
12    private List<String> phoneNumbers;
13    private List<String> emails;
14    private String company;
15    private String notes;
16    private boolean favorite;
17
18    ...
19 }
```

#### 2.1.1 Metodi costruttori

Il costruttore vuoto inizializza i campi vuoti di default e genera automaticamente un UUID. Quest'ultimo è stato inserito per favorire le operazioni che prevedono l'individuazione di un contatto preciso: visione, modifica e eliminazione

```

1 public Contact() {
2     this(UUID.randomUUID(), "", "", new ArrayList<>(), new
        ArrayList<>(), "", "", false);
}
```

Il costruttore parametrico consente di impostare tutti i campi e invoca metodi di validazione per nomi, numeri e email.

```

1 public Contact(
2     UUID id,
```

```
3      String firstName,
4      String lastName,
5      List<String> phoneNumbers,
6      List<String> emails,
7      String company,
8      String notes,
9      boolean favorite) {
10
11      this.id = (id == null) ? UUID.randomUUID() : id;
12      this.phoneNumbers = new ArrayList<>();
13      this.emails = new ArrayList<>();
14      this.company = (company == null) ? "" : company.trim();
15      this.notes = (notes == null) ? "" : notes.trim();
16      this.favorite = favorite;
17
18      setFullName(firstName, lastName);
19      setPhoneNumbers(phoneNumbers);
20      setEmails(emails);
21 }
```

## 2.1.2 Getter

Restituiscono i valori dei campi del contatto.

```
1 public UUID getId() {
2     return id;
3 }
```

```
1 public String getFirstName() {
2     return firstName;
3 }
```

```
1 public String getLastName() {
2     return lastName;
3 }
```

Le liste di telefoni ed email vengono restituite come collezioni non modificabili per garantire l'integrità dei dati.

```
1
2 public List<String> getPhoneNumbers() {
3     return Collections.unmodifiableList(phoneNumbers);
4 }
```

```
1 public List<String> getEmails() {  
2     return Collections.unmodifiableList(emails);  
3 }
```

```
1 public String getCompany() {  
2     return company;  
3 }
```

```
1 public String getNotes() {  
2     return notes;  
3 }
```

```
1 public boolean isFavorite() {  
2     return favorite;  
3 }
```

```
1 public String getFullName() {  
2     String fullName = (firstName + " " + lastName).trim();  
3     return fullName;  
4 }
```

## 2.1.3 Setter

Permettono la modifica controllata dei campi. Eseguono verifiche di validità su nome, numeri di telefono ed email e gestiscono eccezioni per input non validi.

```
1 public void setFullName(String firstName, String lastName) {  
2     String fN = (firstName == null) ? "" : firstName.trim();  
3     String lN = (lastName == null) ? "" : lastName.trim();  
4  
5     if (fN.isEmpty() && lN.isEmpty()) {  
6         throw new IllegalArgumentException("At least a first  
7             or last name must be specified.");  
8     }  
9     this.firstName = fN;  
10    this.lastName = lN;  
11 }
```

```
1 public void setPhoneNumbers(List<String> phones) {  
2     if (this.phoneNumbers == null) {  
3         this.phoneNumbers = new ArrayList<>();  
4     } else {  
5         this.phoneNumbers.clear();  
6     }  
7 }
```

```

7     if (phones == null) return;
8
9     for (String p : phones) {
10        if (p == null || p.trim().isEmpty()) continue;
11        if (!isValidPhone(p))
12            throw new IllegalArgumentException("Invalid_
                phone_number:_" + p);
13        if (phoneNumbers.size() >= MAX_PHONE_COUNT)
14            throw new IllegalStateException("Maximum_number_
                of_phone_numbers_reached.");
15        phoneNumbers.add(p.trim());
16    }
17 }

```

```

1 public void setEmails(List<String> emails) {
2     if (this.emails == null) {
3         this.emails = new ArrayList<>();
4     } else {
5         this.emails.clear();
6     }
7     if (emails == null) return;
8
9     for (String e : emails) {
10        if (e == null || e.trim().isEmpty()) continue;
11        if (!isValidEmail(e))
12            throw new IllegalArgumentException("Invalid_
                email_format:_" + e);
13        if (this.emails.size() >= MAX_EMAIL_COUNT)
14            throw new IllegalStateException("Maximum_number_
                of_emails_reached.");
15        this.emails.add(e.trim());
16    }
17 }

```

```

1 public void setCompany(String company) {
2     this.company = (company == null) ? "" : company.trim();
3 }

```

```

1 public void setNotes(String notes) {
2     this.notes = (notes == null) ? "" : notes.trim();
3 }

```

```

1 public void setFavorite(boolean favorite) {
2     this.favorite = favorite;
3 }

```

### 2.1.4 Restanti metodi

Implementano la logica per aggiungere o rimuovere numeri di telefono ed email. Controllano la validità del formato e rispettano i limiti di massimo tre elementi per tipo.

