Applicativo Java per la gestione dei ricoveri ospedalieri
Elaborato ingegneria del software

Marika Rucco

A.A 2023/2024

UNIVERSITA' DEGLI STUDI DI FIRENZE
Facoltà di ingegneria
Corso di Laurea in Ingegneria Informatica

Indice

1 Descrizione del progetto **3**

2 Requisiti funzionali **3**
2.1 Use case diagram . . . 3
2.2 Use case templates . . . 4
2.3 Mockups . . . 6

3 Progettazione **8**
3.1 Scelte implementative . . . 8
3.2 Class diagram . . . 8
3.3 Classi ed interfacce . . . 9
3.3.1 Bed . . . 9
3.3.2 DailyCheck . . . 10
3.3.3 Doctor . . . 11
3.3.4 HospitalizationRecord . . . 12
3.3.5 Nurse . . . 13
3.3.6 Patient . . . 13
3.3.7 NursePage . . . 14
3.3.8 ResidentPage . . . 15
3.3.9 SpecialistPage . . . 16
3.3.10 PatientDaoImpl . . . 16
3.3.11 DailyCheckDaoImpl . . . 19
3.3.12 HospitalizationRecordDaoImpl . . . 21
3.4 Design patterns . . . 23
3.4.1 Observer . . . 24
3.4.2 DAO . . . 24
3.5 Disposizione delle classi nei package . . . 25

4 Unit Test **25**
4.0.1 NursePageTest . . . 25
4.0.2 ResidentPageTest . . . 26
4.0.3 SpecialistPageTest . . . 27

1 Descrizione del progetto

L'intento della realizzazione di questo applicativo è quello di gestire i ricoveri in ambito ospedaliero. Quando un paziente viene ricoverato, un infermiere raccoglie i suoi dati personali per poterli salvare nel sistema e gli assegna un letto disponibile. Successivamente, uno specializzando assegna una cartella di ricovero al paziente. Ogni mattina, viene effettuato il giro visite, durante il quale lo specializzando visita il paziente e compila una diaria, ovvero un documento in cui sono indicati: il paziente, il medico, i parametri del paziente ed eventuali nuovi sintomi. Una volta compilata la diaria, questa viene visionata da un medico strutturato che può approvarla o meno. Solo nel momento in cui lo strutturato approva la diaria, questa potrà essere aggiunta alla cartella di ricovero del paziente. Quando il paziente può essere dimesso, lo strutturato incarica un inferiemere di liberare il posto letto occupato.

2 Requisiti funzionali

2.1 Use case diagram

In questa sezione si riporta lo Use Case Diagram relativo al modello di dominio che si desidera rappresentare nell'ambito di questo progetto. Nel diagramma sottostante si individuano tre attori che corrispondono alle tre figure professionali che solitamente si occupano della gestione di un paziente ricoverato in uno specifico reparto: l'infermiere, lo specializzando e lo strutturato.

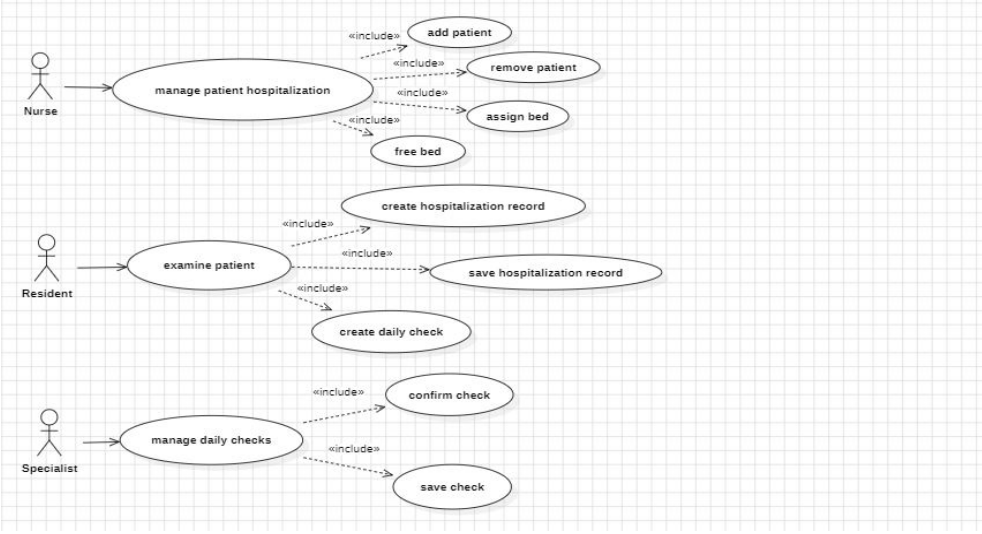


Figura 1: Use Case Diagram

2.2 Use case templates

Si riportano di seguito alcuni Use Case Templates relativi ai casi d'uso di maggior interesse.

UC1	Registra paziente
Livello	User Goal
Descrizione	L'infermiere registra il paziente ricoverato nel database.
Attori	Infermiere
Pre-condizioni	L'infermiere deve essere a conoscenza di alcune informazioni personali del paziente al fine di poterlo registrare correttamente.
Post-condizioni	Il paziente è correttamente registrato nel database.
Normale svolgimento	1. L'infermiere ottiene i dati del paziente, ad esempio, nome, cognome ed età. 2. L'infermiere apre la sua schermata e clicca su "aggiungi un nuovo paziente". 3. Si aprirà una finestra modale in cui l'infermiere aggiungerà i dati ottenuti precedentemente dal paziente. 4. L'infermiere clicca su "aggiungi".

Figura 2: Use Case Template relativo al caso d'uso di registrazione del paziente

UC2	Assegna letto
Livello	User Goal
Descrizione	L'infermiere assegna un letto disponibile al paziente ricoverato.
Attori	Infermiere
Pre-condizioni	L'infermiere deve sapere quali letti sono ancora liberi.
Post-condizioni	Il letto rimane occupato dal paziente per tutta la durata del ricovero.
Normale svolgimento	1. L'infermiere apre la sua schermata in cui è presente una rappresentazione di tutti i letti e ne seleziona uno libero. 2. L'infermiere assegna il letto libero al nuovo paziente ricoverato.

Figura 3: Use Case Template relativo al caso d'uso di assegnazione del letto al paziente

UC3	Crea cartella di ricovero
Livello	User Goal
Descrizione	Lo specializzando crea una nuova cartella di ricovero per il paziente ricoverato.
Attori	Specializzando
Pre-condizioni	Lo specializzando deve essere a conoscenza di alcune informazioni personali del paziente per poter creare una cartella di ricovero.
Post-condizioni	Viene creata una cartella di ricovero in cui sono raccolte tutte le diarie relative al paziente.
Normale svolgimento	1. Lo specializzando acquisisce alcuni dati personali del paziente ricoverato. 2. Lo specializzando apre la sua schermata e clicca su "crea nuova cartella di ricovero". 3. Lo specializzando inserisce i dati relativi al paziente. 4. L'infermiere clicca su "salva".

