INGEGNERIA DEL SOFTWARE
Approfondimento

Alessandro Raspanti

April 3, 2021

Contents

1 Introduzione

1.1 Scenario e finalità

In questo elaborato viene approfondito l'utilizzo di alcuni Design Pattern trattati durante il corso. L'obiettivo è quello di combinare i Design Patterns Observer, Singleton e Factory Method in modo da realizzare un'applicazione software che permetta la gestione di un semplice sistema di voli e aereoporti. Il progetto è stato versionato con Git ed è disponibile su GitHub al link: `https://github.com/alethecine96/FlightCenter`
Si desidera inoltre che il sistema possa:

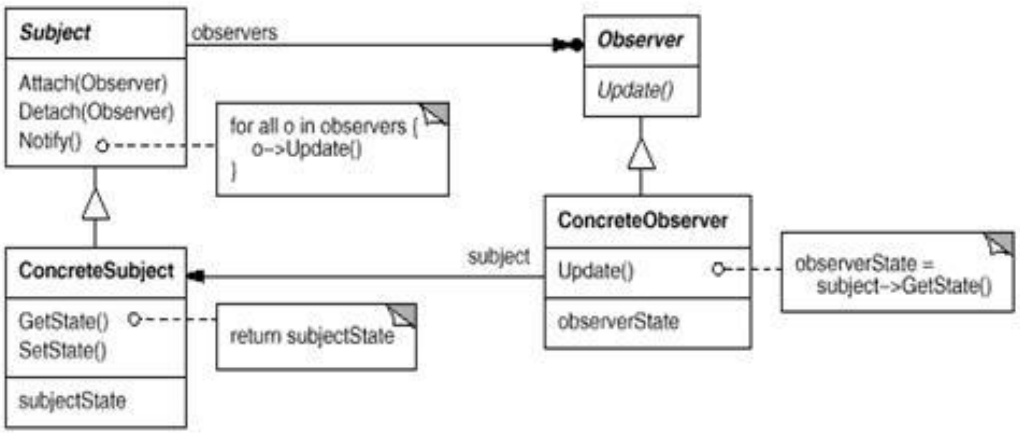
- Permettere la creazione di differenti tipologie di aerei da assegnare a seconda della caratteristica del volo
- Permettere la gestione dei voli pronti al decollo e all'atterraggio.
-

Durante lo sviluppo, è risultato ragionevole anche l'utilizzo del Pattern Singleton per garantire l'unicità di un Time che scandisce il tempo

1.2 Design Patters

1.2.1 Observer

L'Observer è un pattern comportamentale che permette di definire una dipendenza uno a molti in modo tale che se un oggetto cambia il suo stato, tutti gli oggetti dipendenti da questo ricevano la notifica di tale cambiamento.



Questo pattern è composto dai seguenti partecipanti:

- Subject:

  Mantiene una collezione di Observer e fornisce operazioni per l'aggiunta, la cancellazione e per la notifica di cambiamento di stato agli Observer.

- Observer:

  Specifica un'interfaccia per l'aggiornamento dello stato degli Observer in base agli eventi che interessano il ConcreteSubject.

- ConcreteSubject: classe ObservedSubject.

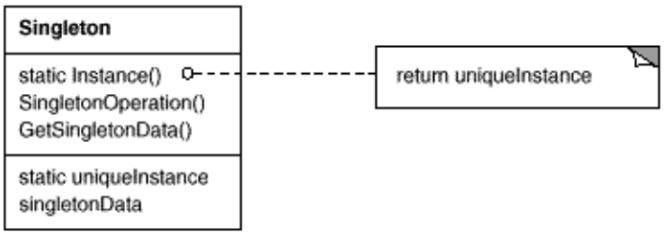
  Mantiene lo stato del soggetto osservato e notifica gli Observer del proprio cambiamento di stato invocando le operazioni di notifica ereditate dal Subject.

