Air Traffic Controller

Ingegneria del software - Progetto e sviluppo di un domain model

Niccolò Benedetto MAT. 7024656

Data stesura:	Luglio 2023
Partners:	Takoukam Tonga Yvan
Docente:	Enrico Vicario

1 Introduzione

In questo elaborato viene posta l'attenzione sul design pattern Mediator, nonchè sul buon utilizzo di alcune pratiche dedite all'analisi, alla progettazione e all'implementazione di un programma JAVA e quindi di un modello di dominio *well-formed*.

In ingegneria del software il Mediator pattern è un design pattern utilizzato nella programmazione orientata agli oggetti che incapsula le modalità con cui oggetti diversi interagiscono fra loro. Si tratta di un pattern comportamentale, ossia operante nel contesto delle interazioni tra oggetti, che ha l'intento di disaccoppiare entità del sistema che devono comunicare fra loro. Il pattern infatti fa in modo che queste entità non si referenzino reciprocamente, ma si riferiscano a un agente d'intermediazione, riducendo così di gran lunga il numero di interconnessioni. Il beneficio principale consiste nel permettere la modifica agile delle politiche di interazione, poiché le entità coinvolte devono fare riferimento al loro interno solamente al mediatore.

La stesura dell'elaborato riporta:

- Uno scenario reale rappresentativo del modello di dominio scelto;
- Un'analisi complessiva in cui si esplicitano i vantaggi del pattern scelto in relazione al contesto da modellare, dunque si pone l'attenzione sugli strumenti di analisi da utilizzarsi in fase di progettazione;
- La progettazione della struttura del modello, con particolare circospezione circa i suoi partecipanti e le relazioni che tra questi sussistono;
- Il dettaglio di frammenti di codice della realizzazione che illustrano aspetti salienti dello schema, dunque la definizione di casi di test, realizzati con *JUnit*, che esercitino lo schema in uno scenario che ne caratterizza l'intento.

2 Scenario proposto

I piloti di veivoli che decollano dagli aereoporti non comunicano direttamente tra di loro. Questi infatti dialogano con un controllore del traffico aereo che si trova in un alta torre da qualche parte vicino alla pista. Se così non fosse ogni pilota dovrebbe conoscere le intenzioni di tutti gli altri colleghi, scaturendo non banali problematiche di comunicazione attraverso il canale radio. Dunque la torre di controllo non si occupa dell'intera tratta di volo, ma il suo compito è quello di mantenere l'ordine e organizzare i decolli di ogni veivolo.

Supponiamo che nella piccola isola turistica di *Lanai* (Hawaii) sia presente un aereporto dotato di due sole piste di decollo. Dalla $Runway_1$ (2500 metri) in genere decollano i voli di linea, individuati attraverso la sigla "AL" che precede

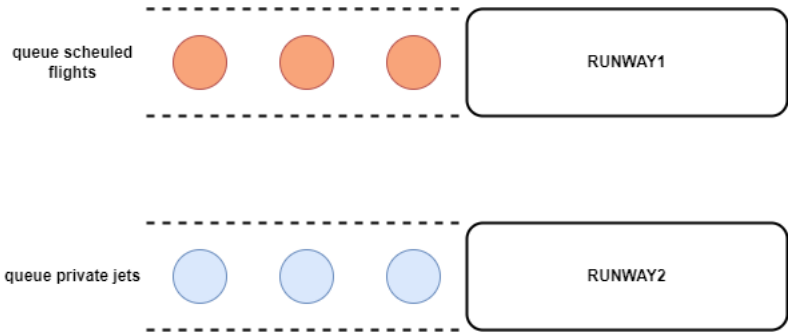

Figure 1: proposed scenario

il resto dell'identificatore del volo (es. "AL-001"), mentre la $Runway_2$ (2000 metri) è riservata ai private jet, il quale identifier prevede il prefisso "PJ" (es. "PJ-001"). Ogni veivolo per decollare correttamente deve in prima battuta effettuare una fase di rullaggio sulla pista assegnata dalla torre di controllo.

Le politiche aereoportuali dell'isola, affinchè sia garantita la minor congestione possibile, permettono ai piloti, in caso di code generate in fase di rullaggio, di poter fare richiesta di un cambio pista. Qualora la pista richiesta non abbia veivoli accodati il controllore del traffico aereo può soddisfare la volontà dei piloti. Segue una raffigurazione delle possibili configurazioni che si possono venire a creare.