```

1 public void addPhoneNumber(String number) {
2     if (this.phoneNumbers == null) {
3         this.phoneNumbers = new ArrayList<>();
4     }
5     if (number == null || number.trim().isEmpty()){
6         return;
7     } else if (!isValidPhone(number)){
8         throw new IllegalArgumentException("Invalid_phone_
          number_format.");
9     } else if (phoneNumbers.size() >= MAX_PHONE_COUNT){
10        throw new IllegalStateException("Maximum_number_of_
          phone_numbers_reached.");
11    }
12    phoneNumbers.add(number.trim());
13 }

```

```

1 public void removePhoneNumber(String number) {
2     if (this.phoneNumbers != null) {
3         phoneNumbers.remove(number);
4     }
5 }

```

```

1 public void addEmail(String email) {
2     if (this.emails == null) {
3         this.emails = new ArrayList<>();
4     }
5     if (email == null || email.trim().isEmpty()){
6         return;
7     } else if (!isValidEmail(email)) {
8         throw new IllegalArgumentException("Invalid_email_
          format.");
9     } else if (emails.size() >= MAX_EMAIL_COUNT){
10        throw new IllegalStateException("Maximum_number_of_
          emails_reached.");
11    } else emails.add(email.trim());
12 }

```

```

1 public void removeEmail(String email) {
2     if (this.emails != null) {

```

```

3         emails.remove(email);
4     }
5 }

1 public static boolean isValidPhone(String number) {
2     if (number == null || number.trim().isEmpty()){
3         return false;
4     } else {
5         return number.matches("^\\+?[0-9\\s-]{5,20}$");
6     }
7 }

1 public static boolean isValidEmail(String email) {
2     if (email == null || email.trim().isEmpty()) return
3         false;
4     return email.matches("^([A-Za-z0-9._%+-]+@[A-Za-z0
5         -9.-]+\\.[A-Za-z]{2,})$");
6 }

```

### 2.1.5 @Override

Ridefinisce i metodi `toString()`, `equals()` e `hashCode()` per fornire una rappresentazione leggibile e garantire l'univocità dei contatti tramite l'UUID.

```

1     @Override public String toString() {
2         return "Contact{" + getFullName() + "}";
3     }
4
5     @Override public boolean equals(Object obj) {
6         return obj instanceof Contact && id != null && id.
7             equals(((Contact) obj).id);
8     }
9
10    @Override public int hashCode() {
11        return id == null ? 0 : id.hashCode();
12    }

```

## 2.2 ContactRepository.java

Interfaccia che definisce i comportamenti fondamentali della rubrica. Stabilisce le operazioni per aggiungere, modificare, eliminare, cercare, filtrare e ordinare i contatti, mantenendo un'astrazione dal tipo di persistenza utilizzato.

```

1 public interface ContactRepository {
2     boolean addContact(Contact contact);
3     boolean saveContact(Contact contact);
4     boolean removeContact(Contact contact);
5     Contact getContact(UUID id);
6     ObservableList<Contact> search(String substring);
7     List<Contact> filterFavorites();
8     void sortByFirstNameAscending();
9     void sortByFirstNameDescending();
10    void sortByLastNameAscending();
11    void sortByLastNameDescending();
12    int indexOf(UUID id);
13    ObservableList<Contact> getObservableContacts();
14    boolean contains(Contact contact);
15 }

```

## 2.3 AddressBook.java

Classe che implementa l'interfaccia `ContactRepository`. Rappresenta la rubrica vera e propria e utilizza una `ObservableList<Contact>` per gestire i dati mostrati nella UI. Offre funzionalità CRUD complete e supporta ricerca, filtro e ordinamento alfabetico.

```

1 public class AddressBook implements ContactRepository {
2
3     /** The observable list of contacts that backs the UI
4     table */
5     private final ObservableList<Contact> contacts;
6
7     ...
8 }

```

### 2.3.1 Costruttore

Inizializza la lista osservabile dei contatti.

L'uso di `FXCollections.observableArrayList()` consente di mantenere automaticamente sincronizzata la UI quando i dati cambiano.

```

1 public AddressBook() {
2     this.contacts = FXCollections.observableArrayList();
3 }

```



### 2.3.2 @Overrides

Implementa i metodi dell'interfaccia `ContactRepository`. Le operazioni comprendono: - ricerca di un contatto tramite UUID; - aggiunta, aggiornamento e rimozione di contatti; - ordinamento crescente/decrescente per nome o cognome; - filtro dei contatti preferiti; - accesso all'elenco osservabile dei contatti per l'integrazione con la UI.

```
1 @Override
2 public int indexOf(UUID id) {
3     for (int i = 0; i < contacts.size(); i++)
4         if (contacts.get(i).getId().equals(id)) return i;
5     return -1;
6 }
```

```
1 @Override
2 public ObservableList<Contact> getObservableContacts() {
3     return contacts;
4 }
```

```
1 @Override
2 public boolean addContact(Contact contact) {
3     if (contact == null || contains(contact)) return false;
4     return contacts.add(contact);
5 }
```

```
1 @Override
2 public boolean saveContact(Contact contact) {
3     if (contact == null) return false;
4     int idx = indexOf(contact.getId());
5     if (idx < 0) return false;
6     contacts.set(idx, contact);
7     return true;
8 }
```

```
1 @Override
2 public boolean removeContact(Contact contact) {
3     if (contact == null) return false;
4     return contacts.removeIf(x -> x.getId().equals(contact.getId()));
5 }
```

```
1 @Override
2 public Contact getContact(UUID id) {
```

```

3     for (Contact contact : contacts) if (contact.getId().
        equals(id)) return contact;
4     return null;
5 }

```

```

1 @Override
2 public ObservableList<Contact> search(String substring) {
3     if (substring == null || substring.isEmpty()) {
4         return FXCollections.observableArrayList(contacts);
5     }
6     String lowSub = substring.toLowerCase();
7
8     ObservableList<Contact> wanted = FXCollections.
        observableArrayList(
9         contacts
10            .stream()
11            .filter(contact ->
12                contact.getFirstName().toLowerCase().contains(
13                    lowSub) ||
14                contact.getLastName().toLowerCase().contains(
15                    lowSub)
16            )
17            .collect(Collectors.toList())
18 );
19 return wanted;
20 }

```

```

1 @Override
2 public ObservableList<Contact> filterFavorites() {
3     return FXCollections.observableArrayList(
4         contacts.stream()
5             .filter(Contact::isFavorite)
6             .collect(Collectors.toList())
7     );
8 }

```

