



UNIVERSITÀ DEGLI STUDI DI FIRENZE

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Elaborato Ingegneria del Software  
Luca Bindini

**Applicativo Java che simula la  
gestione di un reparto ospedaliero  
adibito a pazienti affetti da Covid-19**

A.A. 2019-2020

<b>Indice</b>	<b>1</b>
<b>Introduzione</b>	<b>3</b>
<b>1 Progettazione</b>	<b>5</b>
1.1 Casi d'Uso . . . . .	5
1.2 Class Diagram . . . . .	6
1.3 Mock-ups . . . . .	7
<b>2 Implementazione</b>	<b>9</b>
2.1 Classi ed Interfacce . . . . .	9
2.1.1 Doctor . . . . .	10
2.1.2 CovidWard . . . . .	10
2.1.3 CovidPatient . . . . .	11
2.1.4 Pathology . . . . .	11
2.1.5 LungVentilator . . . . .	12
2.1.6 MedicalRecord . . . . .	12
2.2 Design Patterns . . . . .	13
2.2.1 Singleton . . . . .	13
2.2.2 Builder . . . . .	14
2.2.3 Observer . . . . .	14
2.2.4 Strategy . . . . .	15
2.2.5 MVC . . . . .	16
2.3 Ulteriori Dettagli Implementativi . . . . .	17
<b>3 Testing ed Esecuzione</b>	<b>18</b>
3.1 Unit Testing . . . . .	18
3.1.1 DoctorTest . . . . .	19
3.1.2 CovidWardTest . . . . .	19
3.1.3 CovidPatientTest . . . . .	20
3.1.4 PathologyTest . . . . .	20
3.1.5 LungVentilatorTest . . . . .	20

---

3.1.6	MedicalRecordTest . . . . .	21
3.1.7	RecoveryRateStrategyTest . . . . .	21
3.2	Sequence Diagram . . . . .	22

### Motivazione e Contenuti

L'elaborato consiste nella realizzazione di un applicativo, scritto in linguaggio Java, che simuli in modo semplicistico la gestione di un reparto ospedaliero adibito a pazienti affetti da Covid-19.

Attraverso l'applicazione l'utente (medico) inserisce i propri dati per essere riconosciuto dal sistema.

A questo punto egli può aggiungere al reparto, finché sono disponibili posti letto, nuovi pazienti affetti da Covid e successivamente ad ogni paziente può aggiungere una o più patologie pregresse precedentemente diagnosticate.

Di ogni paziente quindi si conosce la sua anagrafica ed il numero e il tipo di patologie preesistenti, se ci sono.

Ad ogni paziente è associata una cartella clinica con un codice associato che, oltre ad avere le informazioni di base sul paziente stesso, presenta un Recovery Rate ossia una probabilità di recupero da Covid che ovviamente dipenderà dall'età del paziente, dalla presenza o meno di patologie pregresse e nel secondo caso dal numero e dalla tipologia di tali patologie. Questo rate varierà in modo dinamico all'inserimento di patologie pregresse ad un determinato soggetto.

Un parametro fondamentale per il paziente affetto da Covid è la saturazione dell'ossigeno nel sangue (che può essere aggiornata dal medico) ed eventualmente il medico può assegnare un macchinario per la respirazione assistita di due diverse tipologie: a pressione positiva o a pressione negativa.

Quando un paziente guarisce da Covid e quindi può essere potenzialmente dimesso, il medico può impostare la negatività al tampone per il paziente in questione ed eventualmente dimettere in una volta sola tutti i pazienti risultati ormai negativi al tampone, liberando così di fatto posti letto che potranno essere assegnati successivamente a chi ne avesse bisogno.

## Metodo

Per la realizzazione di tale applicazione è stato utilizzato il linguaggio di programmazione Java attraverso l'IDE IntelliJ IDEA.

Nella fase di progettazione sono stati identificati i casi d'uso e rappresentati attraverso gli use case diagrams.

È stata inoltre definita realizzata una logica di dominio in prospettiva di specifica/implementazione attraverso un class diagram in UML.

In questa fase sono stati anche disegnati alcuni mock-ups di come dovesse essere l'interfaccia finale dell'applicativo utente.

Nella struttura principale del programma (contenuta nel package src/covid/domain) sono stati adottati alcuni pattern comportamentali come il pattern Observer tra la classe che rappresenta il paziente e la sua cartella clinica, il pattern Strategy per il calcolo del Recovery Rate a seconda che il soggetto presenti o meno patologie pregresse e il pattern Singleton per la classe che rappresenta il reparto stesso.

Inoltre è stato utilizzato il pattern creazionale Builder per la creazione dei pazienti.

Per quanto concerne l'interazione con l'utente è stata realizzata una GUI (graphic user interface) attraverso il framework Java Swing adottando il pattern strutturale MVC (model-view-controller) dove il model è rappresentato dalla nostra domain logic contenuta nel package domain e la business logic (realizzazione dei casi d'uso) è affidata sia al model che al controller.

I vari rate sulle patologie preesistenti e sulla variazione in base all'età dei pazienti sono stati reperiti direttamente dai dati offerti dal WHO (World Health Organization), caricati all'interno del programma attraverso dei semplici file di testo (.txt) e successivamente rielaborati.

Una parte fondamentale dello sviluppo è data anche dalla realizzazione di alcune classi di testing (più precisamente di Unit Testing) attraverso il framework JUnit 5 integrato direttamente nell'ambiente di sviluppo.

L'intero progetto è stato versionato con Git e disponibile su GitHub al link:

<https://github.com/lucabindini/Covid-19Ward>.

## 1.1 Casi d'Uso

Dopo aver definito il problema in questione sono stati individuati gli attori in gioco e i vari casi d'uso.