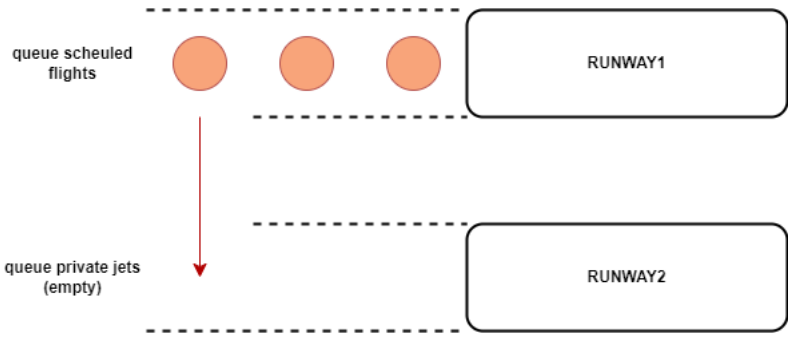

Figure 2: a possible configuration

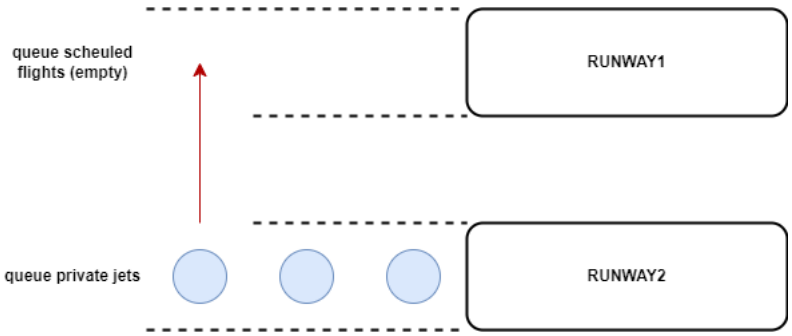

Figure 3: a possible configuration (realistically less likely)

3 Analisi

L'utilizzo del pattern Mediator permette di incapsulare il comportamento collettivo delle diverse classi che compongono il sistema (denominate *colleagues* nel gergo del pattern) mediante il ricorso a una classe separata nota proprio come *Mediator*, che diviene quindi quel famoso agente d'intermediazione tra i diversi oggetti.

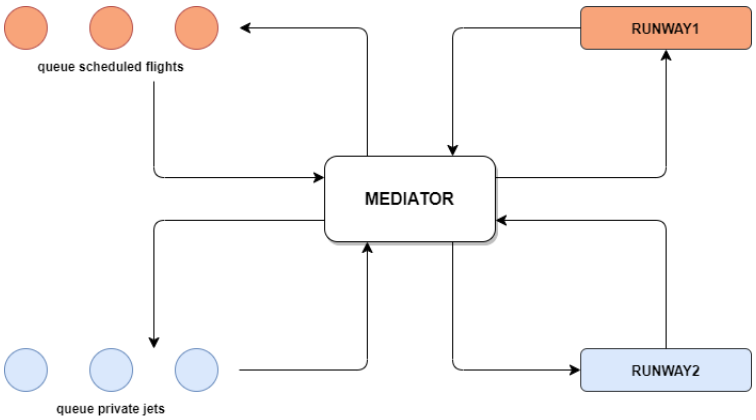

Figure 4: Mediator logic

Di seguito si definisce il sistema pattern Mediator, a cui viene fatto fede in fase di progettazione del prospetto da realizzare, quindi si elargisce una breve descrizione di ogni classe che compone lo schema. La notazione utilizzata è da ricercarsi nel linguaggio visivo dedito alla specifica, costruzione e documentazione di artefatti di sistema conosciuto come UML.

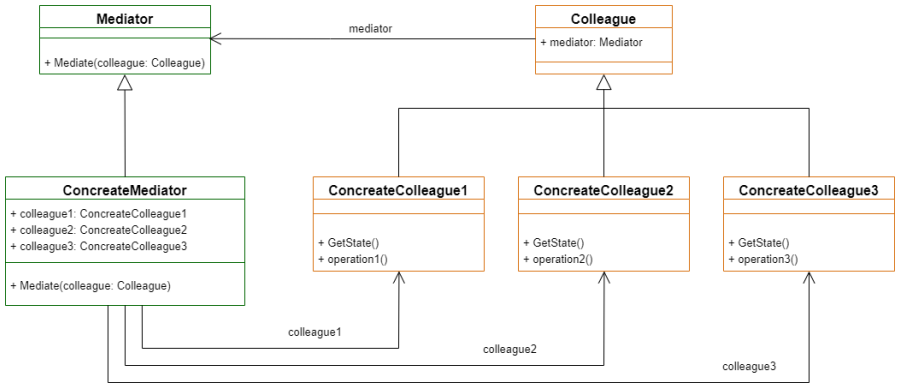

Figure 5: UML Mediator pattern

Mediator Definisce un'interfaccia per la comunicazione tra i colleague objects.