```

1 @Override
2 public void sortByFirstNameAscending() {
3     FXCollections.sort(contacts, Comparator
4         .comparing(Contact::getFirstName, String.
5             CASE_INSENSITIVE_ORDER)
6         .thenComparing(Contact::getLastName, String.
7             CASE_INSENSITIVE_ORDER));
8 }

```

```

1 @Override
2 public void sortByLastNameAscending() {
3     FXCollections.sort(contacts, Comparator
4         .comparing(Contact::getLastName, String.
5             CASE_INSENSITIVE_ORDER)
6         .thenComparing(Contact::getFirstName, String.
7             CASE_INSENSITIVE_ORDER));
8 }

```

```

1 @Override
2 public void sortByFirstNameDescending() {
3     FXCollections.sort(contacts, Comparator
4         .comparing(Contact::getFirstName, String.
5             CASE_INSENSITIVE_ORDER.reversed())
6         .thenComparing(Contact::getLastName, String.
7             CASE_INSENSITIVE_ORDER.reversed()));
8 }

```

```

1 @Override
2 public void sortByLastNameDescending() {
3     FXCollections.sort(contacts, Comparator
4         .comparing(Contact::getLastName, String.
5             CASE_INSENSITIVE_ORDER.reversed())
6         .thenComparing(Contact::getFirstName, String.
7             CASE_INSENSITIVE_ORDER.reversed()));
8 }

```

```

1 @Override
2 public boolean contains(Contact contact) {
3     return contact != null && indexOf(contact.getId()) >= 0;
4 }

```

### 2.3.3

Questa sottosezione chiude l'implementazione di **AddressBook**. I metodi sono progettati per garantire integrità dei dati e aggiornamento in tempo reale della UI grazie all'uso di **ObservableList**.

## 2.4 Persistence.java

Interfaccia che definisce le operazioni di persistenza dei dati dei contatti. Permette di astrarre il metodo di memorizzazione (su file o database) garantendo indipendenza tra logica applicativa e gestione fisica dei dati. Include metodi per

aggiungere, aggiornare, eliminare, caricare e verificare l'esistenza delle risorse di memorizzazione, oltre a un metodo di utilità per il parsing delle liste.

```

1 public interface Persistence {
2     boolean appendContact(Contact contact);
3     boolean updateContact(Contact contact);
4     boolean deleteContact(UUID contactId);
5     List<Contact> loadContacts();
6     boolean fileExists();
7     List<String> parseList(String field);
8
9 }

```

## 2.5 DatabasePersistence.java

Classe che implementa l'interfaccia `Persistence` utilizzando un database PostgreSQL. Gestisce la connessione, la creazione della tabella e le operazioni per la persistenza dei contatti. Ogni metodo opera in modo autonomo aprendo e chiudendo la connessione al database tramite `DriverManager`.

```

1 public class DatabasePersistence implements Persistence {
2
3     private static final String URL = "jdbc:postgresql://
4         localhost:5432/AddressBook";
5     private static final String USER = "postgres";
6     private static final String PASSWORD = "*****";
7
8     ...
9 }

```

### 2.5.1 Costruttore

All'avvio dell'applicazione, il costruttore invoca il metodo di inizializzazione per creare la tabella `contacts` nel database se non ancora presente.

```

1 public DatabasePersistence() {
2     initializeDatabase();
3 }

```

### 2.5.2 Metodi:Inizializzazione

Crea la tabella `contacts` con i campi richiesti (UUID, nomi, telefoni, email, ecc.) se non esiste già nel database. Gestisce eccezioni SQL e stampa messaggi diagnostici per confermare l'avvenuta inizializzazione.

```

1 private void initializeDatabase() {
2     String sql =
3         "CREATE TABLE IF NOT EXISTS contacts (" +
4         "id UUID PRIMARY KEY," +
5         "first_name VARCHAR(50)," +
6         "last_name VARCHAR(50)," +
7         "phones TEXT," +
8         "emails TEXT," +
9         "company VARCHAR(100)," +
10        "notes TEXT," +
11        "favorite BOOLEAN" +
12        ")";
13    try (Connection conn = DriverManager.getConnection(URL,
14        USER, PASSWORD);
15        Statement stmt = conn.createStatement()) {
16        stmt.execute(sql);
17        System.out.println("Database initialized and
18        contacts table ready.");
19    } catch (SQLException e) {
20        System.out.println("Error initializing database: " +
21        e.getMessage());
22    }
23 }

```

### 2.5.3 @Override

Implementa le operazioni di persistenza sui contatti memorizzati nel database. Ogni metodo gestisce in modo sicuro la connessione e restituisce un valore booleano per indicare il successo o il fallimento dell'operazione.

```

1 @Override
2 public boolean updateContact(Contact contact) {
3     if (contact == null) return false;
4     String sql =
5         "UPDATE contacts SET " +
6         "first_name=?, last_name=?, phones=?, emails=?
7         , company=?, notes=?, favorite=?" +
8         "WHERE id=?";
9     try (Connection conn = DriverManager.getConnection(URL,
10        USER, PASSWORD);
11        PreparedStatement ps = conn.prepareStatement(sql))
12    {
13    }
14 }

```

```

11         ps.setString(1, contact.getFirstName());
12         ps.setString(2, contact.getLastName());
13         ps.setString(3, String.join(";", contact.
            getPhoneNumbers()));
14         ps.setString(4, String.join(";", contact.getEmails()
            ));
15         ps.setString(5, contact.getCompany());
16         ps.setString(6, contact.getNotes());
17         ps.setBoolean(7, contact.isFavorite());
18         ps.setObject(8, contact.getId());
19         int rows = ps.executeUpdate();
20         return rows > 0;
21     } catch (SQLException e) {
22         System.out.println("Error updating contact: " + e.
            getMessage());
23         return false;
24     }
25 }

```

