

UNIVERSITÀ DEGLI STUDI DI FIRENZE



Dipartimento di Ingegneria

Corso di Laurea in Ingegneria Informatica

Elaborato Ingegneria del Software

**Applicativo java che simula la gestione delle
prenotazioni online dei voli di un aeroporto**

LEONARDO GORI

27 agosto 2020

Anno Accademico 2019/2020

Indice

1	Introduzione	3
2	Progettazione	5
2.1	Casi d'uso	5
2.2	Class Diagram	6
2.3	Sequence Diagram	7
3	Implementazione	8
3.1	Classi ed Interfacce	8
3.1.1	Airport	9
3.1.2	Airplane	10
3.1.3	Flight	11
3.1.4	FlightTable	12
3.1.5	Customer	13
3.1.6	Reservation	14
3.1.7	RealReservableFlightArchive	16
3.2	Design Patterns	17
3.2.1	Singleton	17
3.2.2	Observer	18
3.2.3	Protection Proxy	20
4	Testing ed Esecuzione	21
4.1	Unit Testing	21

4.1.1	AirplanetTest	22
4.1.2	FlightTest	23
4.1.3	CustomerTest	24
4.1.4	ReservableProxyTest	25
4.1.5	ObserverTest	26

Capitolo 1

Introduzione

L'elaborato consiste nella realizzazione di un applicativo, scritto in linguaggio Java, che simuli in modo semplicistico la gestione delle prenotazioni online dei voli di un complesso di aeroporti. Attraverso l'applicazione l'utente può vedere tutti i voli disponibili del complesso di aeroporti, è necessario registrare un profilo con username e password per poter associargli dei voli. Dopo essersi registrato egli può prenotare voli, visualizzarli e disdire prenotazioni. Per semplicità di progettazione il complesso di aeroporti e i relativi voli è precostituito, e alla fine dell'interazione degli utenti col programma, viene volontariamente annullato un volo e, nel caso fosse stato prenotato da uno o più utenti, a questi ultimi verrà notificato. 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 e realizzata una logica di dominio in prospettiva di specifica/implementazione attraverso un class diagram in UML.

Nella realizzazione del progetto sono stati utilizzati i seguenti Design Patterns:

- Per la realizzazione della classe RealReservableFlightArchive, rappresentante l'archivio dei voli prenotabili di tutti gli aeroporti, è stato utilizzato il design pattern Singleton, il quale si occupa di garantire la creazione e l'accesso di un'unica istanza della classe sul quale viene applicato.
- Per permettere la prenotazione o dismissione di voli e la visualizzazione dei voli prenotati (considerati dati sensibili) da parte dei soli utenti registrati, è stato utilizzato il design pattern Protection Proxy, implementato attraverso la classe ReservableProxy, la quale si interfaccia tra le classi Customer e RealReservableArchive.
- Per permettere la notifica automatica della cancellazione di un volo prenotato dagli utenti è stato utilizzato il design pattern Observer. Protagoniste di tale pattern sono le classi Reservation, che racchiude i voli prenotati da un singolo utente, la quale estende la classe Observable e FlightTable, contenente i voli prenotabili di un singolo aeroporto, la quale implementa l'interfaccia Observer.