ConcreateMediator Implementa l'interfaccia definita dal Mediator, quindi mantiene la lista dei colleague objects e coordina lo scambio dei messaggi tra questi.

Colleague Definisce la classe astratta (o l'interfaccia) dei colleague objects.

ConcreateColleague Implementa le funzioni presentate nella classe *Colleague*. In uno scenario realistico ce ne possono essere di tanti tipi diversi.

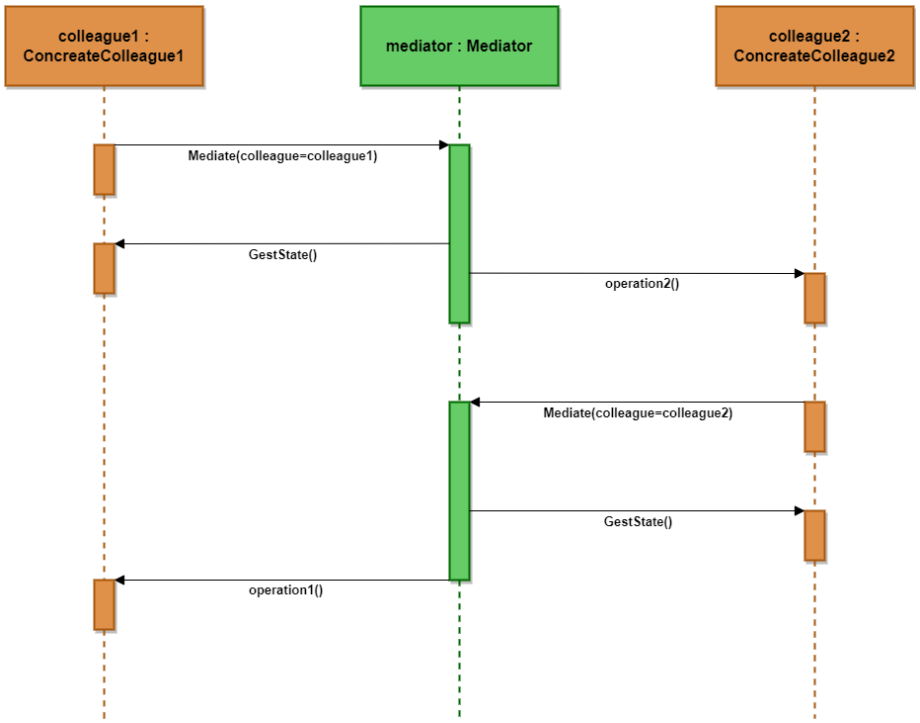

Figure 6: Sequence diagram Mediator

Il sequence diagram è un tipo di diagramma appartenente sempre all'*Unifed Modeling Language* (UML) che mostra sostanzialmente come gli "attori" del sistema si scambiano le informazioni in un ordine particolare. Si tenga presente infatti che in un sistema vengono costantemente effettuate richieste e inviate risposte. Il destinatario prende una decisione in base alla richiesta specifica e alle sue regole predeterminate. Una simile rete di possibli decisioni e interazioni (conosciuta come *Activity Diagram*) assume le sembianze di un albero fortemente ramificato, ed è direttamente ottenuto a sua volta da un altro particolare artefatto denominato *Use Cases Diagram*, che pone l'attenzione appunto sugli "attori" e sulle attività del sistema di cui questi si fanno carico. Il sequence diagram non rappresenta altro che un percorso specifico all'interno di questa rete, rete che quindi definisce una modalità di analisi degli *Uses cases*. Per cui un tale strumento supporta l'analisi logica per sottosezioni di sistemi, ma perde di valore se utilizzato per rappresentare l'intero apparato. Il sequence diagram sopra riportato definisce uno scenario di test puramente dimostrativo per il pattern Mediator. Si tenga presenta che l'elaborato debba riportare in maniera rigorosa nella sezione di Progettazione un *Use Case Diagram* suggestivo del sistema e il rispettivo *Activity Diagram*.