- ConcreteObserver:

  Implementa l'interfaccia dell'Observer definendo il comportamento in caso di cambio di stato del soggetto osservato

1.2.2 Singleton

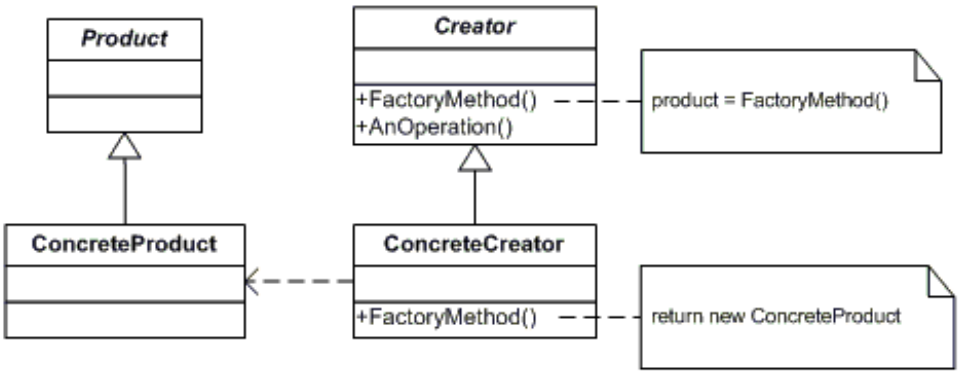
Il singleton, è un design pattern creazionale che ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza, fornendo un punto di accesso globale a tale istanza. L'implementazione più semplice di questo pattern prevede che la classe singleton abbia un unico costruttore privato, in modo da impedire l'istanziazione diretta della classe. La classe fornisce inoltre un metodo "getter" statico che restituisce l'istanza della classe (sempre la stessa), creandola alla prima chiamata del metodo, e memorizzandone il riferimento in un attributo privato anch'esso statico.



1.2.3 Factory Method

Il Factory Method, è un design pattern creazionale che fornisce un'interfaccia comune per la creazione di oggetti. Rimanendo sull'astratto, garantisce che

non ci sia necessità da parte dei client di specificare la tipologia delle classi concrete utilizzate all'interno del proprio codice, riducendo quindi l'accoppiamento ed evitando che vengano introdotte delle limitazioni sulla portabilità del sistema. La creaziane degli oggetti avviene delegando un Creator che, in base alle informazioni ricevute, saprà quale oggetto restituire. In questo modo si garantisce che un sistema sia indipendente dall'implementazione degli oggetti concreti e si da la possibilità a un client, attraverso un'interfaccia comune, di utilizzare diverse famiglie di prodotti dotate di stesse funzionalità comuni ma con diverse implementazioni.



Questo pattern è composto dai seguenti partecipanti:

- Creator: interfaccia delle operazioni mirate alla creazione dai prodotti concreti.
- ConcreteCreator: implementazione delle operazioni per la creazione dei prodotti concreti.
- Product: interfaccia delle operazioni dei prodotti concreti creati dal Creator.
- ConcreteProduct: implementazione delle operazioni dei prodotti concreti.

2 Diagramma delle classi

3 Discussione delle classi

- AbstractPlane: definisce la classe base che viene estesa dalle 3 classi MidPassengnerPlane, LittlePassengerPlane e BigPassengerPlane e i metodi/attributi in comune tra le 3 classi derivate.

- Airport definisce gli aereoporti con coordinate, nome e numero di piste di atterraggio. Questi attributi vengono definiti in un file di testo e caricati quando vengono creati gli aereoporti e scelto l'aereoporto da controllare.

- BigPassengerPlane: è la tipologia di aereo più grande con la maggior capacità di passeggeri, di autonomia ma minor velocità

- Flight: definisce il volo in tutte le sue caratteristiche, tra le quali il chilometraggio del volo calcolato attraverso le coordinate dell'aereoporto di partenza e quello di arrivo. Tra i metodi troviamo il land() e il take-off().