```

1 @Override
2 public boolean deleteContact(UUID contactId) {
3     if (contactId == null) return false;
4     String sql = "DELETE FROM contacts WHERE id=?";
5     try (Connection conn = DriverManager.getConnection(URL,
        USER, PASSWORD);
6         PreparedStatement ps = conn.prepareStatement(sql))
7     {
8         ps.setObject(1, contactId);
9         int rows = ps.executeUpdate();
10        return rows > 0;
11    } catch (SQLException e) {
12        System.out.println("Error deleting contact: " + e.
            getMessage());
13        return false;
14    }
15 }

```

```

1 @Override
2 public List<Contact> loadContacts() {
3     List<Contact> contacts = new ArrayList<>();
4     String sql = "SELECT * FROM contacts";
5     try (Connection conn = DriverManager.getConnection(URL,
        USER, PASSWORD);
6         Statement stmt = conn.createStatement());

```

```

7      ResultSet rs = stmt.executeQuery(sql)) {
8
9      while (rs.next()) {
10         UUID id = (UUID) rs.getObject("id");
11         String firstName = rs.getString("first_name");
12         String lastName = rs.getString("last_name");
13         List<String> phones = parseList(rs.getString("
            phones"));
14         List<String> emails = parseList(rs.getString("
            emails"));
15         String company = rs.getString("company");
16         String notes = rs.getString("notes");
17         boolean favorite = rs.getBoolean("favorite");
18
19         Contact contact = new Contact(id, firstName,
            lastName, phones, emails, company, notes,
            favorite);
20         contacts.add(contact);
21     }
22     System.out.println("Contacts successfully loaded
        from database.");
23 } catch (SQLException e) {
24     System.out.println("Error loading contacts: " + e.
        getMessage());
25 }
26 return contacts;
27 }

```

```

1 @Override
2 public boolean fileExists() {
3     try (Connection conn = DriverManager.getConnection(URL,
4         USER, PASSWORD)) {
5         return conn.isValid(2);
6     } catch (SQLException e) {
7         return false;
8     }
9 }

```

```

1 @Override
2 public List<String> parseList(String field) {
3     if (field == null || field.isEmpty()) return new
4         ArrayList<>();
5     String[] parts = field.split(";");
6     List<String> list = new ArrayList<>();

```

```

6   for (String p : parts) {
7       if (!p.trim().isEmpty()) list.add(p.trim());
8   }
9   return list;
10 }

```

## 2.6 FilePersistence.java

Classe che implementa l'interfaccia `Persistence` per la gestione dei contatti tramite file CSV locale. Gestisce la creazione, lettura, scrittura e aggiornamento del file, assicurando la conservazione dei dati tra sessioni dell'applicazione.

```

1 public class FilePersistence implements Persistence {
2     public static final String DEFAULT_PATH = "contacts.csv"
3     ;
4     private final Path filePath;
5     ...
6 }

```

### 2.6.1 Costruttore

Crea un'istanza di persistenza su file, inizializzando il percorso del CSV. Se il file non esiste, viene creato e viene scritta l'intestazione delle colonne.

```

1 public FilePersistence() {
2     this(DEFAULT_PATH);
3 }

```

### 2.6.2 Metodi: file path

Gestisce l'inizializzazione del percorso e la creazione del file CSV.

```

1 public FilePersistence(String path) {
2     this.filePath = Paths.get(path);
3     initializeFile();
4 }

```

### 2.6.3 Metodi: inizializzazione

Crea il file CSV se assente e scrive la riga di intestazione con i nomi dei campi dei contatti.



```

1 private void initializeFile() {
2     if (Files.exists(filePath)){
3     } else {
4         try (BufferedWriter writer = Files.newBufferedWriter
5             (filePath)) {
6             writer.write("UUID,FirstName,LastName,Phones,
7                 Emails,Company,Notes,Favorite");
8             writer.newLine();
9             System.out.println("CSV_file_created_at_first_
10                 startup:_" + filePath);
11         } catch (IOException e) {
12             System.out.println("Error_creating_CSV_file:_" +
13                 e.getMessage());
14         }
15     }
16 }

```

#### 2.6.4 @Overrides

Implementa i metodi per aggiungere, aggiornare, cancellare e leggere i contatti dal file CSV. Le operazioni scrivono o riscrivono il contenuto del file in modo sicuro, mantenendo la coerenza con il repository.

```

1 @Override
2 public boolean appendContact(Contact contact) {
3     if (contact == null) return false;
4
5     String pathToFile = filePath.toString();
6     boolean fileExists = Files.exists(filePath);
7
8     try (BufferedWriter bw = new BufferedWriter(new
9         FileWriter(pathToFile, true))) {
10
11         if (!fileExists) {
12             bw.write("UUID,FirstName,LastName,Phones,Emails,
13                 Company,Notes,Favorite");
14             bw.newLine();
15         }
16
17         String phones = String.join(";", contact.
18             getPhoneNumbers());
19         String emails = String.join(";", contact.getEmails()
20             );
21     }
22 }

```

```

17
18     String[] values = {
19         contact.getId().toString(),
20         contact.getFirstName(),
21         contact.getLastName(),
22         phones,
23         emails,
24         contact.getCompany(),
25         contact.getNotes(),
26         String.valueOf(contact.isFavorite())
27     };
28
29     bw.write(String.join(",", values));
30     bw.newLine();
31
32     System.out.println("Contact successfully saved to
33         CSV.");
34     return true;
35 } catch (IOException e) {
36     System.out.println("Error writing CSV file: " + e.
37         getMessage());
38     return false;
39 }

```