Si riassumano adesso i vantaggi derivanti dall'utilizzo del pattern Mediator in un tale scenario:

- Disaccoppiamento dei colleagues: i colleagues dialogano tra loro passando esclusivamente per il Mediator. Questo permette di poter rimpiazzare un object nella struttura creata con un object differente senza influenzare le classi definite;
- Semplificazione delle connessioni: il Mediator permette di ridurre le connessioni dei colleagues da *many-to-many* a *one-to-many*. Si osservi che però, all'aumentare progressivo dei colleagues, il Mediator diviene più complesso e quindi aumentano le difficoltà del suo mantenimento;
- Controllo centralizzato: il controllo delle comunicazioni è centralizzato, per cui si ha una visione complessiva del sistema e una gestione più efficiente delle modifiche.
- Subclassing ridotto: partendo dal presupposto che l'intera logica dello schema è incapsulata nel Mediator, qualora volessi aggiungere una dipendenza a una classe, bisognerà solamente espandere la Mediator class. Quindi in generale non è necessaria la realizzazione di subclasses.

Si osservi tuttavia che realizzando una tale struttura concettuale, si espone l'intero sistema a un *single point of failure*: nel caso di malfunzionamento del Mediator l'apparato complessivo sarà coinvolto con conseguente isolamento di ogni colleague.

4 Progettazione e Codice

Nel sistema sono presenti 7 classi principali (di cui due di queste adempiono al mero compito di definire una prova di test del sistema puramente dimostrativa; queste classi sono riconoscibili mediante il prefisso *Test*, ma è stato scelto di non rappresentarle all'interno del modello UML col fine di porre l'attenzione maggiormente sulla struttura realizzativa del sistema e non sul suo collaudo). L'interazione tra queste emula il comportamento presentato dal pattern Mediator. Per il corretto funzionamento di tutto il sistema, è necessaria una stretta collaborazione tra le varie classi, ognuna delle quali ha delle responsabilità a cui non si può sottrarre. Si è data particolare importanza ai modificatori di visibilità, per garantire che nessuna classe possa eseguire operazioni che non le sono permesse. Inoltre si è pensato di racchiudere tutte le classi all'interno di uno stesso package denominato *designmodel.mediator*. Segue la modelizzazione UML del sistema.

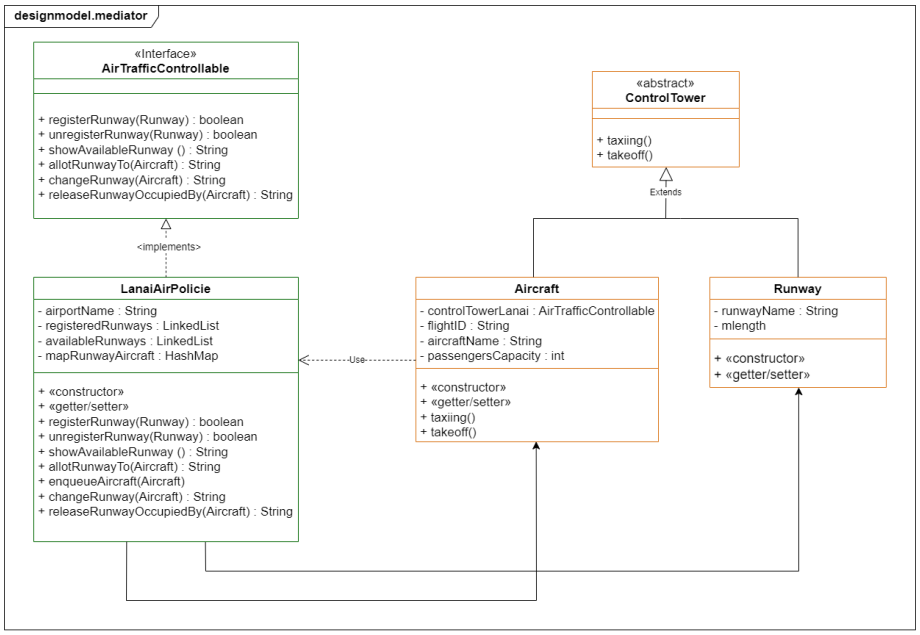

Figure 7: UML

AirTrafficControllable Interfaccia che definisce una serie di funzionalità concretizzate da una torre di controllo, circoscritte allo scenario scelto. Nell'UML del pattern rappresenta proprio la classe *Mediator*.