- FlightManager: La classe Game è la classe principale del progetto poiché gestisce l'interfaccia del centro di controllo con l'utente. E' un observer concreto, perchè deve osservare la classe time (subject) in modo tale da far andare avanti le ore del giorno. Inizialmente permette di scegliere l'aereoporto che si vuole controllare attraverso il metodo setairportcontrol(); una volta fatta la scelta attraverso il metodo createFlight() si occupa di organizzare i voli verso gli altri aereoporti a orari casuali e con un numero di passeggeri casuali, numero che inciderà sulla creazione di una tipologia di aereo adatta da assegnare al volo; non solo il numero di passeggeri incide sulla tipologia di aereo, ma anche la distanza di volo calcolata attraverso le coordinate dell'aereoporto di partenza e quello di arrivo. L'observer pattern è stato implementato in modalità push(è l'observer a prendere il dato dal subject): tramite il metodo startTime() viene fatto partire il conteggio delle ore e ogni volta che l'observer viene informato che è passata un'ora avvia la gestione dei voli in partenza e in arrivo, con output del volo in partenza e richiesta all'utente di selezionare la pista per il decollo/atterraggio.

- LandingStrip: è la classe rappresentante le piste di atterraggio di un aereoporto, forniscono i metodi per capire se sono occupate o libere per permettere al centro di controllo di far decollare o atterrare un aereo.

- LittlePassengerPlane: è la tipologia di aereo più piccolo con la minor capacità di passeggeri, di autonomia ma maggior velocità.

- MidPassengerPlane : è la tipologia di aereo di dimensioni media e con una capacità di passeggeri, di autonomia e velocità che si trova a metà strada tra l'aereo più piccolo e quello più grande.

- Observer: definisce un interfaccia implementata dagli Observer concreti. Per questa implementazione definisce solo il metodo update().

- PlaneCreator: Assume il ruolo di Concrete Creator tipico del pattern Factory Method; definisce l'operazione che si occupa di istanziare oggetti di classe AbstractPlane.

- Subject: è una classe astratta per rappresentare i Subject come descritto dal design pattern Observer. Per questa implementazione ha un solo un attributo Observer e il metodo notify().

- Time: è il subject del design pattern Observer, inoltre è implementato come singleton. Quando viene chiamato il metodo startTime() il quale ogni 3 secondi incrementa il tempo e a sua volta chiama notify().

4 Possibilità di espansione

5 Implementazione classi

Vengono riportati frammenti di codice per ogni classe. Sono stati tralasciati gli import, i getter e setter e alcune funzioni di stampa.

Listing 1: AbstractPlane

```
1  public abstract class AbstractPlane {
2      private String planeName;
3      private int planeId;
4      private int speed;
5      private int range;
6
7          AbstractPlane(){
8              planeId = (int)(Math.random()*1000000);
9          }
10 }
```

Listing 2: Airport

```
1  public class Airport {
2      private ArrayList<LandingStrip> landingStrips = new↩
           ArrayList<>();
3      private String name;
4      private double latitudine;
5      private double longitudine;
6
7          Airport(String name,int landing, double ↩
              latitudine, double longitudine){
8              for(int i=0; i<landing; i++){
9              landingStrips.add(new LandingStrip(i));
10             }
11             this.name = name;
12             this.latitudine = latitudine;
13             this.longitudine = longitudine;
14         }
15
16     boolean getFullLanding(){
17         for (LandingStrip landingStrip : landingStrips)↩
               {
18         if(!landingStrip.getFull()){
19         return false;
20             }
21         }
22         return true;
23     }
24 }
```

Listing 3: BigPassengerPlane

```
1  public class BigPassengerPlane extends AbstractPlane {
2      BigPassengerPlane() {
3          super();
4          setSpeed(750);
5          setRange(3000);
6      }
7
8  }
```