```

1 @Override
2 public boolean updateContact(Contact contact) {
3     if (contact == null) return false;
4     List<Contact> all = loadContacts();
5     boolean updated = false;
6
7     for (int i = 0; i < all.size(); i++) {
8         if (all.get(i).getId().equals(contact.getId())) {
9             all.set(i, contact);
10            updated = true;
11            break;
12        }
13    }
14
15    return updated && saveAll(all);
16 }

```

```

1 @Override

```

```

2 public boolean deleteContact(UUID contactId) {
3     if (contactId == null) return false;
4     List<Contact> all = loadContacts();
5     boolean removed = all.removeIf(c -> c.getId().equals(
6         contactId));
7     return removed && saveAll(all);
8 }

1 @Override
2 public List<Contact> loadContacts() {
3     List<Contact> contacts = new ArrayList<>();
4     String pathToFile = filePath.toString();
5
6     if (!Files.exists(filePath)) {
7         System.out.println("CSV file not found. No contacts
8             loaded.");
9         return contacts;
10    }
11
12    try (BufferedReader br = new BufferedReader(new
13        FileReader(pathToFile))) {
14        String line;
15        boolean headerSkipped = false;
16
17        while ((line = br.readLine()) != null) {
18            if (!headerSkipped) {
19                headerSkipped = true;
20                continue;
21            }
22
23            String[] values = line.split(",", -1);
24            if (values.length < 8) continue;
25
26            UUID id = UUID.fromString(values[0]);
27            String firstName = values[1];
28            String lastName = values[2];
29            List<String> phones = parseList(values[3]);
30            List<String> emails = parseList(values[4]);
31            String company = values[5];
32            String notes = values[6];
33            boolean favorite = Boolean.parseBoolean(values
34                [7]);
35
36            Contact contact = new Contact(id, firstName,

```

```

        lastName, phones, emails, company, notes,
        favorite);
34         contacts.add(contact);
35     }
36
37     System.out.println("Contacts successfully loaded
        from CSV.");
38
39     } catch (IOException e) {
40         System.out.println("Error reading CSV file:" + e.
            getMessage());
41     }
42
43     return contacts;
44 }

```

```

1  @Override
2  public boolean fileExists() {
3      return Files.exists(filePath);
4  }

```

```

1  @Override
2  public List<String> parseList(String field) {
3      if (field == null || field.isEmpty()) return new
        ArrayList<>();
4      String[] parts = field.split(";");
5      List<String> list = new ArrayList<>();
6      for (String p : parts) {
7          if (!p.trim().isEmpty()) list.add(p.trim());
8      }
9      return list;
10 }

```

### 2.6.5 Altri metodi di FilePersistence

Metodi di supporto utilizzati internamente per salvare tutti i contatti in blocco e per convertire un oggetto Contact in stringa CSV serializzata.

```

1  private boolean saveAll(List<Contact> contacts) {
2      try (BufferedWriter writer = Files.newBufferedWriter(
3          filePath,
4          StandardOpenOption.CREATE,
5          StandardOpenOption.TRUNCATE_EXISTING)) {
6

```

```

7         writer.write("UUID,FirstName,LastName,Phones,Emails,
8             Company,Notes,Favorite");
9         writer.newLine();
10        for (Contact c : contacts) {
11            writer.write(serializeContact(c));
12            writer.newLine();
13        }
14        return true;
15    } catch (IOException e) {
16        System.out.println("Error saving contacts to CSV: "
17            + e.getMessage());
18        return false;
19    }
20 }

```

```

1 private String serializeContact(Contact c) {
2     String phones = String.join(";", c.getPhoneNumbers());
3     String emails = String.join(";", c.getEmails());
4     String[] values = {
5         c.getId().toString(),
6         c.getFirstName(),
7         c.getLastName(),
8         phones,
9         emails,
10        c.getCompany(),
11        c.getNotes(),
12        String.valueOf(c.isFavorite())
13    };
14    return String.join(",", values);
15 }

```

### 3 Controller

### 4 Controller

#### 4.1 MainViewController

Controller principale dell'interfaccia utente. Gestisce la tabella dei contatti e tutte le azioni principali: aggiunta, ricerca, filtro, ordinamento e apertura della vista di dettaglio. Si interfaccia con il modello tramite le classi `ContactRepository` e `Persistence`.

```

1 public class MainViewController implements Initializable {
2
3     /*UI Components*/
4     @FXML private TableView<Contact> contacts;
5     @FXML private TableColumn<Contact, String>
6         contactFirstName;
7     @FXML private TableColumn<Contact, String>
8         contactLastName;
9     @FXML private TextField search;
10    @FXML private Button searchBtn;
11    @FXML private Button addBtn;
12    @FXML private Button sortByFirstNameBtn;
13    @FXML private Button sortByLastNameBtn;
14    @FXML private Button filterByFavouriteBtn;
15    @FXML private Button clearBtn;
16
17    /*-Model-*/
18    private ContactRepository repository;
19
20    private Persistence persistence;
21
22    /* FilteredList allows dynamic the observable contact
23       list */
24    private FilteredList<Contact> filteredContacts;
25
26    /* UI state flags */
27    private boolean showingFavorites = false;
28    private boolean sortedByFirstName = false;
29    private boolean sortedByLastName = false;
30
31    ...
32 }

```

#### 4.1.1 Metodi: initialize()

Metodo eseguito automaticamente all'avvio della vista. Configura le colonne della tabella e imposta la policy di ridimensionamento.