Figura 4: Use Case Template relativo al caso d'uso di creazione di una cartella di ricovero

UC4	Crea diaria
Livello	User Goal
Descrizione	Lo specializzando crea una diaria in cui riporta i dati del paziente, la data, i parametri ed eventuali nuovi sintomi.
Attori	Specializzando
Pre-condizioni	Lo specializzando deve aver precedentemente visitato il paziente.
Post-condizioni	Viene creata una diaria che verrà successivamente controllata da un medico strutturato.
Normale svolgimento	1. Lo specializzando visita il paziente. 2. Lo specializzando apre la sua schermata e clicca su "crea nuova diaria". 3. Lo specializzando inserisce il nome del paziente, i suoi parametri ed eventuali nuovi sintomi. 4. Lo specializzando attende l'approvazione da parte del medico strutturato della diaria appena creata.

Figura 5: Use Case Template relativo al caso d'uso di creazione di una diaria

UC5	Conferma diaria
Livello	User Goal
Descrizione	Il medico strutturato riceve una notifica della creazione di una nuova diaria e ne controlla il contenuto.
Attori	Strutturato
Pre-condizioni	Prima di poter confermare una diaria è necessario che questa sia stata redatta in maniera completa dal medico specializzando.
Post-condizioni	La diaria viene aggiunta alla cartella di ricovero del paziente.
Normale svolgimento	1. Lo strutturato viene notificato della creazione di una nuova diaria. 2. Lo strutturato controlla i vari campi. 3. Lo strutturato approva la diaria e la aggiunge alla cartella di ricovero del paziente.

Figura 6: Use Case Template relativo al caso d'uso di conferma di una diaria

2.3 Mockups

In questa sezione vengono mostrate delle possibili realizzazioni di interfacce grafiche relative ad alcuni casi d'uso presentati nella sezione precedente.
Il primo mockup individua una possibile schermata principale di un infermiere, nella quale sono presenti i letti liberi e quelli occupati e dalla quale è possibile svolgere alcune azioni come aggiungere e rimuovere un paziente.

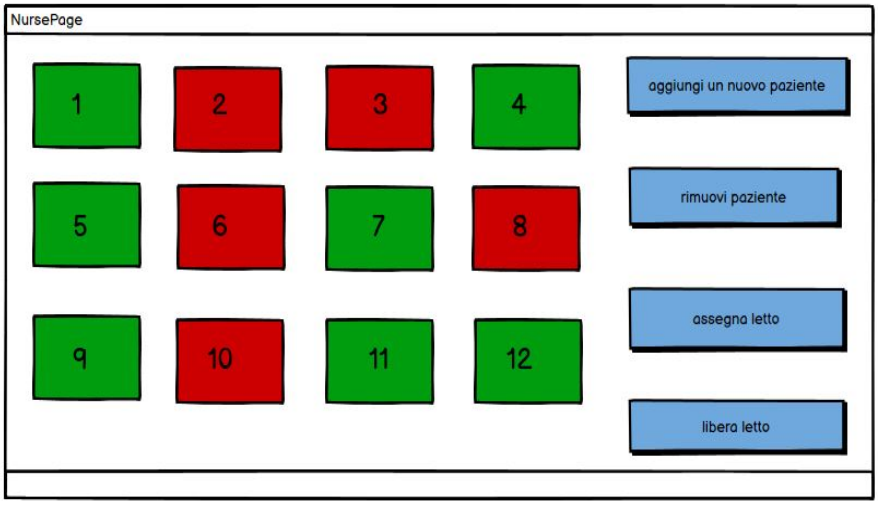


Figura 7: Mockup raffigurante la schermata principale di un infermiere

Il secondo mockup rappresenta una possibile interfaccia grafica per l'aggiunta di un nuovo paziente nel database. Questa finestra modale si apre cliccando su "ag-

giungi un nuovo paziente" dalla schermata principale della pagina dell'infermiere (vedi Figura 7).

Figura 8: Mockup raffigurante la schermata per aggiungere un nuovo paziente

Il seguente mockup è relativo alla creazione di una nuova diaria. Questa possibile interfaccia è realizzata per essere utilizzata dai medici specializzandi che ogni giorno visitano i pazienti e successivamente registrano i dati raccolti.

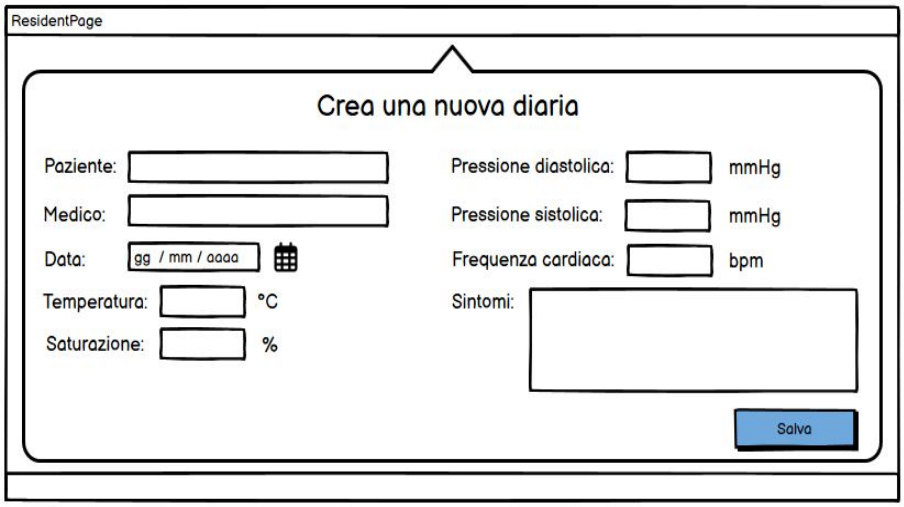


Figura 9: Mockup raffigurante la schermata per creare una nuova diaria

Quest'ultimo mockup rappresenta una possibile realizzazione della schermata del medico strutturato. In particolare si mostra il caso relativo alla conferma di una diaria precedentemente redatta da un medico specializzando.