Listing 4: Flight

```
1  public class Flight {
2      private int departurehour;
3      private int idFlight;
4      private int numberOfPassengers;
5      private AbstractPlane abstractPlane;
6      private Airport airportdeparture;
7      private Airport airportarrive;
8      private LandingStrip landingStrip;
9      private boolean inflight;
10     private int houroffly;
11     private boolean isWaitingforLanding;
12     private int lenghtofflight;
13
14
15         Flight(Airport airportdeparture, Airport ↩
               airportarrive, AbstractPlane abstractPlane, ↩
               int numberOfPassengers, int departurehour) {
16             this.departurehour = departurehour;
17             this.numberOfPassengers = ↩
                   numberOfPassengers;
18             this.abstractPlane = abstractPlane;
19             this.airportdeparture = airportdeparture;
20             this.airportarrive = airportarrive;
21             idFlight = (int)(Math.random()*10000);
22         }
23
24     void takeoff() {
25         System.out.println("Il volo " + getIdFlight() +↩
               " con a bordo "+numberOfPassengers+" ↩
               passeggeri sta decollando dalla pista numero↩
               "+ (getLandingStrip().getNumber()+1));
26         inflight=true;
27         getLandingStrip().setFull(true);
28         for(int j=0;j<30;j++) {
29             System.out.print("-");
30         try {
```

```
31              Thread.sleep(100);
32          } catch (InterruptedException e) {
33              e.printStackTrace();
34          }
35          }
36
37      }
38
39      void land(){
40          System.out.println("Il volo " + getIdFlight() +↩
                " sta atterrando sulla pista numero " + (↩
               getLandingStrip().getNumber()+1));
41          System.out.println("");
42          getLandingStrip().setFull(true);
43          inflight = false;
44      }
45  }
```