```

1 @Override
2 public void initialize(URL location, ResourceBundle
3     resources) {
4     contactFirstName.setCellValueFactory(new
5         PropertyValueFactory<>("firstName"));
6 }

```

```

4     contactLastName.setCellValueFactory(new
        PropertyValueFactory<>("lastName"));
5     contacts.setColumnResizePolicy(TableView.
        CONSTRAINED_RESIZE_POLICY);
6 }

```

#### 4.1.2 Metodi: init()

Inizializza il controller collegando repository e livello di persistenza. Popola la tabella con la lista osservabile dei contatti.

```

1 public void init(ContactRepository repository, Persistence
    persistence) {
2     this.repository = repository;
3     this.persistence = persistence;
4     filteredContacts = new FilteredList<>(repository.
        getObservableContacts(), c -> true);
5     contacts.setItems(filteredContacts);
6 }

```

#### 4.1.3 @FXML handle

Gestisce le azioni dell'utente associate ai pulsanti e agli eventi UI. Comprende operazioni di aggiunta, ricerca, filtro, ordinamento e apertura della vista dettaglio.

```

1 @FXML
2 public void handleAddContact() {
3     try {
4         FXXMLLoader loader = new FXXMLLoader(getClass().
            getResource("/view/ContactEditView.fxml"));
5         Parent root = loader.load();
6         ContactEditController editController = loader.
            getController();
7         editController.init(repository, persistence, null);
8         Stage stage = App.getPrimaryStage();
9         stage.setScene(new Scene(root, 1000, 700));
10        stage.setTitle("Add Contact");
11    } catch (IOException exception) {
12        exception.printStackTrace();
13    }
14 }

```

```

1 @FXML
2 public void handleSortByFirstName() {

```

```

3     if (sortedByFirstName) repository.
        sortByFirstNameDescending();
4     else repository.sortByFirstNameAscending();
5     sortedByFirstName = !sortedByFirstName;
6     sortedByLastName = false;
7     contacts.refresh();
8 }

```

```

1 @FXML
2 public void handleSortByLastName() {
3     if (sortedByLastName) repository.
        sortByLastNameDescending();
4     else repository.sortByLastNameAscending();
5     sortedByLastName = !sortedByLastName;
6     sortedByFirstName = false;
7     contacts.refresh();
8 }

```

```

1 @FXML
2 public void handleFilterByFavorite() {
3     if (showingFavorites) filteredContacts.setPredicate(c ->
        true);
4     else filteredContacts.setPredicate(Contact::isFavorite);
5     showingFavorites = !showingFavorites;
6     contacts.refresh();
7 }

```

```

1 @FXML
2 public void handleClearBtn() {
3     if (search != null) search.clear();
4     showingFavorites = false;
5     filteredContacts.setPredicate(contact -> true);
6     contacts.refresh();
7 }

```

```

1 @FXML
2 public void handleSearch() {
3     String q = (search != null) ? search.getText().trim().
        toLowerCase() : "";
4     filteredContacts.setPredicate(contact ->
5         q.isEmpty() ||
6         contact.getFirstName().toLowerCase().contains(q)
7         ||
            contact.getLastName().toLowerCase().contains(q))
            ;

```



```

8     contacts.refresh();
9 }

1 @FXML
2 public void handleOpenContact(MouseEvent event) {
3     if (event.getClickCount() == 2 && contacts.
4         getSelectionModel().getSelectedItem() != null) {
5         Contact selected = contacts.getSelectionModel().
6             getSelectedItem();
7         showDetailView(selected);
8     }
9 }

```

#### 4.1.4 Metodi: showDetailView()

Apri la vista dei dettagli di un contatto. Carica il file FXML ContactDetailView.fxml, inizializza il controller associato e visualizza la scena corrispondente.

```

1 private void showDetailView(Contact contact) {
2     try {
3         FXMLLoader loader = new FXMLLoader(getClass().
4             getResource("/view/ContactDetailView.fxml"));
5         Parent root = loader.load();
6         ContactDetailController detailController = loader.
7             getController();
8         detailController.init(repository, persistence,
9             contact);
10        Stage stage = App.getPrimaryStage();
11        stage.setScene(new Scene(root, 1000, 700));
12        stage.setTitle("Contact Details");
13    } catch (IOException exception) {
14        exception.printStackTrace();
15    }
16 }

```

## 4.2 ContactDetailController

Controller che gestisce la vista di dettaglio di un contatto. Visualizza tutte le informazioni del contatto selezionato e consente la modifica o l'eliminazione. Permette inoltre di tornare alla vista principale.

```

1 public class ContactDetailController {

```

```

2
3      /*-Data Models-*/
4      private ContactRepository repository;
5      private Persistence persistence;
6      private Contact contact;
7
8      /*-UI Components-*/
9      @FXML private Label firstNameField;
10     @FXML private Label lastNameField;
11     @FXML private Label companyLabel;
12     @FXML private Label notesLabel;
13     @FXML private Label favoriteLabel;
14     @FXML private Label phone1, phone2, phone3;
15     @FXML private Label email1, email2, email3;
16     @FXML private Button editContactBtn;
17     @FXML private Button deleteContactBtn;
18     @FXML private Button backBtn;
19
20     ...
21 }

```

#### 4.2.1 Metodo: init()

Inizializza il controller assegnando repository, persistenza e contatto selezionato. Chiama `updateLabels()` per popolare la vista con i dati del contatto.

```

1 public void init(ContactRepository repository, Persistence
   persistence, Contact contact) {
2     this.repository = repository;
3     this.persistence = persistence;
4     this.contact = contact;
5
6     updateLabels();
7 }

```

#### 4.2.2 @FXML handle

Metodi associati agli eventi dei pulsanti dell'interfaccia. Gestiscono modifica, eliminazione e ritorno alla schermata principale.