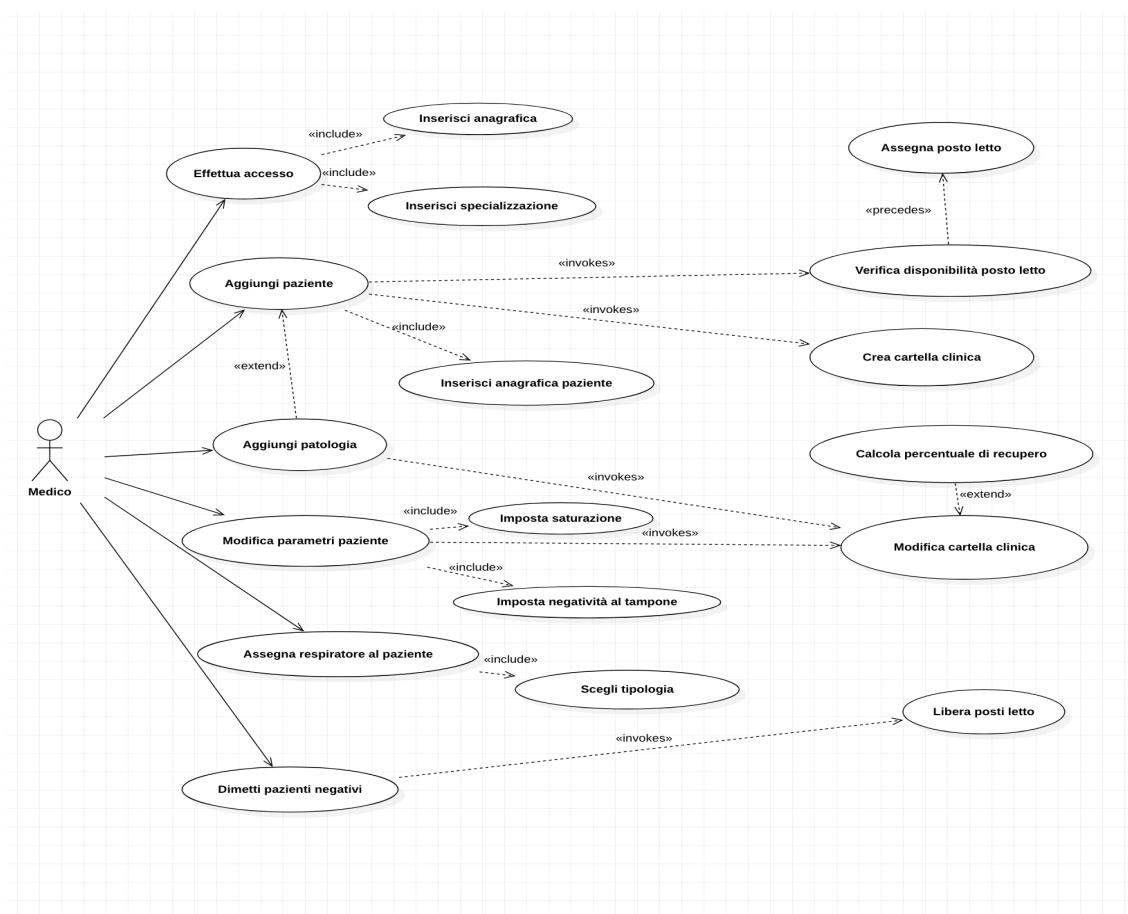


Figura 1.1: Use Case Diagram

## 1.2 Class Diagram

Qui di seguito la realizzazione del diagramma UML che descrive la logica di dominio in prospettiva di implementazione.

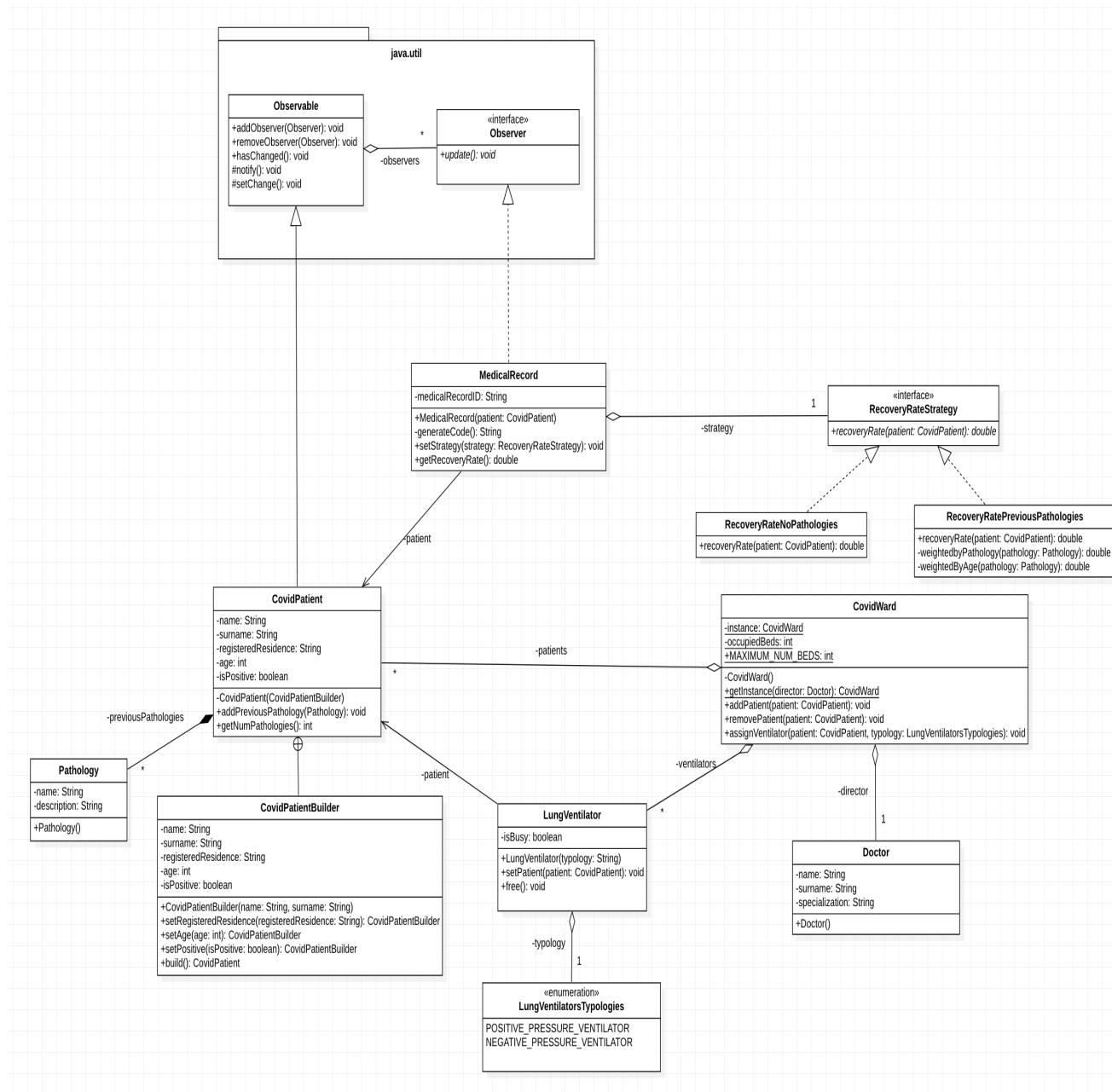



Figura 1.2: Uml Diagram

## 1.3 Mock-ups

Sono stati realizzati alcuni mock-ups dell'interfaccia utente finale.