Listing 5: FlightManager

```
1   public class FlightManager implements Observer {
2       private PlaneCreator planeCreator;
3       private ArrayList<Flight> fly;
4       private Time time;
5       private Airport airporttocontrol;
6       public ArrayList<Airport> airports;
7       private ArrayList<Flight> flights;
8
9           FlightManager(){
10              planeCreator = new PlaneCreator();
11              fly = new ArrayList<>();
12              airports = new ArrayList<>();
13              time = Time.createTime(this);
14              flights = new ArrayList<>();
15          }
16
17      void setairportcontrol() throws ↩
           FileNotFoundException{
18          Scanner scanner = new Scanner(new File("./File/↩
               Coordinate"));
19          while (scanner.hasNextLine()) {
20              airports.add(new Airport(scanner.next(),↩
                   scanner.nextInt(), getCoordinate(scanner↩
                   .nextInt(), scanner.nextInt(),scanner.↩
                   nextInt()), getCoordinate(scanner.↩
                   nextInt(), scanner.nextInt(),scanner.↩
                   nextInt())));
21          }
```

```
22          System.out.println("Quale aereoporto vuoi ↩
                controllare?");
23          for(int j = 0; j<airports.size();j++){
24              System.out.println((j+1)+" - "+airports.get↩
                    (j).getName());
25          }
26          Scanner input = new Scanner(System.in);
27          System.out.println(" ");
28          int l = input.nextInt()-1;
29          System.out.println("stai controllando l'↩
                aereoporto di "+airports.get(l).getName());
30          airporttocontrol = airports.get(l);
31      }
32
33      void createFlight(){
34          Airport airportdeparture = airporttocontrol;
35          int[] departurehour = {1, 5, 8, 11, 17};
36          int numberOfPassengers = 0;
37          int departurehours = 0;
38          Airport airportarrive;
39          for (Airport airport : airports) {
40              if (airport.getName().equals(↩
                    airportdeparture.getName()))
41                  continue;
42              numberOfPassengers = (int) (Math.random() *↩
                     250);
43              departurehours = departurehour[(int) (Math.↩
                    random() * (departurehour.length))];
44              airportarrive = airport;
45              double distance = getDistance(↩
                    airportdeparture.getLatitudine(), ↩
                    airportdeparture.getLongitudine(), ↩
                    airport.getLatitudine(), airport.↩
                    getLongitudine());
46              String planeType;
47              if (numberOfPassengers <= 75 && distance↩
                    <1200) planeType = "LITTLE";
48              else if (numberOfPassengers <= 150 && ↩
                    distance<1700) planeType = "MID";
49              else planeType = "BIG";
50              AbstractPlane plane = planeCreator.↩
                    createPlane(planeType);
51              Flight flight = new Flight(airportdeparture↩
                    , airportarrive, plane, ↩
                    numberOfPassengers, departurehours);
52              flight.setLenghtofflight((int)(distance));
53              flight.setHouroffly((int)(Math.ceil(↩
                    distance/plane.getSpeed())));
54              flights.add(flight);
```

```
55          }
56      }
57
58      void start(){
59          time.startTime();
60      }
61
62      @Override
63      public void update() {
64          System.out.println("");
65          System.out.println("Sono le ore " + time.↩
                getHour() + ":00");
66          time.waitSec(3);
67          for (Flight flight : flights) {
68              if (flight.isInflight()) {
69              int infly = (time.getHour() - flight.↩
                    getDeparturehour());
70                  if (flight.getHouroffly() == infly) {
71                      System.out.println("Il volo " + ↩
                            flight.getIdFlight() + " sta ↩
                            atterrando all'aereoporto di " +↩
                             flight.getAirportarrive().↩
                            getName());
72                      System.out.println(" ");
73                  } else if (flight.getHouroffly() * 2 ==↩
                        infly || flight.isWaitingforLanding↩
                        ()) {
74                      if(!flight.getAirportdeparture().↩
                            getFullLanding()) {
75                          System.out.println("");
76                          System.out.println("Selezionare↩
                                una pista per l' ↩
                                atterraggio");
77                          int landing = ↩
                                selectlandingstrip();
78                          flight.setLandingStrip(↩
                                airporttocontrol.↩
                                getLandingStrips().get(↩
                                landing));
79                          flight.land();
80                          time.waitSec(2);
81                      }
82                  else{
83                      System.out.println("non posso ↩
                            atterrare ci sono tutte le piste↩
                             piene, rimango in volo e ↩
                            effettuo un nuovo tentativo all'↩
                            ora successiva");
84                      flight.setWaitingforLanding(true);
```

```
85                      }
86                  }
87              }
88              if (flight.getDeparturehour() == time.↩
                    getHour()) {
89                  if (flight.getAirportdeparture().↩
                        getFullLanding()) {
90                      flight.setDeparturehour(flight.↩
                            getDeparturehour() + 1);
91                  } else {
92                      System.out.println("");
93                      System.out.println("Selezionare una↩
                            pista per il decollo");
94                      int landing = selectlandingstrip();
95                      flight.setLandingStrip(↩
                            airporttocontrol.↩
                            getLandingStrips().get(landing))↩
                            ;
96                      flight.takeoff();
97                      time.waitSec(2);
98              }
99      }
100
101     }
102     for (LandingStrip landingStrip : airporttocontrol.↩
            getLandingStrips()) {
103         landingStrip.setFull(false);
104     }
105 }
106
107     double getDistance(double latitudestart, double ↩
            longitudestart, double latitudearrive, double ↩
            longitudearrive){
108         double difference = (longitudestart-↩
                longitudearrive);
109         double dist = Math.acos(Math.sin(latitudestart)↩
                * Math.sin(latitudearrive) + Math.cos(↩
                latitudestart) * Math.cos(latitudearrive) * ↩
                Math.cos(difference)) * 6371;
110         return dist;
111     }
112
113     double getCoordinate(double gradi, double primi, ↩
            double secondi){
114         double risultato;
115         risultato = gradi+(primi/60)+(secondi/3600);
116         double u = (risultato*2*3.14/360);
117         return u;
118     }
```

```
119
120     int selectlandingstrip()  {
121         for(int j = 0; j<airporttocontrol.↩
                getLandingStrips().size();j++){
122             if(airporttocontrol.getLandingStrips().get(↩
                    j).getFull())
123                 //Stampa pista occupata
124             else
125                 //Stampa pista libera
126         }
127         Scanner input = new Scanner(System.in);
128         System.out.println(" ");
129         int i = input.nextInt()-1;
130         while(i>=airporttocontrol.getLandingStrips().↩
                size() || airporttocontrol.getLandingStrips↩
                ().get(i).getFull() ) {
131             //Stampa errore
132             i = input.nextInt()-1;
133             }
134
135             return i;
136         }
137     }
```

Listing 6: LandingStrip

```
1  public class LandingStrip {
2      private int number;
3      private Boolean full = false;
4
5      LandingStrip(int number) {
6          this.number = number;
7      }
8
9  }
```