Capitolo 2

Progettazione

2.1 Casi d'uso

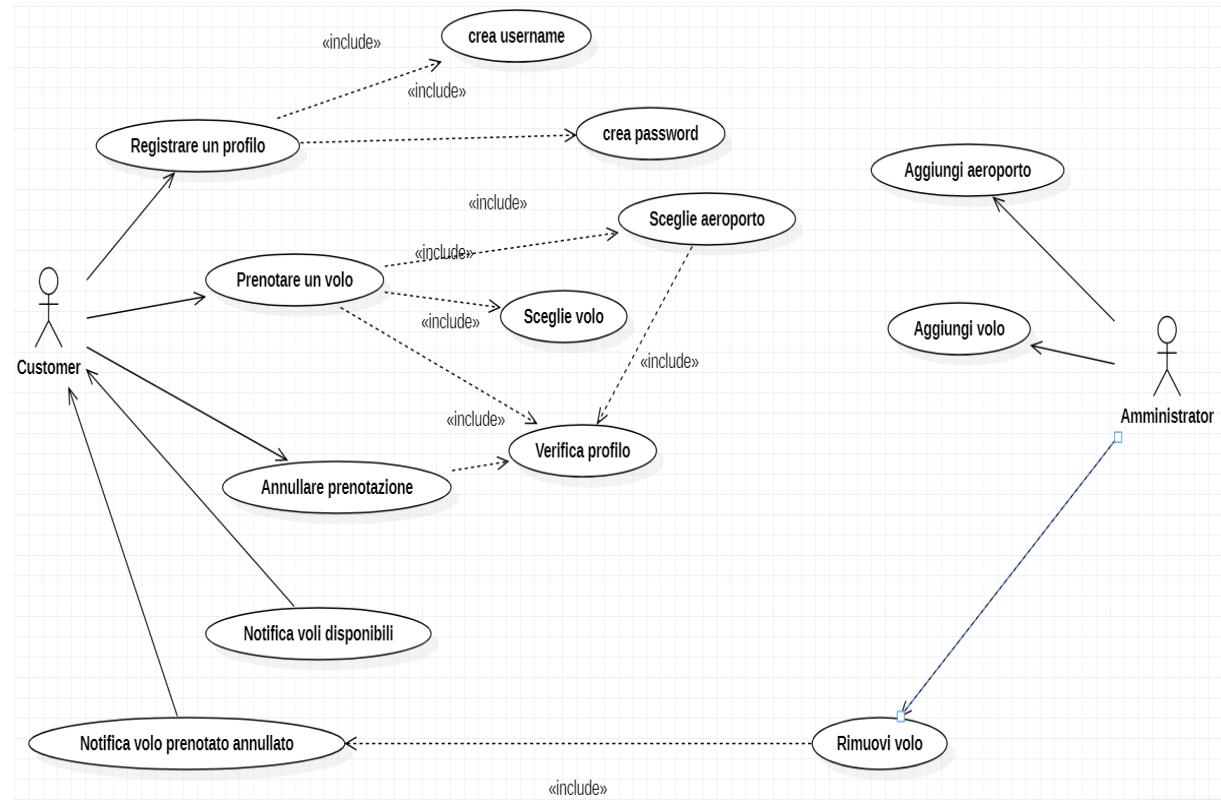


Figura 2.1: Use Case Diagram

2.2 Class Diagram

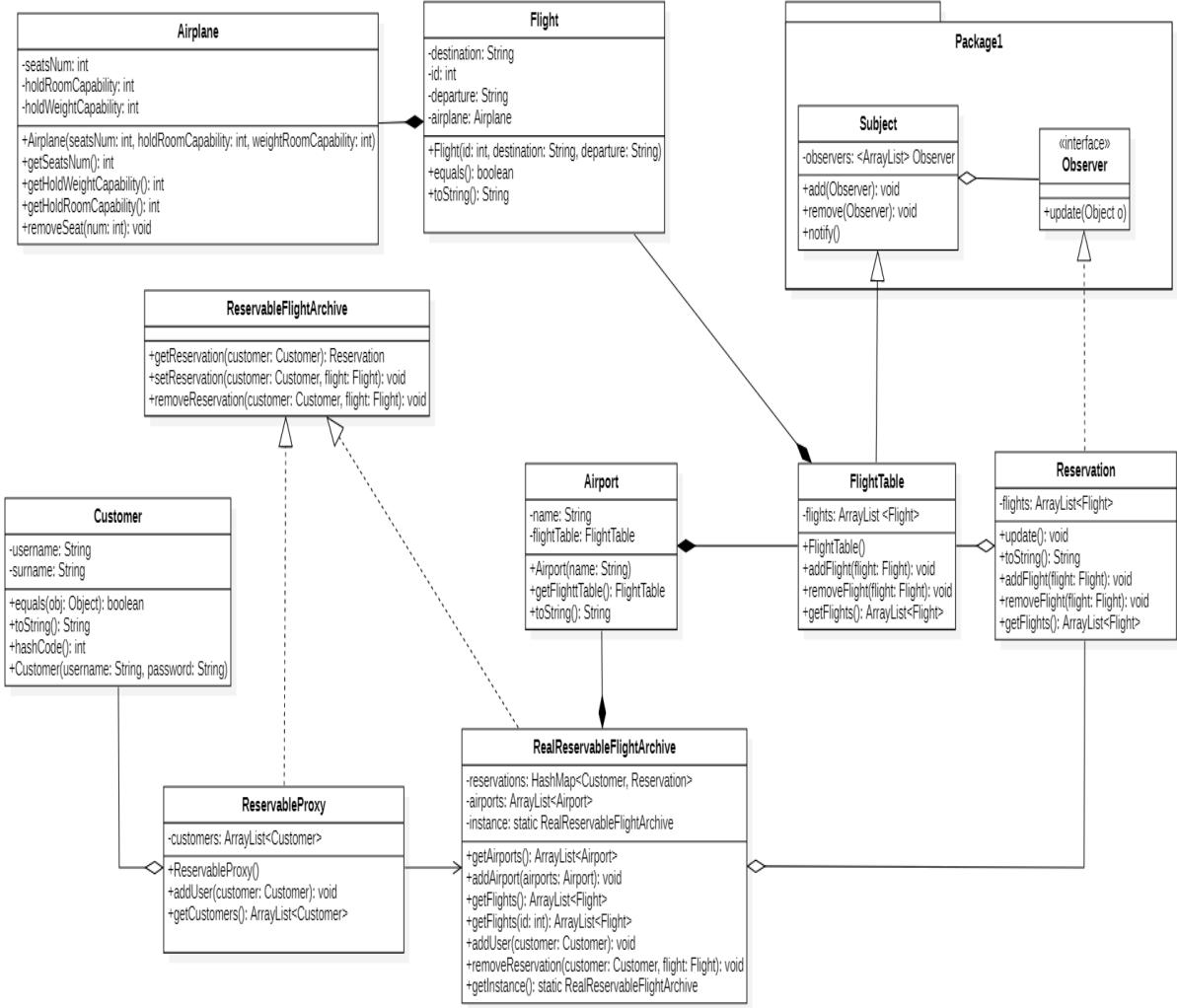


Figura 2.2: Class Diagram

2.3 Sequence Diagram

Qui di seguito un Sequence Diagram che simula un possibile flusso d'esecuzione del programma con la prenotazione di un volo da parte di un utente registrato.