The image shows a mock-up of a login window titled "Login". The window has a light gray background and a title bar with standard macOS window controls (red, yellow, and gray buttons). The main content area features the title "Login Direttore Reparto" in bold red text. Below the title, there are three text input fields. The first field is labeled "Nome:" and contains the text "Mario". The second field is labeled "Cognome:" and contains the text "Rossi". The third field is labeled "Specializzazione:" and contains the text "Pneumologia". At the bottom center of the window, there is a blue button with the text "Login".

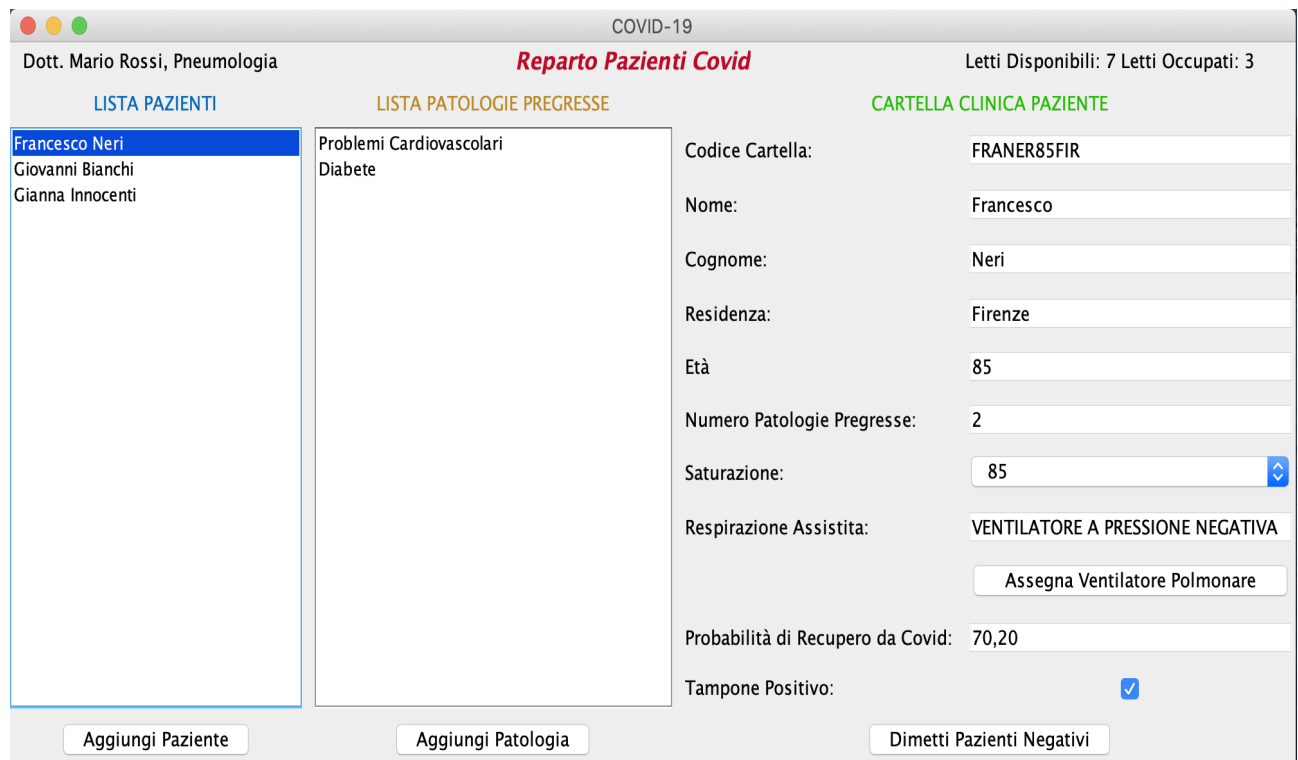
**Figura 1.3:** Login Frame



The image shows a mock-up of a patient insertion window titled "Inserisci Paziente". The window has a light gray background and a title bar with standard macOS window controls (red, yellow, and gray buttons). The main content area features the title "Inserisci Nuovo Paziente Covid" in bold red text. Below the title, there are four input fields. The first field is labeled "Nome:" and contains the text "Francesco". The second field is labeled "Cognome:" and contains the text "Neri". The third field is labeled "Residenza:" and contains the text "Firenze". The fourth field is labeled "Età:" and contains the text "85" next to a blue dropdown arrow. At the bottom center of the window, there is a blue button with the text "Inserisci Paziente".

**Figura 1.4:** Inserimento Paziente Frame





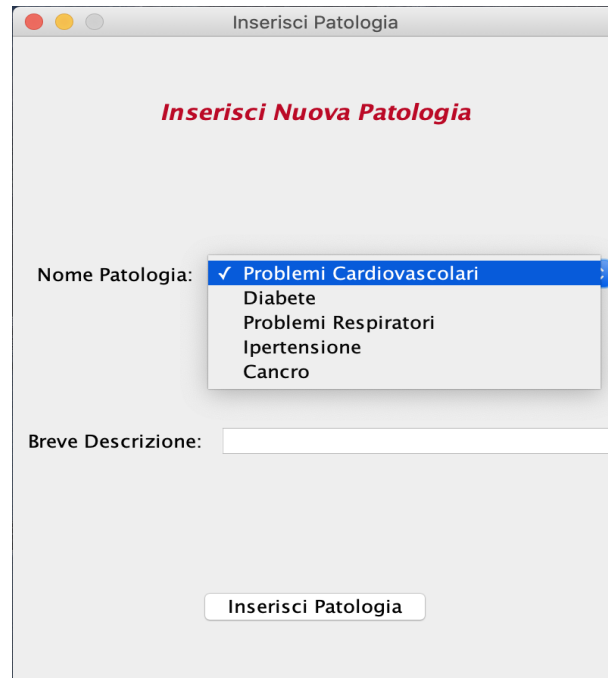
The main frame of the application is titled "COVID-19" and "Reparto Pazienti Covid". It is divided into three main sections: "LISTA PAZIENTI", "LISTA PATOLOGIE PREGRESSE", and "CARTELLA CLINICA PAZIENTE".

**LISTA PAZIENTI:** A list of patients with the following entries: Francesco Neri, Giovanni Bianchi, and Gianna Innocenti. Below the list is a button "Aggiungi Paziente".