Listing 7: LittlePassengerPlane

```
1  public class LittlePassengerPlane extends AbstractPlane↩
       {
2      LittlePassengerPlane() {
3          super();
4          setSpeed(1500);
5          setRange(1200);
6      }
7  }
```

Listing 8: MidPassengerPlane

```
public class MidPassengerPlane extends AbstractPlane {
    MidPassengerPlane(){
        super();
        setSpeed(1000);
        setRange(1700);
    }
}
```

Listing 9: Observer

```
public interface Observer {
    void update();
}
```

Listing 10: PlaneCreator

```

public class PlaneCreator {

    AbstractPlane createPlane(String plane){
        switch (plane){
            case "LITTLE":
                return new LittlePassengerPlane();
            case "MID":
                return new MidPassengerPlane();
            case "BIG":
                return new BigPassengerPlane();
        }
        return null;
    }
}
```

Listing 11: Subject

```
abstract class Subject {
    Observer observer;
    void notifyObserver(){
        observer.update();
    }
}
```

Listing 12: Time

```
class Time extends Subject{
```

```
2       private static Time time;
3       private int hour = 0;
4
5       private Time(Observer observer) {
6           this.observer = observer;
7       }
8
9       static Time createTime(Observer observer){
10          if(time==null)
11              time = new Time(observer);
12          return time;
13      }
14
15      void startTime(){
16          while(hour !=24){
17              waitSec(3);
18              notifyObserver();
19              hour++;
20          }
21      }
22
23      void waitSec(int seconds){
24      try {
25          Thread.sleep(seconds*1000);
26      }catch (InterruptedException e){
27          e.printStackTrace();
28      }
29  }
```

6 Test

Per i test è stato usato 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 pattern utilizzati.

Listing 13: AirportTest

```
1
2   public class AirportTest {
3       private Airport airportTest;
4
5       @BeforeEach
6       void setAirportTest(){
7           airportTest = new Airport("Test", 5, 0, 0);
8       }
9
10      @Test
```

```
11      void getName(){assertEquals("Test",airportTest.↩
            getName());}
12
13      @Test
14      void getLandingFull(){
15              for(LandingStrip landingStrip : airportTest↩
                    .getLandingStrips()){
16                  assertFalse(airportTest.getFullLanding↩
                        ());
17                  landingStrip.setFull(true);
18                  }
19              assertTrue(airportTest.getFullLanding());
20      }
21  }
```

Listing 14: FlightManagerTest

```
1   public class FlightManagerTest {
2       private FlightManager flightManagerTest;
3       private ArrayList<Airport> airports;
4
5       @BeforeEach
6       void setFlightManagerTest(){
7           flightManagerTest = new FlightManager();
8           airports = new ArrayList<>();
9       }
10
11      @Test
12      void airportCreation() throws FileNotFoundException↩
            {
13          Scanner scanner = new Scanner(new File("./File/↩
                Coordinate"));
14          while (scanner.hasNextLine()) {
15              airports.add(new Airport(scanner.next(),↩
                    scanner.nextInt(), flightManagerTest.↩
                    getCoordinate(scanner.nextInt(), scanner↩
                    .nextInt(),scanner.nextInt()), ↩
                    flightManagerTest.getCoordinate(scanner.↩
                    nextInt(), scanner.nextInt(),scanner.↩
                    nextInt())));
16          }
17          for(Airport airport: airports) {
18              for (Airport airport1 : airports) {
19                  int distance = (int) (flightManagerTest↩
                        .getDistance(airport.getLatitudine()↩
                        , airport.getLongitudine(), airport1↩
                        .getLatitudine(), airport1.↩
                        getLongitudine()));
```

```
20                  int distance1 = (int) (↩
                        flightManagerTest.getDistance(↩
                        airport1.getLatitudine(), airport1.↩
                        getLongitudine(), airport.↩
                        getLatitudine(), airport.↩
                        getLongitudine()));
21                  assertEquals(distance, distance1);
22              }
23          }
24          Scanner scanner2 = new Scanner(new File("./File↩
                /Coordinate"));
25          ArrayList<String> airportName = new ArrayList↩
                <>();
26          while(scanner2.hasNextLine()){
27              airportName.add(scanner2.next());
28              scanner2.nextLine();
29          }
30          int i = 0;
31          for(Airport airport3: airports){
32              assertEquals(airport3.getName(), ↩
                    airportName.get(i));
33              i++;
34          }
35      }
36  }
```