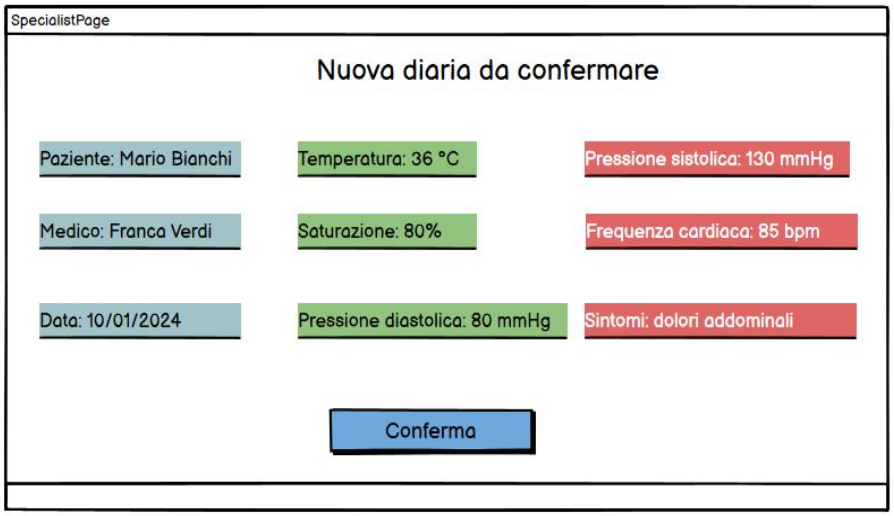


Figura 10: Mockup raffigurante la schermata per confermare una diaria

3 Progettazione

3.1 Scelte implementative

Per l'implementazione del progetto nel linguaggio Java è stato utilizzato l'IDE Intellij IDEA. Lo Use Case Diagram e il Class Diagram sono stati realizzati con il software StarUML, mentre i mockups sono stati creati attraverso l'utilizzo del software Balsamiq Wireframes. Per la stesura di questa relazione è stato impiegato l'editor online Overleaf. Per quanto riguarda la persistenza dei dati è stato utilizzato un database relazionale PostgreSQL; la connessione al database è stata stabilita grazie all' impiego del driver JDBC postgresql-42.7.1 .

3.2 Class diagram

Si presenta in questa sezione il Class Diagram inerente alle classi che sono state implementate per la realizzazione dell'applicativo. Nelle sezioni successive ciascuna classe verrà analizzata in maniera dettagliata e approfondita in relazione alla sua funzione.

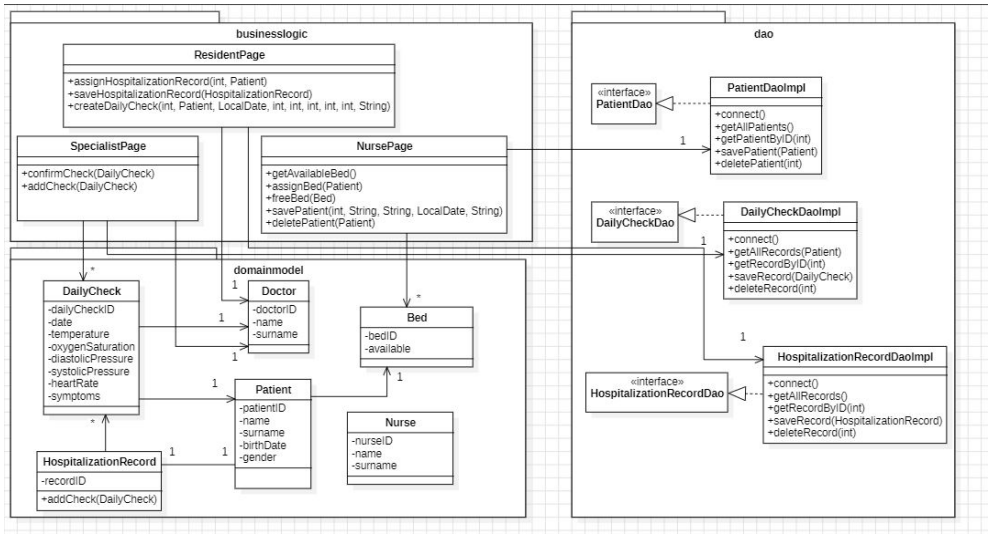


Figura 11: Class Diagram

3.3 Classi ed interfacce

In questa parte, come anticipato precedentemente, verranno mostrate più nel dettaglio le classi che sono state implementate per la realizzazione dell'applicativo. La descrizione partirà dalle classi utili alla modellazione del dominio, proseguendo con quelle relative alla business logic ed infine verranno analizzate le classi per le interazioni con il database.

3.3.1 Bed

Questa classe rappresenta un letto del reparto ed è caratterizzata da un intero bedID, utile per poterlo associare ad un paziente nel database, e da un booleano che indica se è libero o occupato.

```java
public class Bed {
    3 usages
    private int bedID;
    2 usages
    private boolean available = true;

    4 usages
    public Bed(int bedID) { this.bedID = bedID; }

    no usages
    public void setBedID(int bedID) { this.bedID = bedID; }

    2 usages
    public int getBedID() { return bedID; }

    2 usages
    public void setAvailable(boolean available) { this.available = available; }

    5 usages
    public boolean isAvailable() { return available; }

}
```

Figura 12: Porzione di codice relativa alla classe Bed

3.3.2 DailyCheck

La seguente classe è stata realizzata con lo scopo di implementare una diaria, ovvero un documento che viene compilato ogni giorno e che contiente informazioni relative allo stato di salute del paziente. In questo caso la classe presenta un riferimento a un oggetto di tipo Patient e un riferimento a un oggetto di tipo Doctor, e presenta, inoltre, una serie di interi che identificano i principali parametri che vengono presi al paziente durante una visita come, ad esempio: temperatura, saturazione, pressione sistolica e pressione diastolica, frequenza cardiaca e nuovi sintomi.