**LISTA PATOLOGIE PREGRESSE:** A list of pre-existing conditions with the following entries: Problemi Cardiovascolari and Diabete. Below the list is a button "Aggiungi Patologia".

**CARTELLA CLINICA PAZIENTE:** A form for patient data. It includes fields for: Codice Cartella (FRANER85FIR), Nome (Francesco), Cognome (Neri), Residenza (Firenze), Età (85), Numero Patologie Pregresse (2), Saturazione (85), Respirazione Assistita (VENTILATORE A PRESSIONE NEGATIVA), Probabilità di Recupero da Covid (70,20), and Tampone Positivo (checked). There is a button "Assegna Ventilatore Polmonare" and a button "Dimetti Pazienti Negativi".

Figura 1.5: Main Frame



The "Inserisci Patologia" frame is titled "Inserisci Nuova Patologia". It contains a form for adding a new pathology. The "Nome Patologia:" field has a dropdown menu with the following options: Problemi Cardiovascolari (selected), Diabete, Problemi Respiratori, Ipertensione, and Cancro. The "Breve Descrizione:" field is empty. Below the form is a button "Inserisci Patologia".

Figura 1.6: Inserimento Patologia Frame

## 2.1 Classi ed Interfacce

Per l'implementazione sono state definite nuove classi ed interfacce o riutilizzate quelle della libreria standard di Java (e.g le classi e le interfacce contenute nel package `java.util`). Le principali classi dell'applicazione contenute nel package `com.covid.domain` sono:

1. **Doctor**
2. **CovidWard**
3. **CovidPatient**
4. **Pathology**
5. **LungVentilator**
6. **MedicalRecord**
7. **RecoveryRateStrategy** (interfaccia avente due diverse realizzazioni **RecoveryRateNoPathologies** e **RecoveryRatePreviousPathologies**)

Oltre a queste classi principali ne sono state definite delle altre sia all'interno del package `com.covid.domain` sia all'interno degli altri due package in `src` (`com.covid.controller` e `com.covid.view`).

In particolare sono state definite alcune classi per la gestione delle **eccezioni**, la classe **MainController** (che ha il ruolo di controller nel MVC pattern) e varie classi **view** che estendono `JFrame` (dal framework Java Swing) per la parte grafica dell'applicativo (contenute in `com.covid.view`).

### 2.1.1 Doctor

La classe **Doctor** rappresenta un medico, in particolare nel nostro contesto sarà colui che interagirà con l'applicativo ossia il medico direttore del reparto.

```
public class Doctor {
    private final String name;
    private final String surname;
    private final String specialization;

    public Doctor(String name, String surname, String specialization) {
        this.name = name;
        this.surname = surname;
        this.specialization = specialization;
    }

    public String getName() {
        return name;
    }

    public String getSurname() {
        return surname;
    }

    public String getSpecialization() {
        return specialization;
    }
}
```

Figura 2.1: Frammento di codice classe Doctor

### 2.1.2 CovidWard

La classe **CovidWard** rappresenta l'intero reparto medico. Esso conterrà una lista di pazienti con le rispettive cartelle cliniche e una lista di respiratori. Avrà anche un attributo di tipo **Doctor** che si riferisce al direttore del dipartimento.

```
public class CovidWard {
    private static Doctor director;
    private static List<CovidPatient> patients;
    private static List<MedicalRecord> records;
    private static LungVentilator[] ventilators;
    private static int occupiedBeds;
    public static final int MAXIMUM_NUM_BEDS = 10;
    public static final int NUM_VENTILATORS = 6;

    public Doctor getDirector() { return director; }

    public List<CovidPatient> getPatients() { return patients; }

    public List<MedicalRecord> getRecords() { return records; }

    public LungVentilator[] getVentilators() { return ventilators; }

    public int getOccupiedBeds() { return occupiedBeds; }

    public void addPatient(CovidPatient patient) throws NoBedsException {
        if (occupiedBeds < MAXIMUM_NUM_BEDS) {
            patients.add(patient);
            records.add(new MedicalRecord(patient));
            occupiedBeds++;
        } else {
            throw new NoBedsException();
        }
    }

    public void removePatient(CovidPatient patient) {
        if (patient != null) {
            int index = patients.indexOf(patient);
            patients.remove(patient);
        }
    }
}
```

Figura 2.2: Frammento di codice classe CovidWard

### 2.1.3 CovidPatient

La classe **CovidPatient** rappresenta un paziente affetto da Covid attraverso la sua anagrafica e alcuni dati sulla sua salute, in particolare una lista di patologie pregresse.

```
public class CovidPatient extends Observable {
    private final String name;
    private final String surname;
    private final String registeredResidence;
    private final int age;
    private int saturation;
    private boolean isPositive;
    private final List<Pathology> previousPathologies;

    private CovidPatient(CovidPatientBuilder builder) {
        name = builder.name;
        surname = builder.surname;
        registeredResidence = builder.registeredResidence;
        age = builder.age;
        isPositive = builder.isPositive;
        saturation = 0;
        previousPathologies = new ArrayList<>();
    }

    public String getName() { return name; }

    public String getSurname() { return surname; }

    public String getRegisteredResidence() { return registeredResidence; }

    public int getAge() { return age; }

    public List<Pathology> getPreviousPathologies() { return previousPathologies; }

    public int getSaturation() { return saturation; }

    public boolean isPositive() { return isPositive; }
}
```

**Figura 2.3:** Frammento di codice classe CovidPatient

### 2.1.4 Pathology

La classe **Pathology** rappresenta una patologia pregressa che può essere associata ad un paziente Covid. Oltre al tipo di patologia vi è la possibilità di specificare una breve descrizione della patologia stessa.

```
public class Pathology {
    private final String name;
    private final String description;

    public Pathology(String name, String description) {
        this.name = name;
        this.description = description;
    }

    public String getName() { return name; }

    public String getDescription() { return description; }
}
```

**Figura 2.4:** Frammento di codice classe Pathology

### 2.1.5 LungVentilator

La classe **LungVentilator** rappresenta un ventilatore polmonare che può essere assegnato ad un paziente in caso di bisogno da parte del medico. Possono esserci due tipi di ventilatori polmonari: a pressione positiva o a pressione negativa.

```
public class LungVentilator {
    private final LungVentilatorsTypologies typology;
    private CovidPatient patient;
    private boolean isBusy;

    public LungVentilator(LungVentilatorsTypologies typology) {
        this.typology = typology;
        patient = null;
        isBusy = false;
    }

    public boolean isBusy() { return isBusy; }

    public LungVentilatorsTypologies getTypology() { return typology; }

    public CovidPatient getPatient() { return patient; }

    public void setPatient(CovidPatient patient){
        this.patient = patient;
        isBusy = true;
    }

    public void free() {
        patient = null;
        isBusy = false;
    }
}
```