Listing 15: FlightTest

```
1   public class FlightTest {
2       private Flight flighttest;
3       private Airport airportstart;
4       private Airport airportarrive;
5       private PlaneCreator planeCreator;
6
7       @BeforeEach
8       void setFlighttest(){
9           airportstart = new Airport("Partenza", 2, 0, 0)↩
                ;
10          airportarrive = new Airport("Arrivo", 2, 0, 0);
11          planeCreator = new PlaneCreator();
12          flighttest = new Flight(airportstart, ↩
                airportarrive, planeCreator.createPlane("↩
                LITTLE"),50, 2 );
13      }
14
15      @Test
16      void flighttest(){
17          assertEquals(flighttest.getNumberOfPassengers()↩
```

```
            , 50);
18          assertEquals(flighttest.getDeparturehour(), 2);
19      }
20
21      @Test
22      void landTakeOffTest(){
23          flighttest.setLandingStrip(airportstart.↩
                getLandingStrips().get(0));
24          flighttest.takeoff();
25          assertTrue(flighttest.isInflight());
26          assertEquals(flighttest.getLandingStrip().↩
                getNumber(), 0);
27          flighttest.land();
28          assertFalse(flighttest.isInflight());
29      }
30  }
```

Listing 16: PlaneBuilderTest

```
1   public class PlaneBuilderTest {
2       private PlaneCreator testplaneCreator;
3
4       @BeforeEach
5       void setTestPlaneCreator(){
6           testplaneCreator = new PlaneCreator();
7           testplaneCreator.createPlane("LITTLE");
8       }
9
10      @Test
11      void testPlaneCreation(){
12          LittlePassengerPlane littlePassengerPlane = new↩
                LittlePassengerPlane(1500, 1200);
13          MidPassengerPlane midPassengerPlane = new ↩
                MidPassengerPlane(1000, 1700);
14          MidPassengerPlane bigPassengerPlane = new ↩
                MidPassengerPlane(750, 3000);
15          assertEquals(littlePassengerPlane.getSpeed(), ↩
                testplaneCreator.createPlane("LITTLE").↩
                getSpeed());
16          assertEquals(littlePassengerPlane.getRange(), ↩
                testplaneCreator.createPlane("LITTLE").↩
                getRange());
17          assertEquals(midPassengerPlane.getSpeed(), ↩
                testplaneCreator.createPlane("MID").getSpeed↩
                ());
18          assertEquals(midPassengerPlane.getRange(), ↩
                testplaneCreator.createPlane("MID").getRange↩
                ());
```

```
19          assertEquals(bigPassengerPlane.getSpeed(), ↩
                testplaneCreator.createPlane("BIG").getSpeed↩
                ());
20          assertEquals(bigPassengerPlane.getRange(), ↩
                testplaneCreator.createPlane("BIG").getRange↩
                ());
21
22      }
23  }
```

Listing 17: TimeTest

```
1   public class TimeTest {
2       class TestObserver implements Observer{
3       boolean called = false;
4
5           @Override
6           public void update() {
7               called = true;
8           }
9       }
10
11      private TestObserver testObserver = new ↩
            TestObserver();
12      private Time testTime;
13
14      @BeforeEach
15      void createTime(){
16          testTime = Time.createTime(testObserver);
17      }
18
19      @Test
20      void testSingleton(){
21          Time testTime2 = Time.createTime(testObserver);
22          assertEquals(testTime,testTime2);
23      }
24
25      @Test
26      void testCalledUpdate(){
27          testTime.observer = testObserver;
28          testTime.notifyObserver();
29          assertTrue(testObserver.called);
30      }
31  }
```