```

1 @FXML
2 public void handleEditContact() {
3     if (contact == null) return;
4

```

```

5      try {
6          FXMLLoader loader = new FXMLLoader(getClass().
              getResource("/view/ContactEditView.fxml"));
7          Parent root = loader.load();
8
9          ContactEditController editController = loader.
              getController();
10         editController.init(repository, persistence, contact
              );
11
12         Stage stage = App.getPrimaryStage();
13         stage.setScene(new Scene(root, 1000, 700));
14         stage.setResizable(false);
15         stage.setTitle("Edit Contact");
16     } catch (IOException e) {
17         e.printStackTrace();
18     }
19 }

```

```

1 @FXML
2 public void handleDeleteContact() {
3     if (contact == null) return;
4
5     Alert confirm = new Alert(AlertType.CONFIRMATION);
6     confirm.setTitle("Delete Contact");
7     confirm.setHeaderText("Are you sure you want to delete
            this contact?");
8     confirm.setContentText(contact.getFullName());
9
10    confirm.showAndWait().ifPresent(response -> {
11        if (response.getButtonData() == ButtonBar.ButtonData
            .OK_DONE) {
12            repository.removeContact(contact);
13            persistence.deleteContact(contact.getId());
14            showMainView();
15        }
16    });
17 }

```

```

1 @FXML
2 public void handleBack() {
3     showMainView();
4 }

```

### 4.2.3 Metodi: showMainView()

Ritorna alla vista principale caricando MainView.fxml e reinizializzando il controller.

```

1 private void showMainView() {
2     try {
3         FXMLLoader loader = new FXMLLoader(getClass().
4             getResource("/view/MainView.fxml"));
5         Parent root = loader.load();
6
7         MainViewController mainController = loader.
8             getController();
9         mainController.init(repository, persistence);
10
11         Stage stage = App.getPrimaryStage();
12         stage.setScene(new Scene(root, 1000, 700));
13         stage.setResizable(false);
14         stage.setTitle("Address_Book");
15     } catch (IOException exception) {
16         exception.printStackTrace();
17     }
18 }

```

### 4.2.4 Metodi: updateLabels()

Aggiorna le etichette della vista con i dati del contatto attualmente selezionato.

```

1 private void updateLabels() {
2     if (contact == null) return;
3
4     firstNameField.setText(contact.getFirstName());
5     lastNameField.setText(contact.getLastName());
6     companyLabel.setText(contact.getCompany());
7     notesLabel.setText(contact.getNotes());
8     favoriteLabel.setText(contact.isFavorite() ? " " : " ");
9
10    List<String> phones = contact.getPhoneNumbers();
11    phone1.setText(phones.size() > 0 ? phones.get(0) : "");
12    phone2.setText(phones.size() > 1 ? phones.get(1) : "");
13    phone3.setText(phones.size() > 2 ? phones.get(2) : "");
14
15    List<String> emails = contact.getEmails();

```

```

16     email1.setText(emails.size() > 0 ? emails.get(0) : "");
17     email2.setText(emails.size() > 1 ? emails.get(1) : "");
18     email3.setText(emails.size() > 2 ? emails.get(2) : "");
19 }

```

### 4.3 ContactEditController

Controller che gestisce la vista di creazione e modifica dei contatti. Permette all'utente di inserire, aggiornare e salvare le informazioni dei contatti. Si interfaccia con il modello tramite `ContactRepository` e `Persistence`, assicurando che i dati vengano salvati sia in memoria che nel sistema di persistenza.

```

1 public class ContactEditController {
2
3     /*-Data Models-*/
4     private ContactRepository repository;
5     private Persistence persistence;
6     private Contact editing;
7     // The contact currently being edited (null if new)
8     /*-UI Components-*/
9     @FXML private TextField firstNameField;
10    @FXML private TextField lastNameField;
11    @FXML private TextField companyField;
12    @FXML private TextArea notesArea;
13    @FXML private CheckBox favoriteCheck;
14    @FXML private TextField phone1, phone2, phone3;
15    @FXML private TextField email1, email2, email3;
16    @FXML private Button saveBtn, backBtn;
17
18    ...
19 }

```

#### 4.3.1 Metodi - Form

Metodi che gestiscono la compilazione e la lettura del modulo. `populateForm()` riempie i campi con i dati del contatto selezionato per l'editing, mentre `buildContactFromInput()` costruisce un nuovo oggetto `Contact` a partire dai dati inseriti dall'utente. Il metodo `notBlank()` è una funzione di supporto che verifica la presenza di testo nei campi.

```

1 private void populateForm(Contact contact) {
2     firstNameField.setText(contact.getFirstName());
3     lastNameField.setText(contact.getLastName());

```

```

4     companyField.setText(contact.getCompany());
5     notesArea.setText(contact.getNotes());
6     favoriteCheck.setSelected(contact.isFavorite());
7
8     List<String> phones = contact.getPhoneNumbers();
9     phone1.setText(phones.size() > 0 ? phones.get(0) : "");
10    phone2.setText(phones.size() > 1 ? phones.get(1) : "");
11    phone3.setText(phones.size() > 2 ? phones.get(2) : "");
12
13    List<String> emails = contact.getEmails();
14    email1.setText(emails.size() > 0 ? emails.get(0) : "");
15    email2.setText(emails.size() > 1 ? emails.get(1) : "");
16    email3.setText(emails.size() > 2 ? emails.get(2) : "");
17 }