Figura 2.5: Frammento di codice classe LungVentilator

### 2.1.6 MedicalRecord

La classe **MedicalRecord** rappresenta la cartella clinica di un determinato paziente.

```
public class MedicalRecord implements Observer {
    private final CovidPatient patient;
    private final String medicalRecordID;
    private RecoveryRateStrategy strategy;

    public MedicalRecord(CovidPatient patient) {
        this.patient = patient;
        medicalRecordID = generateCode();
        patient.addObserver(⓪: this);
        strategy = new RecoveryRateNoPathologies();
    }

    private String generateCode() {
        String code = "";
        if (patient.getName().length() > 2)
            code += (patient.getName().substring(0, 3)).toUpperCase();
        if (patient.getSurname().length() > 2)
            code += (patient.getSurname().substring(0, 3)).toUpperCase();
        code += Integer.toString(patient.getAge());
        if (patient.getRegisteredResidence().length() > 2)
            code += (patient.getRegisteredResidence().substring(0, 3)).toUpperCase();
        return code;
    }

    public double getRecoveryRate() { return strategy.recoveryRate(patient); }

    public String getMedicalRecordID() { return medicalRecordID; }
}
```

Figura 2.6: Frammento di codice classe MedicalRecord

## 2.2 Design Patterns

Sono stati utilizzati alcuni design patterns per la realizzazione del progetto. In particolare i patterns utilizzati sono:

1. Singleton
2. Builder
3. Observer
4. Strategy
5. MVC

### 2.2.1 Singleton

Il pattern **Singleton** è un pattern comportamentale utilizzato per avere un'unica istanza di una determinata classe. Per realizzare ciò il costruttore viene dichiarato come privato e viene creato un metodo statico **getInstance** che restituirà un'unica istanza della classe. Nel progetto la classe **CovidWard** è stata realizzata come Singleton.

```
private CovidWard(Doctor director) {
    CovidWard.director = director;
    patients = new ArrayList<>();
    records = new ArrayList<>();
    ventilators = new LungVentilator[NUM_VENTILATORS];
    setUpVentilators();
    occupiedBeds = 0;
}

public static CovidWard getInstance(Doctor director) {
    if (instance == null) {
        instance = new CovidWard(director);
    }
    return instance;
}
```

**Figura 2.7:** Frammento di codice Singleton

## 2.2.2 Builder

Il pattern **Builder** è un pattern creazionale utilizzato per la creazione modulare di oggetti aventi numerosi attributi. Per realizzare ciò è stata dichiarata una **inner-class** chiamata **CovidPatientBuilder** delegata alla creazione di un paziente attraverso il metodo statico **build**.

```
public static class CovidPatientBuilder {
    private final String name;
    private final String surname;
    private String registeredResidence;
    private int age;
    private boolean isPositive;

    public CovidPatientBuilder(String name, String surname) {
        this.name = name;
        this.surname = surname;
    }

    public CovidPatientBuilder setRegisteredResidence(String registeredResidence) {
        this.registeredResidence = registeredResidence;
        return this;
    }

    public CovidPatientBuilder setAge(int age) {
        this.age = age;
        return this;
    }

    public CovidPatientBuilder setPositive(boolean positive) {
        isPositive = positive;
        return this;
    }

    public CovidPatient build() { return new CovidPatient( builder: this); }
}
```

Figura 2.8: Frammento di codice Builder

## 2.2.3 Observer

Il pattern **Observer** è un pattern comportamentale utilizzato quando uno o più oggetti (observers) devono monitorare un altro oggetto detto subject. Il subject ha al suo interno una lista degli observers che lo stanno "osservando" e sarà suo compito notificare essi ogni volta che cambierà stato. Per l'implementazione sono state utilizzate la classe **Observable** e l'interfaccia **Observer** entrambe contenute nel package **java.util**. In particolare nel progetto la classe Subject (che estenderà Observable) è la classe del paziente (**CovidPatient**) mentre la classe che funge da Observer è la cartella clinica del paziente (**MedicalRecord**) stesso.

```
@Override
public void update(Observable o, Object arg) {
    if (patient.getNumPathologies() > 0)
        setStrategy(new RecoveryRatePreviousPathologies());
    else
        setStrategy(new RecoveryRateNoPathologies());
}
```

Figura 2.9: Frammento di codice dell'update della classe MedicalRecord

## 2.2.4 Strategy

Il pattern **Strategy** è un pattern comportamentale utilizzato per avere un comportamento diverso di un metodo (incapsulato in un oggetto) a seconda della strategia scelta (mediante il metodo `setStrategy`). Nel nostro caso la cartella clinica (**MedicalRecord**) ha un campo di tipo **RecoveryRate** (interfaccia) e quando viene cambiato il numero di patologie di un paziente (notificato attraverso il pattern **Observer**) viene cambiata la strategia istanziando un oggetto di tipo **RecoveryRateNoPathologies** oppure **RecoveryRatePreviousPathologies**.

```
public interface RecoveryRateStrategy {
    public double recoveryRate(CovidPatient patient);
}
```

**Figura 2.10:** Frammento di codice dell'interfaccia **RecoveryRate**

```
public class RecoveryRateNoPathologies implements RecoveryRateStrategy {
    @Override
    public double recoveryRate(CovidPatient patient) {
        try {
            Scanner scanner = new Scanner(new File( pathname: "./data/recoveryRate.txt"));
            while (scanner.hasNextLine()) {
                if (Integer.parseInt(scanner.nextLine()) == patient.getAge())
                    return Double.parseDouble(scanner.nextLine());
                scanner.nextLine();
            }
        } catch (FileNotFoundException e) {
            System.out.println("File non trovato");
        }
        return 0;
    }
}
```

**Figura 2.11:** Frammento di codice della classe **RecoveryRateNoPathologies**

```
public class RecoveryRatePreviousPathologies implements RecoveryRateStrategy {
    @Override
    public double recoveryRate(CovidPatient patient) {
        double rate = 100;
        for (int i = 0; i < patient.getPreviousPathologies().size(); i++) {
            rate -= weightedByPathology(patient.getPreviousPathologies().get(i));
        }
        rate += weightedByAge(patient);
        return rate;
    }

    private double weightedByPathology(Pathology pathology) {
        try {
            Scanner scanner = new Scanner(new File( pathname: "./data/weightedByPathology.txt"));
            while (scanner.hasNextLine()) {
                if (scanner.nextLine().equalsIgnoreCase(pathology.getName()))
                    return Double.parseDouble(scanner.nextLine());
                scanner.nextLine();
            }
        } catch (FileNotFoundException e) {
            System.out.println("File non trovato");
        }
        return 0;
    }

    private double weightedByAge(CovidPatient patient) {
        try {
            Scanner scanner = new Scanner(new File( pathname: "./data/weightedByAge.txt"));
            while (scanner.hasNextLine()) {
                if (Integer.parseInt(scanner.nextLine()) == patient.getAge())
                    return Double.parseDouble(scanner.nextLine());
                scanner.nextLine();
            }
        }
    }
}
```