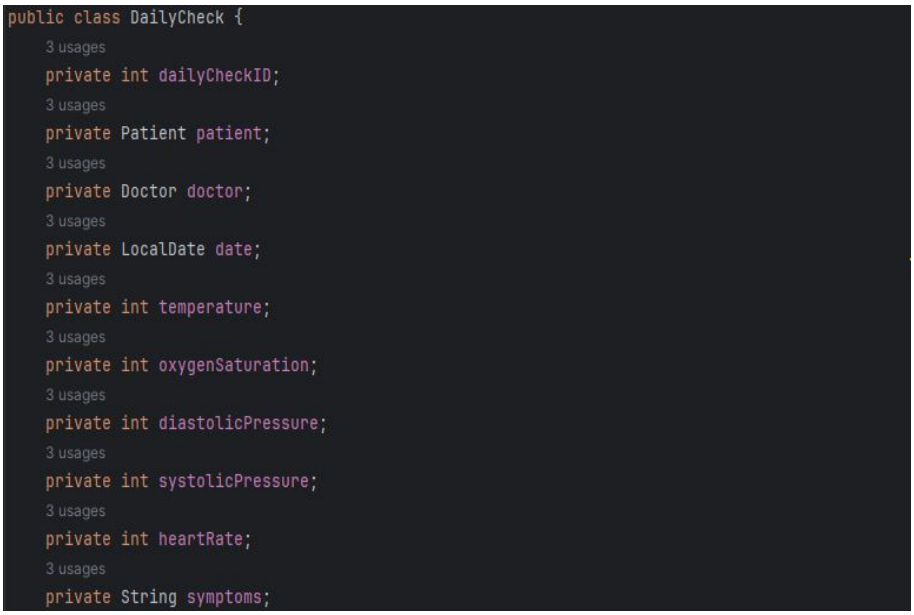


Figura 13: Porzione di codice relativa alla classe DailyCheck

3.3.3 Doctor

Questa classe rappresenta un dottore e viene utilizzata nella business logic sia per rappresentare uno specializzando, sia per indicare un medico strutturato. Essa è caratterizzata da un intero doctorID, utile per tenere traccia del medico che ha scritto la diaria inserita nel database e due riferimenti ad oggetti di tipo String per il nome e il cognome.

```
    private int doctorID;
    3 usages
    private String name;
    3 usages
    private String surname;
    2 usages
    public Doctor( int doctorID) { this.doctorID = doctorID; }
    2 usages
    public Doctor( int doctorID, String name, String surname){
        this.doctorID = doctorID;
        this.name = name;
        this.surname = surname;
    }
    no usages
    public void setDoctorID(int doctorID) { this.doctorID = doctorID; }
    1 usage
    public int getDoctorID() { return doctorID; }
    no usages
    public void setName(String name) { this.name = name; }
    no usages
    public String getName() { return name; }
    no usages
    public void setSurname(String surname) { this.surname = surname; }
    no usages
    public String getSurname() { return surname; }
```

Figura 14: Porzione di codice relativa alla classe Doctor

3.3.4 HospitalizationRecord

La seguente classe rappresenta la cartella di ricovero che viene assegnata ad ogni paziente nel reparto. Essa è caratterizzata da un intero recordID, da un riferimento a un oggetto di tipo Patient e una lista di DailyCheck. Di particolare rilevanza è il metodo addDailyCheck(DailyCheck dc) che viene utilizzato per inserire nella lista sopra citata un nuovo oggetto di tipo DailyCheck nel momento in cui la diaria viene confermata dal medico strutturato.

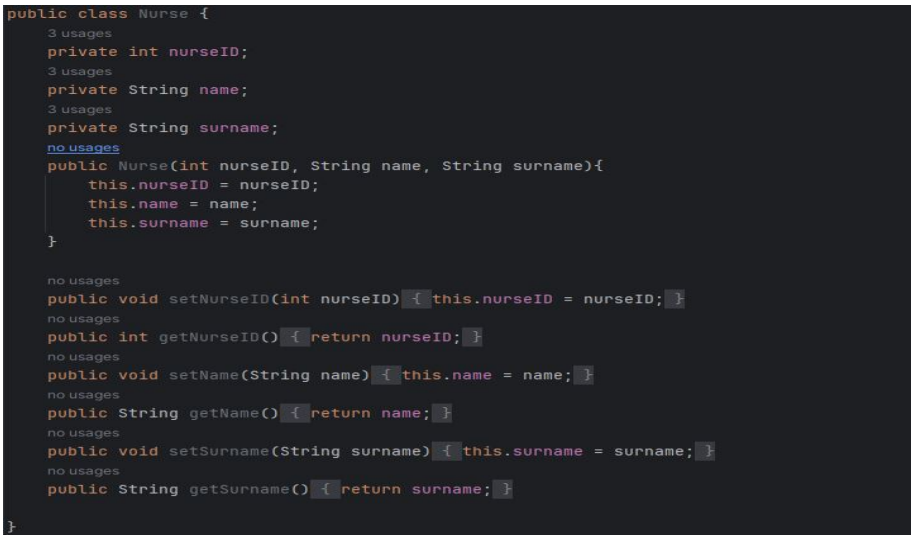
```
public class HospitalizationRecord {
    4 usages
    private int recordID;
    3 usages
    private Patient patient;
    3 usages
    private ArrayList<DailyCheck> checks= new ArrayList<>();;
    2 usages
    public HospitalizationRecord(){
    }
    1 usage
    public HospitalizationRecord(int recordID) { this.recordID = recordID; }
    2 usages
    public HospitalizationRecord(int recordID, Patient patient){
        this.recordID = recordID;
        this.patient = patient;

    }
```

Figura 15: Porzione di codice relativa alla classe HospitalizationRecord

3.3.5 Nurse

Questa classe è stata implementata per rappresentare un infermiere ed è caratterizzata da un intero nurseID e da due riferimenti a oggetti di tipo String per il nome e il cognome.



```
public class Nurse {
    3 usages
    private int nurseID;
    3 usages
    private String name;
    3 usages
    private String surname;
    no usages
    public Nurse(int nurseID, String name, String surname){
        this.nurseID = nurseID;
        this.name = name;
        this.surname = surname;
    }

    no usages
    public void setNurseID(int nurseID) { this.nurseID = nurseID; }
    no usages
    public int getNurseID() { return nurseID; }
    no usages
    public void setName(String name) { this.name = name; }
    no usages
    public String getName() { return name; }
    no usages
    public void setSurname(String surname) { this.surname = surname; }
    no usages
    public String getSurname() { return surname; }

}
```

Figura 16: Porzione di codice relativa alla classe Nurse

3.3.6 Patient

La seguente classe indica un paziente, in particolare presenta: un intero patientID, associato ad ogni paziente in maniera univoca; due riferimenti ad oggetti di tipo String per il nome e il cognome; un riferimento a un oggetto di tipo LocalDate che indica la data di nascita; un riferimento a un oggetto di tipo String che viene utilizzato per indicare il sesso, un riferimento a un oggetto di tipo Bed ed infine un riferimento a un oggetto di tipo HospitalizationRecord.

```
public class Patient {
    4 usages
    private int patientID;
    3 usages
    private String name;
    3 usages
    private String surname;
    3 usages
    private LocalDate birthDate;
    3 usages
    private String gender;
    3 usages
    private Bed bed;
    2 usages
    private HospitalizationRecord hospitalizationRecord;
    public Patient(){

    }
    public Patient (int patientID) { this.patientID = patientID; }
```