1 private Contact buildContactFromInput() {
2     List<String> phones = new ArrayList<>();
3     if (notBlank(phone1)) phones.add(phone1.getText().trim());
4     if (notBlank(phone2)) phones.add(phone2.getText().trim());
5     if (notBlank(phone3)) phones.add(phone3.getText().trim());
6
7     List<String> emails = new ArrayList<>();
8     if (notBlank(email1)) emails.add(email1.getText().trim());
9     if (notBlank(email2)) emails.add(email2.getText().trim());
10    if (notBlank(email3)) emails.add(email3.getText().trim());
11
12    UUID id = (editing == null) ? null : editing.getId();
13
14    return new Contact(
15        id,
16        firstNameField.getText(),
17        lastNameField.getText(),
18        phones,
19        emails,
20        companyField.getText(),
21        notesArea.getText(),
22        favoriteCheck.isSelected()
23    );

```

```
24 }
```

```
1 private boolean notBlank(TextField field) {
2     return field != null && field.getText() != null && !
        field.getText().trim().isEmpty();
3 }
```

#### 4.3.2 @FXML handle

Metodi collegati ai pulsanti della vista. `handleSaveContact()` gestisce sia la creazione di un nuovo contatto sia la modifica di uno esistente, aggiornando il repository e la persistenza. `handleBack()` consente di tornare alla vista principale senza salvare.

```
1 @FXML
2 public void handleSaveContact() {
3     try {
4         Contact contact = buildContactFromInput();
5
6         if (editing == null) {
7             // New contact
8             if (repository.addContact(contact)) {
9                 persistence.appendContact(contact);
10            }
11        } else {
12            // Edit existing contact
13            if (repository.saveContact(contact)) {
14                persistence.updateContact(contact);
15            }
16        }
17
18        showMainView();
19
20    } catch (Exception e) {
21        showError("Error while saving contact: " + e.
            getMessage());
22    }
23 }
```

```
1 @FXML
2 public void handleBack() {
3     showMainView();
4 }
```

### 4.3.3 Metodi: showMainView()

Ricarica la vista principale dopo l'operazione di salvataggio o annullamento. Inizializza di nuovo il controller principale e ripristina la tabella dei contatti.

```

1 private void showMainView() {
2     try {
3         FXMLLoader loader = new FXMLLoader(getClass().
4             getResource("/view/MainView.fxml"));
5         Parent root = loader.load();
6
7         MainViewController mainController = loader.
8             getController();
9         mainController.init(repository, persistence);
10
11         Stage stage = App.getPrimaryStage();
12         stage.setScene(new Scene(root, 1000, 700));
13         stage.setResizable(false);
14         stage.setTitle("Address_Book");
15     } catch (IOException exception) {
16         exception.printStackTrace();
17     }
18 }

```

### 4.3.4 Metodi: showError()

Mostra un messaggio di errore tramite finestra di dialogo in caso di problemi durante il salvataggio o la validazione dei dati.

```

1 private void showError(String message) {
2     Alert alert = new Alert(AlertType.ERROR);
3     alert.setTitle("Validation_Error");
4     alert.setHeaderText(null);
5     alert.setContentText(message);
6     alert.showAndWait();
7 }

```

## 5 View

Questa sezione contiene la classe principale dell'applicazione, responsabile dell'avvio e della gestione della finestra principale. Qui viene inizializzata la persistenza, caricati i contatti e avviata la scena JavaFX principale.



## 5.1 App.java

Classe principale che estende `Application`. Gestisce l'avvio dell'interfaccia grafica e coordina il caricamento dei dati dal sistema di persistenza selezionato (file CSV o database PostgreSQL). All'interno del metodo `start()`, vengono creati e collegati il repository e la persistenza, e viene caricata la vista principale `MainView.fxml`.

```

1 public class App extends Application {
2
3     private static Stage primaryStage;
4     private static ContactRepository repository;
5     private static Persistence persistence;
6
7     ...
8 }

```

### 5.1.1 @Override

Metodo `start()` eseguito automaticamente da JavaFX all'avvio dell'applicazione. Inizializza lo stage principale, imposta il tipo di persistenza (file o database) e carica la vista principale. Il codice è predisposto per consentire il passaggio rapido da persistenza su file a quella su database.

```

1 @Override
2     public void start(Stage stage) throws Exception {
3         primaryStage = stage;
4
5         repository = new AddressBook();
6         // persistence = new FilePersistence();
7         /*
8          * Uncomment line 42 and comment line 46 to switch
9          * persistence
10          * from PostgreSQL database to CSV file.
11          */
12         persistence = new DatabasePersistence();
13
14         // Load contacts from either the database or the CSV
15         // file
16         persistence.loadContacts().forEach(repository::
17             addContact);
18
19         FXMLLoader loader = new FXMLLoader(getClass().
20             getResource("/view/MainView.fxml"));
21         Parent root = loader.load();

```

```
18
19     MainViewController controller = loader.getController
20         ();
21     controller.init(repository, persistence);
22
23     Scene scene = new Scene(root, 1000, 700);
24     stage.setScene(scene);
25     stage.setTitle("Address_Book");
26     stage.setResizable(false);
27     stage.show();
28 }
```

## 5.1.2 Getter

Metodo statico di utilità che restituisce lo **Stage** principale dell'applicazione, permettendo ai controller di accedere alla finestra principale per aggiornare le viste.

```
1 public static Stage getPrimaryStage() {
2     return primaryStage;
3 }
```

## 6 Matrice di tracciabilità

ID	Requisito	Design	Codice	Test	Requisiti collegati
RF01	Gestione dei contatti	Progettato	Implementato	In sviluppo	
RF02	Persistenza dei dati	Progettato	Implementato	In sviluppo	
RF03	Ricerca	Progettato	Implementato	In sviluppo	RF01
RF04	Ordinamento	Progettato	Implementato	In sviluppo	RF05
RF05	Interfaccia	Progettato	Implementato	In sviluppo	RF01, RF03, RF04