**Figura 2.12:** Frammento di codice della classe **RecoveryRatePreviousPathologies**

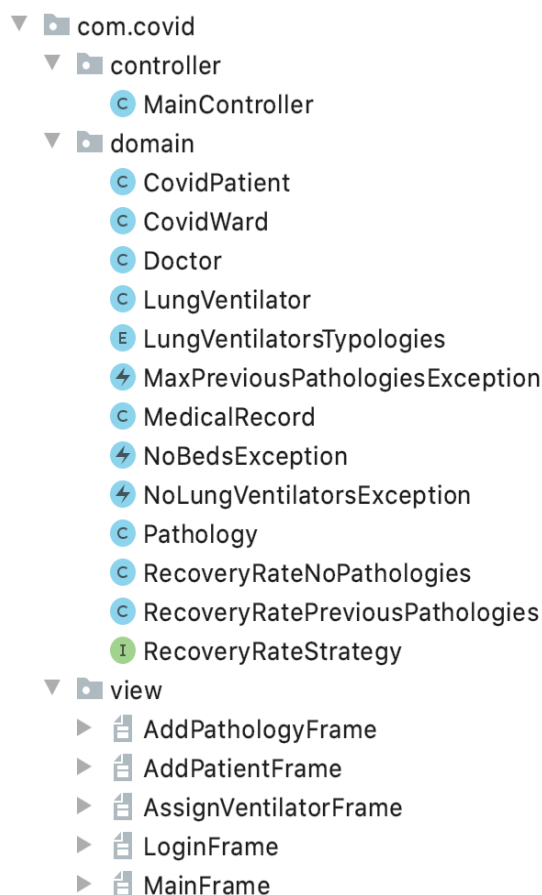


### 2.2.5 MVC

Il pattern architetturale Model-View-Controller ha lo scopo di separare la logica di dominio e di business di un programma dalla logica di presentazione.

Nell'applicativo in questione l'intero codice è stato suddiviso in 3 packages principali:

1. **Model:** il modello fornisce i metodi per accedere ai dati dell'applicazione.  
In questo caso il model coincide con le principali classi del progetto contenute nel package **com.covid.domain**.
2. **Controller:** svolge il ruolo da mediatore tra model e view. Riceve i comandi dall'utente e li attua modificando lo stato degli altri due componenti.  
In questo caso il controller coincide con la classe **MainController** nel package **com.covid.controller**.
3. **View:** ha il compito di visualizzare i dati del model e di interagire con l'utente.  
In questo caso tutte le classi riguardanti l'interfaccia grafica utente (GUI) sono contenute nel package **com.covid.view**.



**Figura 2.13:** Organizzazione del codice in packages

## 2.3 Ulteriori Dettagli Implementativi

- Per quanto riguarda la parte di grafica, come già anticipato, è stato utilizzato il framework **Java Swing** con cui sono state realizzate le classi che rappresentano i vari frames del programma (vedi 1.3) estendendo la classe base `JFrame`.
- Per le varie tipologie di respiratore polmonare è stato utilizzato un enum (**LungVentilatorsTipologies**) al fine di poterlo estendere a nuove tipologie in seguito.
- Per i dati sulla percentuale di recupero legata all'età e alle patologie pregresse, sono stati caricati i vari dati da dei file .txt contenuti nella cartella **data** all'interno della directory principale del progetto. Per la lettura dei suddetti files sono state utilizzate le classi **File** e **Scanner** contenute rispettivamente in **java.io** e **java.util**.

### 3.1 Unit Testing

Quando si parla di Unit Testing si intende la verifica di singole porzioni di codice, nel caso dell'Object-Oriented Programming (OOP) tipicamente saranno le singole classi o i singoli metodi.

Una volta individuate le varie sezioni di codice si potrà procedere con i test che vengono detti **test cases**.

Nel progetto è stato utilizzato il framework **JUnit** nella versione **5.0**.

In JUnit i test-case sono dei metodi anteposti dall'annotazione **@Test** (o eventualmente **@BeforeEach** oppure **@After**).

Per ogni classe principale del progetto sono state create le rispettive classi di Test contenenti i test case relativi ai metodi della classe principale da testare.

In ogni metodo di test vengono verificate delle asserzioni (**asserts**) elementari attraverso vari metodi offerti da JUnit stesso: **assertEquals**, **assertTrue**, **assertFalse**, **assertNull** etc.

Le classi di Test realizzate nel progetto sono:

1. **DoctorTest**
2. **CovidWardTest**
3. **CovidPatientTest**
4. **PathologyTest**
5. **LungVentilatorTest**
6. **MedicalRecordTest**
7. **RecoveryRateStrategyTest**

### 3.1.1 DoctorTest

```
class DoctorTest {
    Doctor doctor;

    @BeforeEach
    void setUp() {
        doctor = new Doctor( name: "Giovanni", surname: "Neri", specialization: "pneumologia");
    }

    @Test
    void getName() {
        assertEquals(doctor.getName(), actual: "Giovanni");
    }

    @Test
    void getSurname() {
        assertEquals(doctor.getSurname(), actual: "Neri");
    }

    @Test
    void getSpecialization() {
        assertEquals(doctor.getSpecialization(), actual: "pneumologia");
    }
}
```