Figura 17: Porzione di codice relativa alla classe Patient

3.3.7 NursePage

Questa classe fa parte dell'implementazione della business logic e presenta una serie di metodi fondamentali per la gestione dei pazienti al momento del ricovero. Innanzitutto, il metodo getAvailableBed() ritorna un riferimento a un oggetto di tipo Bed che presenta l'attributo available = true e che può quindi essere assegnato al paziente mediante il metodo assignBed(Patient p); lo stesso letto al momento delle dimissioni può essere liberato chiamando freeBed(). Sono presenti, inoltre, due metodi che permettono rispettivamente di aggiungere e di rimuovere un record relativo al paziente dalla specifica tabella.

```
public Bed getAvailableBed(){
    Bed b = null;
    for(Bed item : beds){
        if(item.isAvailable()) {
            b = item;
            break;
        }
    }
    return b;
}
4 usages
public void assignBed(Patient p){
    Bed b = getAvailableBed();
    p.setBed(b);
    b.setAvailable(false);
}
```

Figura 18: Prima porzione di codice relativa alla classe NursePage

```
public void freeBed(Bed b) { b.setAvailable(true); }
1 usage
public void savePatient(int patientID, String name, String surname, LocalDate birthDate, String gender)
    Patient p = new Patient(patientID, name, surname, birthDate, gender, bed: null);
    assignBed(p);
    pd.savePatient(p);
}
1 usage
public void deletePatient(Patient p) throws SQLException{
    pd.deletePatient(p.getPatientID());
}
2 usages
public ArrayList<Bed> getBeds() { return beds; }
```

Figura 19: Seconda porzione di codice relativa alla classe NursePage

3.3.8 ResidentPage

Questa classe costituisce anch'essa la business logic dell'applicativo ed in particolare presenta dei metodi che corrispondono alle funzioni svolte da uno specializzando. Il metodo assignHospitalizationRecord(int recordID, Patient p) permette di associare al paziente una cartella di ricovero, mentre il metodo saveHospitalizationRecord(HospitalizationRecord hr) permette di aggiungere la stessa al database. ResidentPage, inoltre, estende la classe Observable appartente al package java.util al fine di poter notificare al medico strutturato la creazione di una nuova diaria. Di particolare importanza è, infatti, il metodo createDailyCheck() che si occupa inoltre di notificare gli osservatori mediante il metodo notifyObservers().

```
public void assignHospitalizationRecord(int recordID, Patient p){
    HospitalizationRecord hr = new HospitalizationRecord(recordID);
    hr.setPatient(p);
    p.setHospitalizationRecord(hr);
}
1 usage
public void saveHospitalizationRecord(HospitalizationRecord hr) { hrd.saveRecord(hr); }
2 usages
public void createDailyCheck(int dailyCheckID, Patient patient, LocalDate date, int temperature,
                             int oxygenSaturation, int diastolicPressure, int systolicPressure,
                             int heartRate, String symptoms){
    DailyCheck dc = new DailyCheck(dailyCheckID, patient, resident, date, temperature,
            oxygenSaturation, diastolicPressure, systolicPressure, heartRate, symptoms);
    setChanged();
    notifyObservers(dc);
}
```

Figura 20: Porzione di codice relativa alla classe ResidentPage

3.3.9 SpecialistPage

Questa classe è l'ultima del package "businesslogic" ed implementa il metodo update() relativo all'interfaccia Observer appartenente al package java.util. Essa presenta i metodi confirmCheck(DailyCheck dc) e addCheck(DailyCheck dc) che rispettivamente aggiungono la diaria alla cartella di ricovero del paziente e la salvano nel database.

```
2 usages
public SpecialistPage(int doctorID, String name, String surname, Observable obs){
    specialist = new Doctor(doctorID, name, surname);
    dcd = new DailyCheckDaoImpl();
    obs.addObserver( o: this);
}

@Override
public void update(Observable residentPage, Object dailyCheck) {
    DailyCheck dc = (DailyCheck) dailyCheck;
    checks.add(dc);
}
1 usage
public void confirmCheck(DailyCheck dc) { dc.getPatient().getHospitalizationRecord().addCheck(dc); }
1 usage
public void addCheck(DailyCheck dc) { dcd.saveRecord(dc); }
```

Figura 21: Porzione di codice relativa alla classe SpecialistPage

3.3.10 PatientDaoImpl

Questa classe implementa i metodi dell'interfaccia PatientDao che si mostra di seguito.

```
1 usage  1 implementation
public interface PatientDao {
    1 usage  1 implementation
    ArrayList<Patient> getAllPatients();
    2 usages  1 implementation
    Patient getPatientByID(int patientID);
    1 usage  1 implementation
    void savePatient(Patient p);
    2 usages  1 implementation
    void deletePatient(int patientID);
}
```

Figura 22: Interfaccia PatientDao

La connessione con il database viene stabilita mediante il metodo connect(). Il metodo getAllPatients() permette di ottenere una lista di tutti i pazienti ricoverati, mentre getPatientByID(int patientID) permette di individuare un record relativo a un paziente nel database conoscendo il suo ID. I due metodi rimanenti, invece, permettono rispettivamente di salvare e rimuovere il paziente dalla tabella. Si riportano di seguito le implementazioni dei metodi elencati:

```
public class PatientDaoImpl implements PatientDao {
    5 usages
    Connection con = null;

    4 usages
    public void connect(){
        String url = "jdbc:postgresql://localhost:5432/HospitalizationManagement";
        String user = "postgres";
        String password = "marikarucco";
        try {
            con = DriverManager.getConnection(url, user, password);
        } catch(SQLException e){
            System.out.println("Connection failed");
        }
    }
```

Figura 23: Implementazione del metodo connect()

```
    @Override
    public ArrayList<Patient> getAllPatients() {
        ArrayList<Patient> patients = new ArrayList<>();
        connect();
        String sql = "SELECT * from patient";
        try{
            Statement statement = con.createStatement();
            ResultSet rs = statement.executeQuery(sql);
            while(rs.next()){
                Patient patient = new Patient();
                patient.setPatientID(rs.getInt( columnIndex: 1));
                patient.setName(rs.getString( columnIndex: 2));
                patient.setSurname(rs.getString( columnIndex: 3));
                patient.setBirthDate(rs.getObject( columnIndex: 4, LocalDate.class));
                patient.setGender(rs.getString( columnIndex: 5));
                patient.setBed(new Bed(rs.getInt( columnIndex: 6)));
                patients.add(patient);
            }
        }catch(SQLException e){
            System.out.println("Failed to get list of patients");
        }
        return patients;
    }
```

Figura 24: Implementazione del metodo getAllPatients()

```java
@Override
public Patient getPatientByID(int patientID) {
    Patient patient = new Patient();
    connect();
    String sql = "SELECT * from patient where patient_id=?";
    try{
        PreparedStatement statement = con.prepareStatement(sql);
        statement.setInt( parameterIndex: 1, patientID);
        ResultSet rs = statement.executeQuery();
        rs.next();
        patient.setPatientID(patientID);
        patient.setName(rs.getString( columnIndex: 2));
        patient.setSurname(rs.getString( columnIndex: 3));
        patient.setBirthDate(rs.getObject( columnIndex: 4, LocalDate.class));
        patient.setGender(rs.getString( columnIndex: 5));
        patient.setBed(new Bed(rs.getInt( columnIndex: 6)));
    }catch(SQLException e){
       System.out.println("Failed to get patient by id");
    }
    return patient;
}
```