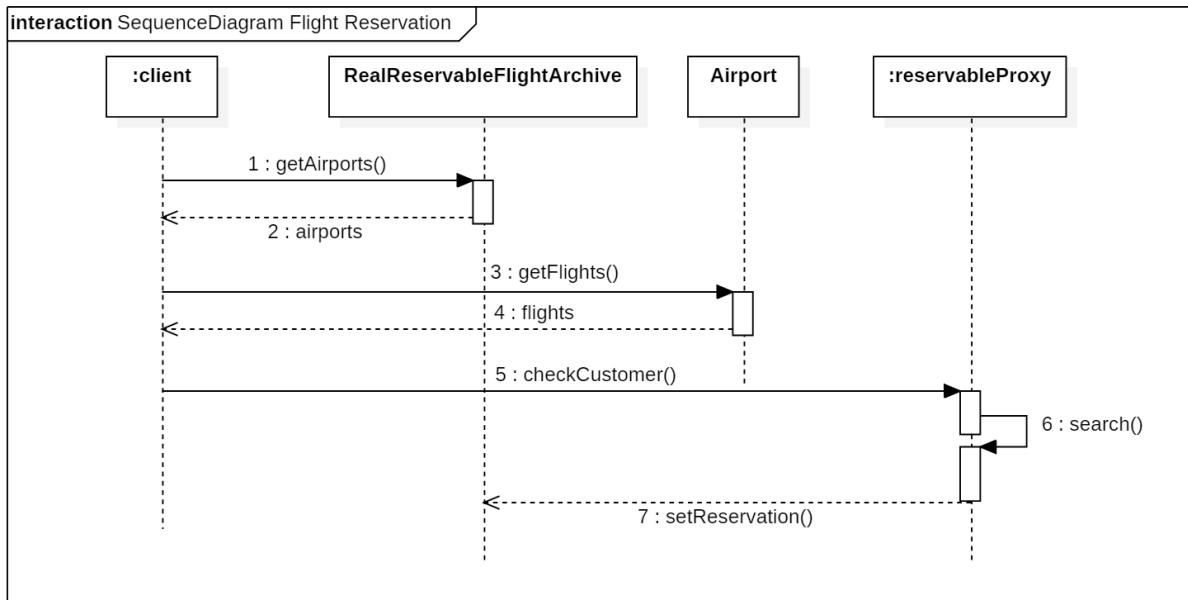


Figura 2.3: Sequence Diagram

Capitolo 3

Implementazione

Durante la fase di implementazione sono state definite e realizzate classi, interfacce e utilizzata la libreria standard contenente il package `java.util`, nonchè i design patterns definiti in precedenza.

3.1 Classi ed Interfacce

Le principali classi utilizzate sono:

1. Airport
2. Airplane
3. Flight
4. FlightTable
5. Customer
6. Reservation
7. ReservableFlightArchive
 - RealReservableFlightArchive

3.1.1 Airport

La classe Airport rappresenta un aeroporto contenente una tabella dei voli disponibili:

```
public class Airport {  
    FlightTable flightTable;  
    String name;  
  
    public String getName() { return name; }  
  
    public Airport(String name) {  
        this.name = name;  
        flightTable = new FlightTable();  
    }  
  
    public FlightTable getFlightTable() { return flightTable; }  
  
    @Override  
    public String toString() { return name; }  
}
```

Figura 3.1: Codice Java della classe Airport

3.1.2 Airplane

La classe Airplane rappresenta un aeroplano, più istanze di aeroplani sono contenute all'interno di un aeroporto:

```
public class Airplane {  
  
    private int seatsNum;  
    private int holdRoomCapability;  
    private int holdWeightCapability;  
    private int availableSeats;  
  
    public int getAvailableSeats() { return availableSeats; }  
  
    public Airplane(int seatsNum, int holdRoomCabability, int holdWeightCapability) {  
        this.seatsNum = this.availableSeats = seatsNum;  
        this.holdRoomCapability = holdRoomCabability;  
        this.holdWeightCapability = holdWeightCapability;  
    }  
  
    public int getSeatsNum() { return seatsNum; }  
  
    public int getHoldRoomCapability() { return holdRoomCapability; }  
  
    public int getHoldWeightCapability() { return holdWeightCapability; }  
  
    public void removeSeat() {  
        availableSeats --;  
        holdWeightCapability -= (holdWeightCapability/seatsNum);  
        holdRoomCapability -= (holdRoomCapability/seatsNum);  
    }  
}
```

Figura 3.2: Codice Java della classe Airplane

3.1.3 Flight

La classe Flight rappresenta un volo prenotabile, a ogni volo è associato un aereo. Per semplicità di realizzazione l'aereo relativo al volo viene creato al momento della creazione di quest'ultimo.

```
class Flight {  
  
    private String departure;  
    private String destination;  
    private Airplane airplane;  
    private int id;  
  
    @Override  
    public boolean equals(Object obj) {  
        if(obj instanceof Flight){  
            Flight flight = (Flight) obj;  
            if(flight != null && this.destination.equals(flight.destination) && this.id == flight.id){  
                return true;  
            }  
        }  
        return false;  
    }  
  
    @Override  
    public String toString() {  
        return "Volo n." + id + " con partenza da " + departure + " con destinazione a " + destination;  
    }  
  
    public Flight(int id, String destination, String departure) {  
        this.destination = destination;  
        this.departure = departure;  
        this.id = id;  
        this.airplane = new Airplane(seatsNum: 100, holdRoomCabability: 200, holdWeightCapability: 300);  
    }  
  
    public String getDestination() { return destination; }  
  
    public void setDestination(String destination) { this.destination = destination; }  
  
    public void removeAvailableSeat() { airplane.removeSeat(); }  
  
    public Airplane getAirplane() { return airplane; }  
}
```

Figura 3.3: Codice Java della classe Flight

3.1.4 FlightTable

La classe FlightTable rappresenta una tabella contenente tutti i voli prenotabili di un aeroporto.