Figura 3.1: Frammento della classe DoctorTest

### 3.1.2 CovidWardTest

```
class CovidWardTest {
    CovidWard ward;

    @BeforeEach
    void setUp() {
        ward = CovidWard.getInstance(new Doctor( name: "Giovanni", surname: "Neri", specialization: "pneumologia"));
    }

    @Test
    void getInstance() { assertEquals(ward, CovidWard.getInstance(new Doctor( name: "Giacomo", surname: "Bianchi", specialization: "pneumologia"))); }

    @Test
    void getDirector() {
        assertEquals(ward.getDirector().getName(), actual: "Giovanni");
        assertEquals(ward.getDirector().getSurname(), actual: "Neri");
        assertEquals(ward.getDirector().getSpecialization(), actual: "pneumologia");
    }

    @Test
    void getOccupiedBeds() throws NoBedsException {
        ward.removeAllPatients();
        ward.addPatient(new CovidPatient.CovidPatientBuilder( name: "Gianni", surname: "Mori").
            setRegisteredResidence("Siena").setAge(78).setPositive(true).build());
        assertEquals(ward.getOccupiedBeds(), actual: 1);
    }

    @Test
    void addPatient() throws NoBedsException {
        ward.removeAllPatients();
        ward.addPatient(new CovidPatient.CovidPatientBuilder( name: "Gianni", surname: "Mori").
            setRegisteredResidence("Siena").setAge(78).setPositive(true).build());
        assertEquals(ward.getPatients().get(0).getName(), actual: "Gianni");
        assertEquals(ward.getPatients().get(0).getSurname(), actual: "Mori");
    }
}
```

Figura 3.2: Frammento della classe CovidWardTest

Nella classe CovidWardTest sono stati realizzati anche alcuni test che verificassero il corretto lancio delle eccezioni attraverso il metodo **assertThrows**.

```
@Test
void testNoBedsExpectedException() throws NoBedsException {
    ward.removeAllPatients();
    ward.addPatient(new CovidPatient.CovidPatientBuilder( name: "Gianni", surname: "Mori").
        setRegisteredResidence("Siena").setAge(78).setPositive(true).build());
    ward.addPatient(new CovidPatient.CovidPatientBuilder( name: "Bruno", surname: "Bruni").
        setRegisteredResidence("Prato").setAge(45).setPositive(true).build());
    ward.addPatient(new CovidPatient.CovidPatientBuilder( name: "Valeria", surname: "Neri").
        setRegisteredResidence("Firenze").setAge(87).setPositive(true).build());
    ward.addPatient(new CovidPatient.CovidPatientBuilder( name: "Serena", surname: "Bianchi").
        setRegisteredResidence("Grosseto").setAge(34).setPositive(true).build());
    ward.addPatient(new CovidPatient.CovidPatientBuilder( name: "Maurizio", surname: "Neri").
        setRegisteredResidence("Massa").setAge(49).setPositive(true).build());
    ward.addPatient(new CovidPatient.CovidPatientBuilder( name: "Alberto", surname: "Berti").
        setRegisteredResidence("Lucca").setAge(47).setPositive(true).build());
    ward.addPatient(new CovidPatient.CovidPatientBuilder( name: "Giorgia", surname: "Rossi").
        setRegisteredResidence("Siena").setAge(98).setPositive(true).build());
    ward.addPatient(new CovidPatient.CovidPatientBuilder( name: "Alberto", surname: "Franchi").
        setRegisteredResidence("Livorno").setAge(92).setPositive(true).build());
    ward.addPatient(new CovidPatient.CovidPatientBuilder( name: "Maria", surname: "Innocenti").
        setRegisteredResidence("Pistoia").setAge(42).setPositive(true).build());
    ward.addPatient(new CovidPatient.CovidPatientBuilder( name: "Paola", surname: "Neri").
        setRegisteredResidence("Firenze").setAge(57).setPositive(true).build());
    assertThrows(NoBedsException.class, () -> ward.addPatient(new CovidPatient.CovidPatientBuilder( name: "Luisa",
        setRegisteredResidence("Prato").setAge(82).setPositive(true).build());
    );
}

@Test
void testNoVentilatorsExpectedException() throws NoBedsException, NoLungVentilatorsException {
    ward.removeAllPatients();
    CovidPatient op1 = new CovidPatient.CovidPatientBuilder( name: "Gianni", surname: "Mori").
        setRegisteredResidence("Siena").setAge(78).setPositive(true).build();
}
```

Figura 3.3: Test sul lancio di un'eccezione

### 3.1.3 CovidPatientTest

Testo il corretto funzionamento del Builder.

```
class CovidPatientTest {
    CovidPatient patient;

    @BeforeEach
    void setUp() {
        patient = new CovidPatient.CovidPatientBuilder( name: "Mario", surname: "Rossi").
            setRegisteredResidence("Firenze").setAge(75).setPositive(true).build();
    }

    @Test
    void getName() { assertEquals(patient.getName(), actual: "Mario"); }

    @Test
    void getSurname() { assertEquals(patient.getSurname(), actual: "Rossi"); }

    @Test
    void getRegisteredResidence() { assertEquals(patient.getRegisteredResidence(), actual: "Firenze"); }

    @Test
    void getAge() { assertEquals(patient.getAge(), actual: 75); }

    @Test
    void isPositive() { assertTrue(patient.isPositive()); }

    @Test
    void setPositive() {
        patient.setPositive(false);
        assertFalse(patient.isPositive());
    }
}
```

**Figura 3.4:** Frammento della classe CovidPatientTest

### 3.1.4 PathologyTest

```
class PathologyTest {
    Pathology pathology;

    @BeforeEach
    void setUp() {
        pathology = new Pathology( name: "Ipertensione", description: "prova");
    }

    @Test
    void getName() { assertEquals(pathology.getName(), actual: "Ipertensione"); }

    @Test
    void getDescription() { assertEquals(pathology.getDescription(), actual: "prova"); }
}
```

**Figura 3.5:** Frammento della classe PathologyTest

### 3.1.5 LungVentilatorTest

```
class LungVentilatorTest {
    LungVentilator ventilator;

    @BeforeEach
    void setUp() {
        ventilator = new LungVentilator(LungVentilatorsTypologies.POSITIVE_PRESSURE_VENTILATOR);
    }

    @Test
    void isBusy() {
        ventilator.setPatient(new CovidPatient.CovidPatientBuilder( name: "Mario", surname: "Meri").
            setRegisteredResidence("Firenze").setAge(78).setPositive(true).build());
        assertTrue(ventilator.isBusy());
    }

    @Test
    void getTypology() {
        assertEquals(ventilator.getTypology(), LungVentilatorsTypologies.POSITIVE_PRESSURE_VENTILATOR);
    }

    @Test
    void setPatient() {
        ventilator.setPatient(new CovidPatient.CovidPatientBuilder( name: "Gianni", surname: "Rossi").
            setRegisteredResidence("Firenze").setAge(85).setPositive(true).build());
        assertEquals(ventilator.getPatient().getName(), actual: "Gianni");
        assertEquals(ventilator.getPatient().getSurname(), actual: "Rossi");
        assertEquals(ventilator.getPatient().getRegisteredResidence(), actual: "Firenze");
        assertEquals(ventilator.getPatient().getAge(), actual: 85);
        assertTrue(ventilator.getPatient().isPositive());
    }
}
```