Figura 25: Implementazione del metodo getPatientByID(int patientID)

```java
@Override
public void savePatient(Patient p) {
    connect();
    String sql = "INSERT INTO patient(patient_id, name, surname, birth_date, gender, bed_id) VALUES (?,?,?,?,?,?)";
    try{
        PreparedStatement statement = con.prepareStatement(sql);
        statement.setInt( parameterIndex: 1, p.getPatientID());
        statement.setString( parameterIndex: 2, p.getName());
        statement.setString( parameterIndex: 3, p.getSurname());
        statement.setObject( parameterIndex: 4, p.getBirthDate());
        statement.setString( parameterIndex: 5, p.getGender());
        statement.setInt( parameterIndex: 6, p.getBed().getBedID());
        statement.executeUpdate();
    }catch(SQLException e){
        System.out.println("Failed to add new patient to db");
    }
}
```

Figura 26: Implementazione del metodo savePatient(Patient p)

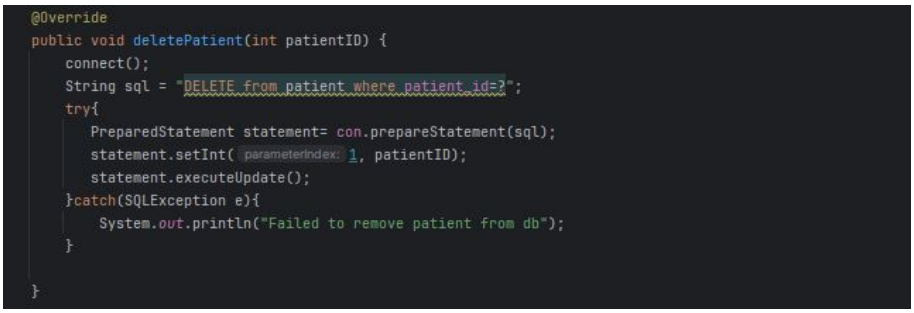


Figura 27: Implementazione del metodo deletePatient(int patientID)

3.3.11 DailyCheckDaoImpl

Questa classe implementa i metodi dell'interfaccia DailyCheckDao che viene riportata di seguito:

Figura 28: Interfaccia DailyCheckDao

Anche in questo caso la connessione con il database viene stabilita mediante il metodo connect() (vedi Figura 23).
Il metodo getAllRecords(Patient p) restituisce una lista contenente tutte le diarie relative al paziente indicato nei paramentri; il metodo getRecordByID(int recordID) permette di individuare una specifica diaria conoscendo l'attributo dailyCheckID; gli ultimi due metodi, invece, permettono di salvare e di eliminare record dalla database. Si riportano di seguito le implementazioni:

```java
public ArrayList<DailyCheck> getAllRecords(Patient p) {
    ArrayList<DailyCheck> checks = new ArrayList<>();
    connect();
    int patientID = p.getPatientID();
    String sql = "SELECT * from daily_check where patient_id=?";
    try{
        PreparedStatement statement = con.prepareStatement(sql);
        statement.setInt( parameterIndex: 1, patientID);
        ResultSet rs = statement.executeQuery();
        while(rs.next()){
            DailyCheck dc = new DailyCheck();
            dc.setDoctor(new Doctor(rs.getInt( columnIndex: 2)));
            dc.setPatient(new Patient(rs.getInt( columnIndex: 3)));
            dc.setDate(rs.getObject( columnIndex: 4, LocalDate.class));
            dc.setTemperature(rs.getInt( columnIndex: 5));
            dc.setOxygenSaturation(rs.getInt( columnIndex: 6));
            dc.setDiastolicPressure(rs.getInt( columnIndex: 7));
            dc.setSystolicPressure(rs.getInt( columnIndex: 8));
            dc.setHeartRate(rs.getInt( columnIndex: 9));
            dc.setSymptoms(rs.getString( columnIndex: 10));
            checks.add(dc);
        }
    }catch(SQLException e){
        System.out.println("Failed to get list of daily checks");
    }
    return checks;
}
```

Figura 29: Implementazione del metodo getAllRecords(Patient p)

```java
@Override
public DailyCheck getRecordByID(int recordID) {
    DailyCheck dc = new DailyCheck();
    connect();
    String sql = "SELECT * from daily_check where daily_check_id=?";
    try{
        PreparedStatement statement = con.prepareStatement(sql);
        statement.setInt( parameterIndex: 1, recordID);
        ResultSet rs = statement.executeQuery();
        rs.next();
        dc.setDoctor(new Doctor(rs.getInt( columnIndex: 2)));
        dc.setPatient(new Patient(rs.getInt( columnIndex: 3)));
        dc.setDate(rs.getObject( columnIndex: 4, LocalDate.class));
        dc.setTemperature(rs.getInt( columnIndex: 5));
        dc.setOxygenSaturation(rs.getInt( columnIndex: 6));
        dc.setDiastolicPressure(rs.getInt( columnIndex: 7));
        dc.setSystolicPressure(rs.getInt( columnIndex: 8));
        dc.setHeartRate(rs.getInt( columnIndex: 9));
        dc.setSymptoms(rs.getString( columnIndex: 10));
    }catch(SQLException e){
        System.out.println("Failed to get daily check by id");
    }
    return dc;
}
```

Figura 30: Implementazione del metodo getRecordByID(int recordID)

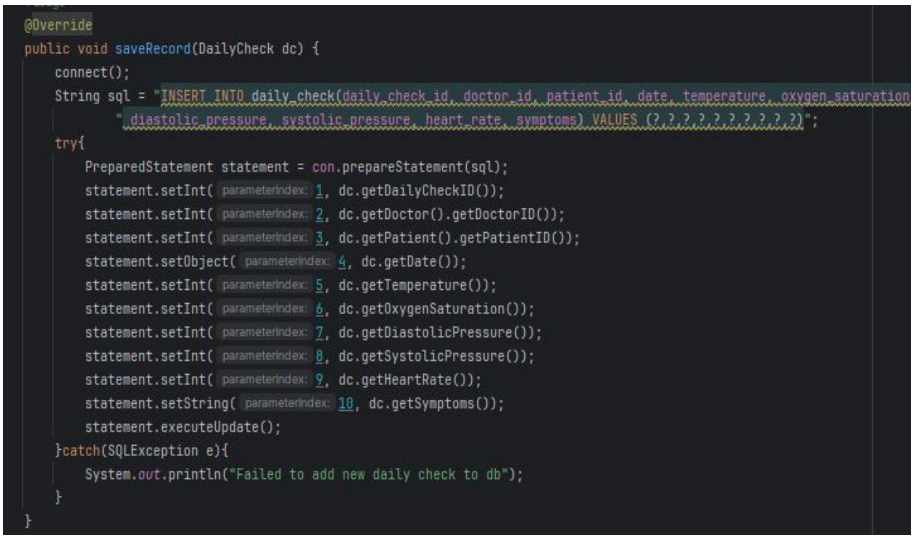


Figura 31: Implementazione del metodo saveRecord(DailyCheck dc)