All'interno del progetto essa estende la classe Observable e ne eredita i principali metodi usati per la notifica del cambiamento di stato quali setChanged() e notifyObservers():

```
import java.util.ArrayList;
import java.util.Observable;

public class FlightTable extends Observable{

    ArrayList<Flight> flights;

    public FlightTable() { flights = new ArrayList(); }

    public void addFlight(Flight flight) { flights.add(flight); }

    public void removeFlight(Flight flight) {
        flights.remove(flight);
        setChanged();
        notifyObservers(flight);
    }

    public ArrayList<Flight> getFlights() { return flights; }
}
```

Figura 3.4: Codice Java della classe FlightTable

3.1.5 Customer

La classe Customer rappresenta un utente che ha intenzione di prenotare uno o più voli, e che si interfaccia con l'applicazione. Per fare ciò è necessario che si registri con username e password.

```
public class Customer {  
  
    private String username;  
    private String password;  
  
    @Override  
    public boolean equals(Object obj) {  
        if(obj instanceof Customer){  
            Customer customer = (Customer) obj;  
            if(customer != null && this.username.equals(customer.username) && this.password.equals(customer.password)){  
                return true;  
            }  
        }  
        return false;  
    }  
  
    @Override  
    public String toString() { return username; }  
  
    public int hashCode()  
    {  
        String key = username+password;  
        int hash = (int)key.charAt(0);  
        return hash;  
    }  
  
    public Customer(String username, String password){  
        this.username = username;  
        this.password = password;  
    }  
  
    public String getUsername() { return username; }  
  
    public String getPassword() { return password; }  
}
```

Figura 3.5: Codice Java della classe Customer

3.1.6 Reservation

La classe Reservation raccoglie tutti i voli prenotati da un singolo utente senza il bisogno di ricorrere a un riferimento alla classe Customer. All'interno del progetto essa implementa l'interfaccia Observer e il relativo metodo update(). Inoltre, essendo considerato un dato sensibile, è mantenuta protetta grazie all'utilizzo del design pattern Protection Proxy:

```
import java.util.ArrayList;
import java.util.Observable;
import java.util.Observer;

public class Reservation implements Observer {

    private ArrayList<Flight> flights = new ArrayList<>();

    @Override
    public boolean equals(Object obj) {
        if(obj instanceof Reservation){
            Reservation reservation = (Reservation) obj;
            if(reservation != null && this.flights.equals(reservation.flights)){
                return true;
            }
        }
        return false;
    }

    @Override
    public void update(Observable o, Object arg) {
        Flight flight = (Flight) arg;
        if(flights.contains(flight)) {
            flights.remove(flight);
            System.out.println("Il " + flight + " è stato annullato");
        }
    }

    @Override
    public String toString() { return "Hai prenotato per i seguenti voli:\n" + flights; }

    public void addFlight(Flight flight) {
        if(flights.contains(flight))
            System.out.print("Il volo è già stato prenotato da questo utente");
        else {
            flights.add(flight);
            flights.get(flights.indexOf(flight)).removeAvailableSeat();
        }
    }

    public void removeFlight(Flight flight) {
        if(!flights.contains(flight))
            System.out.print("Il volo selezionato non è stato ancora prenotato");
        else
            flights.remove(flight);
    }

    public ArrayList<Flight> getFlights() { return flights; }
}
```

Figura 3.6: Codice Java della classe Reservation

ReservableFlightArchive

L’interfaccia ReservableFlightArchive raccoglie i metodi necessari per permettere la prenotazione, la rimozione e l’accesso ai voli prenotati, considerati contenuti sensibili:

```
public interface ReservableFlightArchive {  
  
    Reservation getReservation(Customer customer);  
    void setReservation(Customer customer, Flight flight);  
    void removeReservation(Customer customer, Flight flight);  
}
```