**Figura 3.6:** Frammento della classe LungVentilatorTest

### 3.1.6 MedicalRecordTest

Testo il cambio di strategia se aggiungo una nuova patologia.

```
class LungVentilatorTest {
    LungVentilator ventilator;

    @BeforeEach
    void setUp() {
        ventilator = new LungVentilator(LungVentilatorsTypologies.POSITIVE_PRESSURE_VENTILATOR);
    }

    @Test
    void isBusy() {
        ventilator.setPatient(new CovidPatient.CovidPatientBuilder( name: "Mario", surname: "Neri").
            setRegisteredResidence("Firenze").setAge(78).setPositive(true).build());
        assertTrue(ventilator.isBusy());
    }

    @Test
    void getTypology() {
        assertEquals(ventilator.getTypology(), LungVentilatorsTypologies.POSITIVE_PRESSURE_VENTILATOR);
    }

    @Test
    void setPatient() {
        ventilator.setPatient(new CovidPatient.CovidPatientBuilder( name: "Gianni", surname: "Rossi").
            setRegisteredResidence("Firenze").setAge(85).setPositive(true).build());
        assertEquals(ventilator.getPatient().getName(), actual: "Gianni");
        assertEquals(ventilator.getPatient().getSurname(), actual: "Rossi");
        assertEquals(ventilator.getPatient().getRegisteredResidence(), actual: "Firenze");
        assertEquals(ventilator.getPatient().getAge(), actual: 85);
        assertTrue(ventilator.getPatient().isPositive());
    }
}
```

**Figura 3.7:** Frammento della classe MedicalRecordTest

### 3.1.7 RecoveryRateStrategyTest

Testo il range dei valori della percentuale di recupero.

```
class RecoveryRateNoPathologiesTest {
    RecoveryRateStrategy strategy;

    @BeforeEach
    void setUp() { strategy = new RecoveryRateNoPathologies(); }

    @Test
    void recoveryRate() {
        CovidPatient patient = new CovidPatient.CovidPatientBuilder( name: "Franco", surname: "Neri").
            setRegisteredResidence("Firenze").setAge((int) (Math.random() * 181)).setPositive(true).build();
        assertTrue( condition: 0 <= strategy.recoveryRate(patient) && strategy.recoveryRate(patient) <= 180);
    }
}
```

**Figura 3.8:** Frammento della classe RecoveryRateNoPathologiesTest

## 3.2 Sequence Diagram

Qui di seguito un Sequence Diagram che simula un possibile flusso d'esecuzione del programma con l'inserimento di un unico paziente.

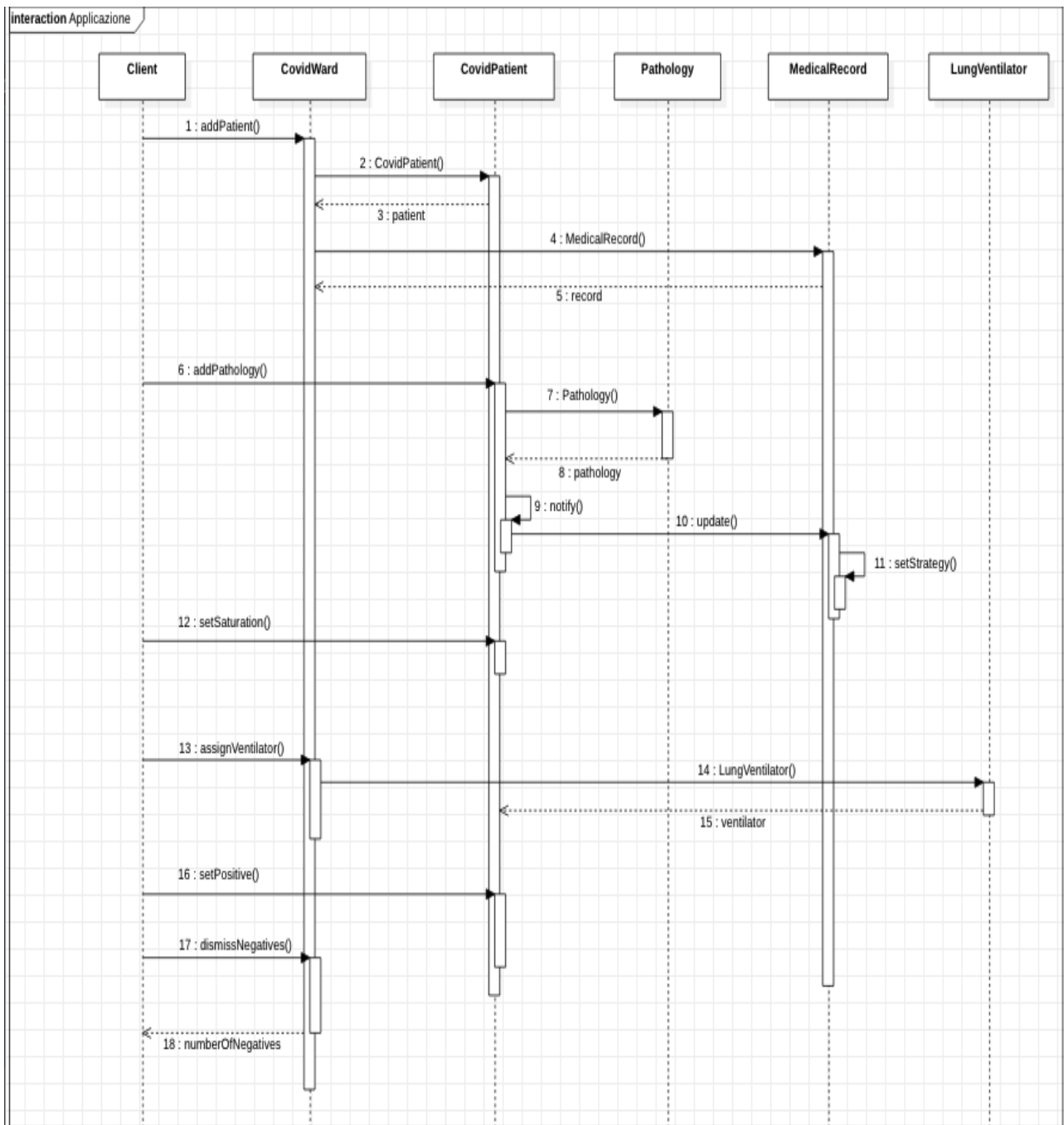


Figura 3.9: Sequence Diagram