LanaiAirPolicie Implementa concretamente le funzionalità introdotte nell'interfaccia tenendo come riferimento le regole e i limiti imposti dal modello. Qui viene ad esempio attuata la logica necessaria per ridurre la congestione del traffico aereo generato dalle attese passive dei veivoli che devono decollare, attraverso il metodo *changeRunway()*. Tale classe assume dunque il ruolo della classe *ConcreateMediator*.

ControlTower Classe astratta che assume il ruolo di *Colleague* e quindi definisce i servizi offerti ai *ConcreateColleague*. Permette l'interfacciamento con quanto presentato attraverso il *ConcreateMediator*. L'utilizzo di una classe astratta e non di un'interfaccia è favorito dal fatto di non avere l'obbligo di implementazione dei metodi qui fissati dentro tutti i *ConcreateCollegue*. Nella progettazione delle relazioni dei partecipanti del modello caratterizzante preso come riferimento, questo vincolo deve essere rispettato.

Aircraft E' uno dei *ConcreateColleague* del nostro modello. Rappresenta uno dei due tipi di veivoli definiti nello scenario. Implementa i metodi del *ConcreateColleague*.

Runway E' l'altro *ConcreateColleague* del modello. Presenta un pezzo del

sistema fondamentale per il corretto funzionamento dei metodi delle classi sopra descritte.

TestManagementRunway Classe che sfrutta il framework *JUnit 4* per testare la corretta gestione delle piste di decollo.

TestLanaiAirPolicieLogic Classe che sfrutta il framework *JUnit 4* per testare la logica presentata nella sezione 2 dell'elaborato.