Figura 3.7: Codice Java dell’interfaccia ReservableFlightArchive

3.1.7 RealReservableFlightArchive

La classe RealReservableFlightArchive implementa l'interfaccia ReservableFlightArchive e i relativi metodi, aggiungendo inoltre funzioni per l'accesso ai voli prenotabili, nonchè ai relativi aeroporti e aeroplani coinvolti.

```
import java.util.*;

public class RealReservableFlightArchive implements ReservableFlightArchive {

    private HashMap<Customer, Reservation> reservations;
    private ArrayList<Airport> airports;
    private static RealReservableFlightArchive instance;

    private RealReservableFlightArchive() {
        reservations = new HashMap<>();
        airports = new ArrayList<>();
    }

    public static RealReservableFlightArchive getInstance() {
        if(instance == null)
            instance = new RealReservableFlightArchive();
        return instance;
    }

    public void addAirport(Airport airport) {
        airports.add(airport);
    }

    public ArrayList<Airport> getAirports() {
        return airports;
    }

    public ArrayList<Flight> getFlights() {
        ArrayList<Flight> flights = new ArrayList();
        for(Airport airport : airports)
            flights.addAll(airport.getFlightTable().getFlights());
        return flights;
    }

    public ArrayList<Flight> getFlights(int id) {
        ArrayList<Flight> flights = new ArrayList();
        flights.addAll(airports.get(id).getFlightTable().getFlights());
        return flights;
    }

    @Override
    public Reservation getReservation(Customer customer) {
        return reservations.get(customer);
    }

    @Override
    public void setReservation(Customer customer, Flight flight) {
        reservations.get(customer).addFlight(flight);
        for(Airport airport : airports) {
            if(airport.getFlightTable().getFlights().contains(flight))
                airport.getFlightTable().addObserver(reservations.get(customer));
        }
    }

    public void addUser(Customer customer) {
        reservations.put(customer, new Reservation());
    }

    public void removeReservation(Customer customer, Flight flight) {
        reservations.get(customer).removeFlight(flight);
    }
}
```

Figura 3.8: Codice Java della classe RealReservableFlightArchive

3.2 Design Patterns

Nella realizzazione del progetto sono stati utilizzati i seguenti Design Patterns:

- Singleton
- Observer
- Protection Proxy

3.2.1 Singleton

Il Singleton è un Design Pattern di tipo creazionale che si occupa di garantire la creazione e l'accesso globale di una sola istanza di una classe. In questo progetto è stato utilizzato per la classe RealReservableFlightArchive poichè rispecchia l'unica fonte di accesso ai dati (sensibili e non) dei voli e dei relativi aeroporti e aeroplani.

```
import java.util.*;

public class RealReservableFlightArchive implements ReservableFlightArchive {

    private HashMap<Customer, Reservation> reservations;
    private ArrayList<Airport> airports;
    private static RealReservableFlightArchive instance;

    private RealReservableFlightArchive() {
        reservations = new HashMap<>();
        airports = new ArrayList<>();
    }

    public static RealReservableFlightArchive getInstance() {
        if(instance == null)
            instance = new RealReservableFlightArchive();
        return instance;
    }
}
```

Figura 3.9: Frammento di codice della classe RealReservableClassArchive responsabile del Singleton Pattern

3.2.2 Observer

L'Observer è un Design Pattern di tipo comportamentale che si occupa di permettere a più istanze di una classe, detta Observer, di monitorare i cambiamenti di stato di un'istanza di un'altra classe, detta Subject, in modo automatico. Come accennato precedentemente, nel progetto le classi protagoniste di tale Pattern sono Reservation che svolgono il ruolo di Observers della classe FlightTable che svolge il ruolo di Subject. L'obiettivo di tale struttura era garantire la notifica automatica a tutti gli utenti dell'annullamento di un volo in un momento successivo alla prenotazione. La classe Subject generalmente raccoglie una lista di istanze della classe Observer tramite aggregazione. A ogni cambiamento di stato Subject notifica ai suoi Observers il cambiamento di stato e la condivisione dei dati modificati a quest' ultimi può essere applicata in modalità Push o Pull. In questo progetto è stata prediletta la modalità Push in quanto non era definito a priori quali voli potessero essere annullati, e a quali utenti notificare l' evento.

```
import java.util.ArrayList;
import java.util.Observable;

public class FlightTable extends Observable{
    ArrayList<Flight> flights;

    public void removeFlight(Flight flight) {
        flights.remove(flight);
        setChanged();
        notifyObservers(flight);
    }
}
```

Figura 3.10: Frammento di codice della classe FlightTable che svolge il ruolo di Subject, notificando in modalità push il volo rimosso dalla lista dei voli prenotabili ai suoi Observers

```
import java.util.ArrayList;
import java.util.Observable;
import java.util.Observer;

public class Reservation implements Observer {

    private ArrayList<Flight> flights = new ArrayList<>();

    @Override
    public void update(Observable o, Object arg) {
        Flight flight = (Flight) arg;
        if(flights.contains(flight)) {
            flights.remove(flight);
            System.out.println("Il " + flight + " è stato annullato");
        }
    }
}
```