Figura 32: Implementazione del metodo deleteRecord(int recordID)

3.3.12 HospitalizationRecordDaoImpl

Questa classe implementa i metodi dell'interfaccia HospitalizationRecordDao nella figura seguente:

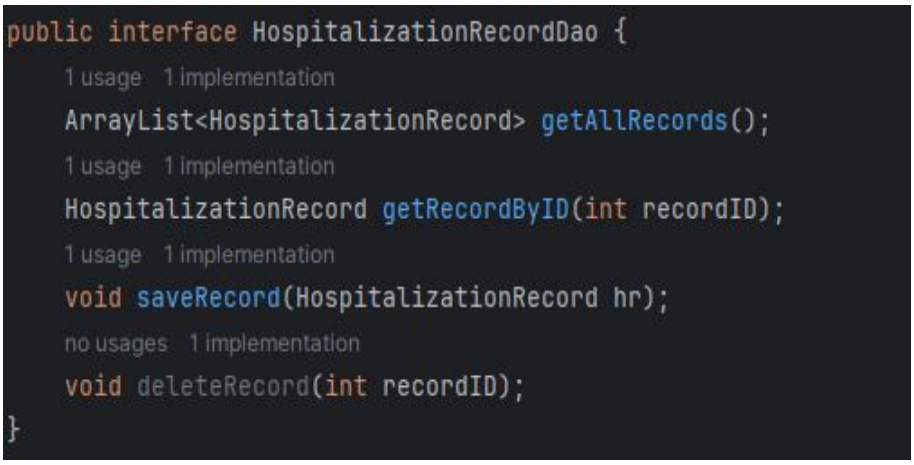


Figura 33: Interfaccia HospitalizationRecordDao

Ancora una volta per stabilire una connessione con il database è stato utilizzato il metodo connect() (vedi Figura 23).
Il metodo getAllRecords() permette di avere una lista di tutte le cartelle di ricovero dei pazienti; il metodo getRecordByID(int recordID) ritorna una cartella di ricovero in base al valore dell'ID fornito, infine, i metodi saveRecord(HospitalizationRecord hr) e deleteRecord(int recordID) permettono di aggiungere e rimuovere una cartella di ricovero dal database. Come fatto precedentemente, si mostrano le implementazioni dei metodi sopra descritti:

```
public ArrayList<HospitalizationRecord> getAllRecords() {
    ArrayList<HospitalizationRecord> records = new ArrayList<>();
    connect();
    String sql = "SELECT * from hospitalization_record";
    try{
        Statement statement = con.createStatement();
        ResultSet rs = statement.executeQuery(sql);
        while(rs.next()){
            HospitalizationRecord hr = new HospitalizationRecord();
            hr.setRecordID(rs.getInt( columnIndex: 1));
            hr.setPatient(new Patient(rs.getInt( columnIndex: 2)));
            records.add(hr);
        }
    }catch(SQLException e){
        System.out.println("Failed to get list of hospitalization records");
    }
    return records;
}
```

Figura 34: Implementazione del metodo getAllRecords()

```java
@Override
public HospitalizationRecord getRecordByID(int recordID) {
    HospitalizationRecord hr = new HospitalizationRecord();
    connect();
    String sql = "SELECT * from hospitalization_record where record_id=?";
    try{
        PreparedStatement statement = con.prepareStatement(sql);
        statement.setInt( parameterIndex: 1, recordID);
        ResultSet rs = statement.executeQuery();
        rs.next();
        hr.setRecordID(recordID);
        hr.setPatient(new Patient(rs.getInt( columnIndex: 2)));
    }catch(SQLException e){
        System.out.println("Failed to get patient by id");
    }
    return hr;
}
```

Figura 35: Implementazione del metodo getRecordByID(int recordID)

```java
public void saveRecord(HospitalizationRecord hr) {
    connect();
    String sql = "INSERT INTO hospitalization_record(record_id, patient_id) VALUES (?,?)";
    try{
        PreparedStatement statement = con.prepareStatement(sql);
        statement.setInt( parameterIndex: 1, hr.getRecordID());
        statement.setInt( parameterIndex: 2, hr.getPatient().getPatientID());
        statement.executeUpdate();
    }catch(SQLException e){
        System.out.println("Failed to add new hospitalization record to db");
    }
}

no usages
@Override
public void deleteRecord(int recordID) {
    connect();
    String sql = "DELETE from hospitalization_record where record_id=?";
    try{
        PreparedStatement statement= con.prepareStatement(sql);
        statement.setInt( parameterIndex: 1, recordID);
        statement.executeUpdate();
    }catch(SQLException e){
        System.out.println("Failed to remove hospitalization record from db");
    }
```

Figura 36: Implementazione dei metodi saveRecord(HospitalizationRecord hr) e deleteRecord(int recordID)

3.4 Design patterns

Nello sviluppo di questo applicativo si è reso necessario l'impiego di due design patterns:

- Observer
- DAO

3.4.1 Observer

Questo design pattern viene utilizzato per creare una relazione tra un oggetto osservato e un insieme di oggetti osservatori. Nel momento in cui l'osservato cambia il suo stato, tutti gli osservatori vengono notificati di tale cambiamento. In particolare, nell'ambito di questo progetto si rende necessario notificare il medico strutturato ogni qual volta uno specializzando crei una nuova diaria. Per questo motivo, la classe "ResidentPage" svolge il ruolo di oggetto osservato, mentre la classe "SpecialistPage" è l'osservatore; quando viene creata una nuova diaria attraverso il metodo createDailyCheck, il medico strutturato viene notificato dell'evento e riceve la diaria attraverso il metodo notifyObservers, successivamente quest'ultimo utilizzerà il metodo update() per poterla confermare ed aggiungere alla lista di diarie del paziente. In questo caso il design pattern Observer è stato utilizzato in modalità push, la quale prevede che l'oggetto osservato notifichi direttamente gli osservatori.