Vengono adesso riportati alcuni frammenti di codice suggestivi del programma JAVA realizzato, organizzati per classe.

```
1  public interface AirTrafficControllable {
2
3      public boolean registerRunway(Runway runway);
4
5      public boolean unregisterRunway(Runway runway);
6
7      public String showAvailableRunway();
8
9      public String allotRunwayTo(Aircraft aircraft);
10
11     public String changeRunway(Aircraft aircraft);
12
13     public String releaseRunwayOccupiedBy(Aircraft aircraft);
14
15 }
```

```
17 public class LanaiAirPolicie implements AirTrafficControllable{
18
19     private String airportName;
20     private LinkedList<Runway> registeredRunways = new
           LinkedList<>();
21     private LinkedList<Runway> availableRunways = new
           LinkedList<>();
22     private HashMap<Aircraft, Runway> mapRunwayAircraft = new
           HashMap<>();
23
24
25     public LanaiAirPolicie(String airportName) {
26         this.airportName = airportName;
27     }
28
29
30     public String getAirportName() {
31         return airportName;
32     }
33
34
```

```
35      public void setAirportName(String airportName) {
36          this.airportName = airportName;
37      }
38
39      @Override
40      public boolean registerRunway(Runway runway) {
41
42          boolean alreadyRegistered = false;
43          boolean registerOutcome = false;
44
45          for(Runway registeredRunway : registeredRunways) {
46              if(registeredRunway.equals(runway)) {
47                  alreadyRegistered = true;
48              }
49          }
50
51          if(!alreadyRegistered) {
52              this.registeredRunways.add(runway);
53              this.availableRunways.add(runway);
54              registerOutcome = true;
55          }
56
57          return registerOutcome;
58      }
59
60      @Override
61      public boolean unregisterRunway(Runway runway) {
62
63          boolean unregisterOutcome = false;
64
65          for(Runway registeredRunway : this.registeredRunways) {
66              if(registeredRunway.equals(runway)) {
67                  unregisterOutcome = true;
68                  this.registeredRunways.remove(runway);
69                  this.availableRunways.remove(runway);
70              }
71          }
72
73          return unregisterOutcome;
74      }
75
76      @Override
77      public String showAvailableRunway() {
78
79          String runwayList = "no-one available runway";
80
81          if(!availableRunways.isEmpty()) {
82
83              runwayList = "|";
84
```

```
85              for(Runway runway : this.availableRunways) {
86                  runwayList = runwayList + runway.getRunwayName
                        () + "|";
87              }
88
89          }
90
91          return runwayList;
92      }
93
94      @Override
95      public String allotRunwayTo(Aircraft aircraft) {
96
97          String allotRunway = null;
98
99          if(!this.availableRunways.isEmpty()) {
100
101             for(Runway runway : this.availableRunways) {
102
103                 if(runway.getRunwayName().equals("Runway1") &&
104                         aircraft.getFlighID().substring(0,2).
                            equals("AL")) {
105
106                     //allot proper runway to scheduled flights
107
108                     allotRunway = runway.getRunwayName();
109                     this.mapRunwayAircraft.put(aircraft, runway
                        );
110                     this.availableRunways.remove(runway);
111
112                 }
113
114                 if(runway.getRunwayName().equals("Runway2") &&
115                         aircraft.getFlighID().substring(0,2).
                            equals("PJ")) {
116
117                     //allot proper runway to private jets
118
119                     allotRunway = runway.getRunwayName();
120                     this.mapRunwayAircraft.put(aircraft, runway
                        );
121                     this.availableRunways.remove(runway);
122
123                 }
124
125             }
126
127         }
128
129         if(!this.mapRunwayAircraft.containsKey(aircraft)) {
```

```
130
131             //there have to be a queue for some aircraft
132             enqueueAircraft(aircraft);
133         }
134
135         return allotRunway;
136     }
137
138     public void enqueueAircraft(Aircraft aircraft) {
139
140         for(Runway runway : this.registeredRunways) {
141
142             if(runway.getRunwayName().equals("Runway1")
143                     && aircraft.getFlighID().substring(0,2).
                        equals("AL")) {
144
145                 //scheduled flights have to be enqueue to the
                        Runway1
146
147                 this.mapRunwayAircraft.put(aircraft, runway);
148                 break;
149
150             }
151
152             if(runway.getRunwayName().equals("Runway2")
153                     && aircraft.getFlighID().substring(0,2).
                        equals("PJ")) {
154
155                 //private jets have to be enqueue to the
                        Runway2
156
157                 this.mapRunwayAircraft.put(aircraft, runway);
158                 break;
159
160             }
161
162         }
163
164     }
165
166     @Override
167     public String changeRunway(Aircraft aircraft) {
168
169         int scheduledFlightEnqueued = 0;
170         int privateJetsEnqueued = 0;
171
172         String newRunway = null;
173
174
```

```
175         for (Aircraft enqueuedAircraft : this.mapRunwayAircraft
                .keySet()) {
176
177             if (enqueuedAircraft.getFlighID().substring(0, 2).
                    equals("AL")
178                     && this.mapRunwayAircraft.get(
                            enqueuedAircraft).
179                     getRunwayName().equals("Runway1")) {
180
181                 //we have to count how many scheduled flights
                        are enqueued
182
183                 scheduledFlightEnqueued++;
184             }
185
186             if (enqueuedAircraft.getFlighID().substring(0, 2).
                    equals("PJ")
187                     && this.mapRunwayAircraft.get(
                            enqueuedAircraft).
188                     getRunwayName().equals("Runway2")) {
189
190                 //we have to count how many private jets are
                        enqueued
191
192                 privateJetsEnqueued++;
193             }
194
195         }
196
197         if (aircraft.getFlighID().substring(0, 2).equals("AL")
                &&
198                 scheduledFlightEnqueued > 1 &&
                        privateJetsEnqueued == 0) {
199
200             //scenario for changing runway for scheduled
                    flights
201
202             for (Runway runway : this.availableRunways) {
203                 if (runway.getRunwayName().equals("Runway2")) {
204                     this.mapRunwayAircraft.put(aircraft, runway
                            );
205                     this.availableRunways.remove(runway);
206                     newRunway = runway.getRunwayName();
207                 }
208             }
209
210         }
211
212         if (aircraft.getFlighID().substring(0, 2).equals("PJ")
                &&
```

```
213                 scheduledFlightEnqueued == 0 &&
                        privateJetsEnqueued > 1) {
214
215             //scenario for changing runway for private jets
216
217             for (Runway runway : this.availableRunways) {
218                 if (runway.getRunwayName().equals("Runway1")) {
219                     this.mapRunwayAircraft.put(aircraft, runway
                            );
220                     this.availableRunways.remove(runway);
221                     newRunway = runway.getRunwayName();
222                 }
223             }
224
225         }
226
227         return newRunway;
228     }
229
230     @Override
231     public String releaseRunwayOccupiedBy(Aircraft aircraft) {
232
233         String releasedRunway = null;
234
235         boolean alreadyAvailable = false;
236
237         for (Runway runway : this.availableRunways) {
238
239             if (runway.equals(this.mapRunwayAircraft.get(
                    aircraft))) {
240
241                 alreadyAvailable = true;
242
243             }
244
245         }
246
247         if (!alreadyAvailable) {
248             this.availableRunways.add(this.mapRunwayAircraft.
                    get(aircraft));
249         }
250
251         releasedRunway = this.mapRunwayAircraft.get(aircraft).
                getRunwayName();
252
253         return releasedRunway;
254     }
255
256 }
```

```
259 public class Aircraft extends ControlTower
260 {
261
262     private AirTrafficControllable controlTower;
263
264     public void taxxing() {
265
266
267         System.out.println("Available runways: " + this.
                controlTower.
268                     showAvailableRunway());
269
270         String allotAirstrip = this.controlTower.allotRunwayTo(
                this);
271
272         if(!Objects.isNull(allotAirstrip)) {
273
274
275             System.out.println(this.aircraftName + "'s [" +
                    this.flighID + "]" +
276                         " taxxing on the " + allotAirstrip);
277
278             System.out.println("Available runways: " + this.
                    controlTower.
279                         showAvailableRunway());
280
281
282         }else {
283
284             System.out.println(this.aircraftName + " [" + this.
                    flighID + "] can't taxi" );
285
286             System.out.println(this.aircraftName + " [" + this.
                    flighID + "] requests "
287                         + "the control tower for a runway change" )
                            ;
288
289             String newRunway = this.controlTower.changeRunway(
                    this);
290
291             if(!Objects.isNull(newRunway)) {
292
293                 System.out.println("Control tower confirms the
                        request and it allot "
294                 + newRunway + " to [" + this.flighID + "]");
295
296                 System.out.println("Available runways: " +
297                         this.controlTower.showAvailableRunway()
                            );
```

```
298
299             }else {
300
301                 System.out.println("Control tower denys the
                        requests because all the "
302                         + "runways are not available");
303
304             }
305
306
307         }
308
309     }
310
311     public void takeoff() {
312
313         String releasedAirstrip = this.controlTower.
                releaseRunwayOccupiedBy(this);
314
315         if(!Objects.isNull(releasedAirstrip)) {
316
317             System.out.println(this.aircraftName + " [" + this.
                    flighID + "] take off "
318                     + "correctly from " + releasedAirstrip);
319
320             System.out.println("Availble runways: " + this.
                    controlTower.showAvailableRunway());
321
322
323         }
324
325     }
326
327 }
```