Figura 3.11: Frammento di codice della classe Reservation che svolge il ruolo di Observer, la quale riceve il volo annullato e rimuove la prenotazione dalla lista dei voli prenotati

3.2.3 Protection Proxy

Il Protection Proxy è un Design Pattern di tipo strutturale che si occupa di fornire un controllo sull'accesso a un'istanza di una classe remota. All'interno del progetto è stato utilizzato per permettere la prenotazione dei voli, nonchè l'annullamento e la visualizzazione dei voli prenotati, proteggendoli tramite l'immissione di campi obbligatori quali username e password per verificare l'identità dell'utente. La classe protagonista di tale struttura è ReservableProxy, la quale estende l'interfaccia ReservableFlightArchive e controlla l'accesso ai metodi della classe RealReservableClassArchive.

```
import java.util.ArrayList;

public class ReservableProxy implements ReservableFlightArchive{

    private ArrayList <Customer> customers;

    public ReservableProxy(){
        customers = new ArrayList<>();
    }

    public void addUser(Customer customer){
        if(customers.contains(customer))
            System.out.println("password non disponibile");
        else {
            customers.add(customer);
            RealReservableFlightArchive.getInstance().addUser(customer);
        }
    }

    @Override
    public Reservation getReservation(Customer customer) {
        if(customers.contains(customer))
            return RealReservableFlightArchive.getInstance().getReservation(customer);
        else {
            System.out.println("utente non riconosciuto, per poter prenotare, annullare un volo o visualizzare le prenotazioni è necessario registrarsi");
            return null;
        }
    }

    @Override
    public void setReservation(Customer customer, Flight flight) {
        if(customers.contains(customer))
            RealReservableFlightArchive.getInstance().setReservation(customer, flight);
        else {
            System.out.println("utente non riconosciuto, per poter prenotare o annullare un volo o visualizzare le prenotazioni è necessario registrarsi");
        }
    }

    public void removeReservation(Customer customer, Flight flight) {
        if(customers.contains(customer))
            RealReservableFlightArchive.getInstance().removeReservation(customer, flight);
        else {
            System.out.println("utente non riconosciuto, per poter prenotare o annullare un volo o visualizzare le prenotazioni è necessario registrarsi");
        }
    }

    public ArrayList<Customer> getCustomers() { return customers; }
}
```

Figura 3.12: Frammento di codice della classe ReservableProxy che verifica l'identità dell'utente prima di permettere l'accesso ai dati sensibili

Capitolo 4

Testing ed Esecuzione

4.1 Unit Testing

Per i test è stato utilizzato il framework di unit testing JUnit 5. E' stato effettuato il test del funzionamento delle varie classi e in particolare del funzionamento dei vari Design Patterns.

4.1.1 AirplanetTest

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.assertEquals;

class AirplaneTest {

    private Airplane airplane;

    @BeforeEach
    void setUp() {
        airplane = new Airplane( seatsNum: 100, holdRoomCabability: 200, holdWeightCapability: 300);
    }

    @Test
    void getSeatsNum() {
        assertEquals(airplane.getSeatsNum(), actual: 100);
    }

    @Test
    void getAvailableSeats() {
        assertEquals(airplane.getAvailableSeats(), actual: 100);
    }

    @org.junit.jupiter.api.Test
    void getHoldRoomCapability() {
        assertEquals(airplane.getHoldRoomCapability(), actual: 200);
    }

    @org.junit.jupiter.api.Test
    void getHoldWeightCapability() {
        assertEquals(airplane.getHoldWeightCapability(), actual: 300);
    }