3.4.2 DAO

Questo design pattern viene utilizzato per gestire la persistenza dei dati e permette di mantenere su due livelli differenti la business logic e la logica di accesso ai dati. In generale, si realizzano:

- una classe per ogni tabella del database, che quindi apparterrà al modello di dominio;
- un'interfaccia che contiene metodi che corrispondono alle operazioni che possono essere effettuate su una tabella per salvare e eliminare dati o semplicemente per estrarli (operazioni CRUD);
- un'implementazione per ogni interfaccia.

Nell'ambito di questo progetto, il design pattern DAO è stato utilizzato per le classi Patient, DailyCheck e HospitalizationRecord, quindi per gestire i dati relativi ai pazienti, alle loro diarie e alle cartelle di ricovero.

3.5 Disposizione delle classi nei package

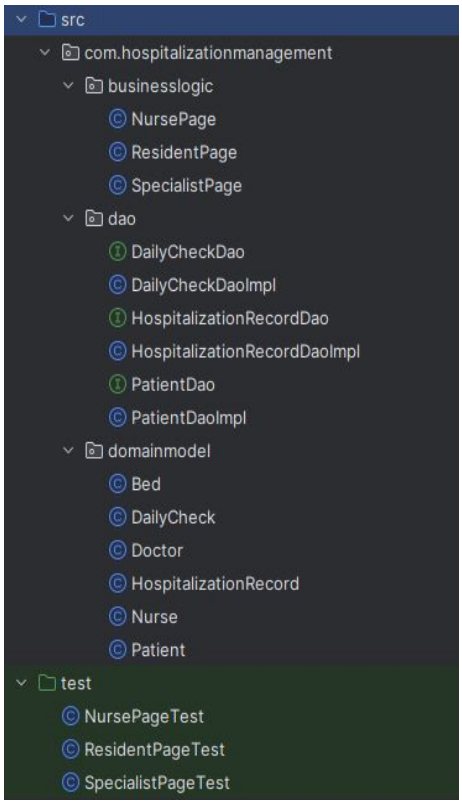


Figura 37: Disposizione delle classi

4 Unit Test

Al fine di verificare che il funzionamento di ciascuna parte dell'applicativo soddisfi i requisiti funzionali attesi sono stati realizzati dei casi di test per le classi che implementano la business logic dell'applicativo. A tale scopo è stato utilizzato il framework JUnit 5.8.1.

4.0.1 NursePageTest

In questo caso sono stati testati tutti i metodi relativi alla classe NursePage, in quanto ritenuti tutti rilevanti per il corretto funzionamento dell'applicativo.

```java
@Test
public void testAssignBed(){
    np.assignBed(p1);
    assertFalse(p1.getBed().isAvailable());
    Patient p2 = new Patient( patientID: 2,  name: "Franca",  surname: "Verdi", LocalDate.of( year: 1962,  month:
    np.assignBed(p2);
    assertFalse(np.getBeds().get(0).isAvailable());
    assertFalse(np.getBeds().get(1).isAvailable());
}

@Test
public void testFreeBed(){
    np.assignBed(p1);
    np.freeBed(p1.getBed());
    assertTrue(p1.getBed().isAvailable());
}
```

Figura 38: Casi di test relativi alla classe NursePage

```java
@Test
public void testSavePatient() throws SQLException {
    np.savePatient( patientID: 2,  name: "Franca",  surname: "Verdi", LocalDate.of( year: 1962,  month: 10,  dayO
    assertEquals(pd.getPatientByID( patientID: 2).getName(),  actual: "Franca");
    assertNotEquals(pd.getPatientByID( patientID: 2).getSurname(), actual: "Bianchi");
}

@Test
public void testDeletePatient() throws SQLException{
    np.deletePatient(p1);
    assertNotEquals(pd.getAllPatients().size(), actual: 2);
}
```

Figura 39: Casi di test relativi alla classe NursePage

4.0.2 ResidentPageTest

Anche per questa classe sono stati testati tutti i metodi. Di particolare importanza sono i casi di test relativi all'interazione tra Observable (ResidentPage) e Observer (SpecialistPage).

```java
@Test
public void testAssignHospitalizationRecord(){
    rp.assignHospitalizationRecord( recordID: 1, p1);
    assertEquals(p1.getHospitalizationRecord().getRecordID(), actual: 1);
    assertEquals(p1.getHospitalizationRecord().getPatient(), p1);
}

@Test
public void testSaveHospitalizationRecord() throws SQLException {
    HospitalizationRecord hr = new HospitalizationRecord( recordID: 1, p1);
    rp.saveHospitalizationRecord(hr);
    assertEquals(hrd.getRecordByID( recordID: 1).getPatient().getPatientID(), p1.getPatientID());
    assertEquals(hrd.getAllRecords().size(), actual: 1);
}
```

Figura 40: Casi di test relativi alla classe ResidentPage

```java
@Test
public void testCreateDailyCheck() throws IllegalArgumentException{
    rp.createDailyCheck( dailyCheckID: 1, p1, LocalDate.now(), temperature: 36, oxygenSaturation: 85,
    assertEquals(sp.getChecks().get(0). getTemperature(), actual: 36);
    assertEquals(sp.getChecks().get(0).getOxygenSaturation(), actual: 85);
    assertEquals(sp.getChecks().get(0).getDiastolicPressure(), actual: 75);
    assertEquals(sp.getChecks().get(0).getSystolicPressure(), actual: 120);
    rp.createDailyCheck( dailyCheckID: 2, p1, LocalDate.now(), temperature: 38, oxygenSaturation: 75,
    assertEquals(sp.getChecks().size(), actual: 2);
}
```

Figura 41: Casi di test relativi alla classe ResidentPage

4.0.3 SpecialistPageTest

In quest'ultima sottosezione si rappresentano i casi di test relativi alla classe SpecialistPage.

```java
public void testConfirmCheck() throws IllegalArgumentException{
    DailyCheck dc = new DailyCheck( dailyCheckID: 1, p1, sp.getSpecialist(), LocalDate.now(), temperature: 36,
    sp.confirmCheck(dc);
    assertEquals(p1.getHospitalizationRecord().getChecks().get(0).getTemperature(), actual: 36);
    assertNotEquals(p1.getHospitalizationRecord().getChecks().get(0).getDoctor().getName(), actual: "Giovan
}
```

Figura 42: Casi di test relativi alla classe SpecialistPage

```java
@Test
public void testAddCheck() throws SQLException, IllegalArgumentException {
    DailyCheck dc = new DailyCheck( dailyCheckID: 2, p1, sp.getSpecialist(), LocalDate.now(),
    sp.addCheck(dc);
    assertEquals(dcd.getRecordByID( recordID: 2).getHeartRate(),  actual: 80);
    assertEquals(dcd.getAllRecords(p1).size(),  actual: 1);
}
```

Figura 43: Casi di test relativi alla classe SpecialistPage