Supponendo di aver un blocco di 4 veivoli, composto da 3 scheduled flights (voli di linea) e 1 private jet, che richiedono il rullaggio (taxi) necessario per il corretto decollo nel seguente ordine:

1. $scheduledfligth_1$
2. $scheduledfligth_2$
3. $privatejet_1$
4. $scheduledfligth_3$

Allora, implementando la logica presentata, vorremmo che il programma rispondesse come di seguito riportato.

```
Available runways: |Runway1|Runway2|
Airbus480's [AL-001] taxiing on the Runway1
Available runways: |Runway2|
************************************************************
************************************************************
Available runways: |Runway2|
Ryanair370 [AL-002] can't taxi
Ryanair370 [AL-002] requests the control tower for a runway
    change
Control tower confirms the request and it allots Runway2 to [AL
    -002]
Available runways: no-one available runway
************************************************************
************************************************************
Available runways: no-one available runway
TurboJet2.0 [PJ-001] can't taxi
TurboJet2.0 [PJ-001] requests the control tower for a runway
    change
Control tower denys the requests because all the runways are
    not available
************************************************************
************************************************************
Available runways: no-one available runway
EasyJet [AL-003] can't taxi
EasyJet [AL-003] requests the control tower for a runway change
Control tower denys the requests because all the runways are
    not available
************************************************************
************************************************************
Airbus480 [AL-001] take off correctly from Runway1
Availble runways: |Runway1|
************************************************************
************************************************************
Ryanair370 [AL-002] take off correctly from Runway2
Availble runways: |Runway1|Runway2|
************************************************************
************************************************************
TurboJet2.0 [PJ-001] take off correctly from Runway2
Availble runways: |Runway1|Runway2|
************************************************************
************************************************************
EasyJet [AL-003] take off correctly from Runway1
Availble runways: |Runway1|Runway2|
************************************************************
```