    @org.junit.jupiter.api.Test
    void removeSeat() {
        airplane.removeSeat();
        assertEquals(airplane.getAvailableSeats(), actual: 99);
    }
}
```

Figura 4.1: Codice java della classe AirplaneTest

4.1.2 FlightTest

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class FlightTest {

    private Flight flight;

    @BeforeEach
    void setUp() {
        flight = new Flight(id: 2302, destination: "Milano", departure: "Roma");
    }

    @Test
    void testEquals() {
        assertEquals(flight, new Flight(id: 2302, destination: "Milano", departure: "Roma"));
    }

    @Test
    void getDestination() {
        assertEquals(flight.getDestination(), actual: "Milano");
    }

    @Test
    void removeAvailableSeat() {
        flight.removeAvailableSeat();
        assertEquals(expected: flight.getAirplane().getSeatsNum() - 1, flight.getAirplane().getAvailableSeats());
    }
}
```

Figura 4.2: Codice java della classe FlightTest

4.1.3 CustomerTest

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class CustomerTest {

    private Customer customer;

    @BeforeEach
    void setUp() {
        customer = new Customer( username: "paolo", password: "password");
    }

    @Test
    void testEquals() {
        assertEquals(customer, new Customer( username: "paolo", password: "password"));
    }

    @Test
    void getUsername() {
        assertEquals(customer.getUsername(), actual: "paolo");
    }

    @Test
    void getPassword() {
        assertEquals(customer.getPassword(), actual: "password");
    }
}
```

Figura 4.3: Codice java della classe CustomerTest

4.1.4 ReservableProxyTest

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import java.util.ArrayList;
import static org.junit.jupiter.api.Assertions.*;

class ReservableProxyTest {

    private Customer customer1, customer2;
    private ReservableProxy reservableProxy;

    @BeforeEach
    void setUp() {
        customer1 = new Customer( username: "Bernardo" , password: "123");
        customer2 = new Customer( username: "Anna" , password: "456");
        reservableProxy = new ReservableProxy();
    }

    @Test
    void addUser() {
        ArrayList<Customer> customers = new ArrayList();
        customers.add(customer1);
        reservableProxy.addUser(customer1);

        assertEquals(reservableProxy.getCustomers(), customers);

        customers.add(customer2);
        reservableProxy.addUser(customer2);

        assertEquals(reservableProxy.getCustomers(), customers);
    }

    @Test
    void setReservation() {
        reservableProxy.addUser(customer1);
        reservableProxy.setReservation(customer1, new Flight( id: 2304, destination: "Milano" , departure: "Roma"));
        Reservation reservation = new Reservation();
        reservation.addFlight(new Flight( id: 2304, destination: "Milano" , departure: "Roma"));

        assertEquals(reservableProxy.getReservation(customer1), reservation);
        assertEquals(reservableProxy.getReservation(new Customer( username: "Bernardo" , password: "456")), actual: null);
    }

    @Test
    void removeReservation() {
        reservableProxy.addUser(customer1);
        reservableProxy.removeReservation(customer1, new Flight( id: 2304, destination: "Milano" , departure: "Roma"));
        Reservation reservation = new Reservation();

        assertEquals(reservableProxy.getReservation(customer1), reservation);
    }
}
```

Figura 4.4: Codice java della classe ReservableProxyTest

4.1.5 ObserverTest

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class ObserverTest {

    private Airport airport;
    private Customer customer;
    private ReservableProxy reservableProxy;
    private Flight flight1, flight2;

    @BeforeEach
    void setUp() {
        airport = new Airport( name: "Firenze");
        reservableProxy = new ReservableProxy();
        flight1 = new Flight( id: 1732, destination: "Roma", airport.getName());
        flight2 = new Flight( id: 3845, destination: "Milano", airport.getName());
        airport.getFlightTable().addFlight(flight1);
        airport.getFlightTable().addFlight(flight2);
        RealReservableFlightArchive.getInstance().addAirport(airport);
        customer = new Customer( username: "Miriam", password: "789");
    }

    @Test
    void testObserver() {
        reservableProxy.addUser(customer);
        reservableProxy.setReservation(customer, flight1);
        reservableProxy.setReservation(customer, flight2);

        Reservation reservation = new Reservation();
        reservation.addFlight(flight1);
        reservation.addFlight(flight2);

        assertEquals(reservableProxy.getReservation(customer), reservation);

        airport.getFlightTable().removeFlight(flight1);
        reservation.removeFlight(flight1);

        assertEquals(reservableProxy.getReservation(customer), reservation);
    }
}
```

Figura 4.5: Codice java della classe ObserverTest