L'output ottenuto deriva da una serie di stampe a video riportate in una classe contenente il metodo principale ***public static void main(String[] args)*** e la sua funzione è prettamente dimostrativa. I reali test infatti vengono eseguiti

attraverso l'utilizzo del framework *JUnit4*. Di seguito si è riportato un frammento di codice caratterizzante di alcune classi di test. Si tenga presente che i test effettuati sono di natura funzionale, ossia cercano di verificare la logica presentata nella sezione 2 dell'elaborato.

```
371 public class TestLanaiAirPolicieLogic {
372
373
374     private AirTrafficControllable controlTower;
375
376     private Runway runway1;
377     private Runway runway2;
378
379     private Aircraft scheduledFlight1;
380     private Aircraft scheduledFlight2;
381     private Aircraft scheduledFlight3;
382     private Aircraft pjet1;
383
384     @Before
385     public void setUp() throws Exception {
386
387         controlTower = new LanaiAirPolicie("LANAI-airport");
388
389         runway1 = new Runway("Runway1", 2500);
390         runway2 = new Runway("Runway2", 2000);
391
392         scheduledFlight1 = new Aircraft(controlTower, "AL-001",
                "Airbus480", 200);
393         scheduledFlight2 = new Aircraft(controlTower, "AL-002",
                "Ryanair370", 150);
394         scheduledFlight3 = new Aircraft(controlTower, "AL-003",
                "EasyJet", 130);
395
396         pjet1 = new Aircraft(controlTower, "PJ-001", "TurboJet2
                .0", 20);
397         assertNotNull(controlTower);
398
399         assertNotNull(runway1);
400         assertNotNull(runway2);
401
402         assertNotNull(scheduledFlight1);
403         assertNotNull(scheduledFlight2);
404         assertNotNull(scheduledFlight3);
405         assertNotNull(pjet1);
406
407         assertEquals(true, controlTower.registerRunway(runway1)
                );
408
409         assertEquals("|Runway1|", controlTower.
```

```
            showAvailableRunway());
410
411         assertEquals(true, controlTower.registerRunway(runway2)
                );
412
413         assertEquals("|Runway1|Runway2|", controlTower.
                showAvailableRunway());
414
415
416     }
417
418     @Test
419     public void testFirstBlockAircraft() {
420
421         //We suppose that the first block of aircrafts arrive
422             //with the following order
423         // - scheduled flight_1
424         // - scheduled flight_2
425         // - private jet_1
426         // - scheduledflight_3
427
428
429         //-------------------------------------------------
430         //taxiing
431
432         assertEquals("|Runway1|Runway2|", controlTower.
                showAvailableRunway());
433
434         assertEquals("Runway1", controlTower.allotRunwayTo(
                scheduledFlight1));
435
436         assertEquals("|Runway2|", controlTower.
                showAvailableRunway());
437
438         assertEquals(null, controlTower.allotRunwayTo(
                scheduledFlight2));
439
440         assertEquals("Runway2", controlTower.changeRunway(
                scheduledFlight2));
441
442         assertEquals("no-one available runway", controlTower.
                showAvailableRunway());
443
444         assertEquals(null, controlTower.allotRunwayTo(pjet1));
445
446         assertEquals(null, controlTower.changeRunway(pjet1));
447
448         assertEquals(null, controlTower.allotRunwayTo(
                scheduledFlight3));
449
```

```
450         assertEquals(null, controlTower.changeRunway(
                scheduledFlight3));
451
452         //-------------------------------------------------
453
454         //-------------------------------------------------
455         //take off
456
457         assertEquals("Runway1", controlTower.
                releaseRunwayOccupiedBy(scheduledFlight1));
458
459         assertEquals("|Runway1|", controlTower.
                showAvailableRunway());
460
461         assertEquals("Runway2", controlTower.
                releaseRunwayOccupiedBy(scheduledFlight2));
462
463         assertEquals("|Runway1|Runway2|", controlTower.
                showAvailableRunway());
464
465         assertEquals("Runway2", controlTower.
                releaseRunwayOccupiedBy(pjet1));
466
467         assertEquals("Runway1", controlTower.
                releaseRunwayOccupiedBy(scheduledFlight3));
468
469         //-------------------------------------------------
470     }
471
472 }
```

Di seguito si è riportato lo *Use Case Diagram* associato al programma con l'intento di catturare ulteriormente il comportamento atteso dal sistema in termini di servizi, compiti e funzioni, garantendo una visione d'insieme relazionale tra "attori" e obiettivi. Nel diagramma è presente anche un caso d'uso, non esplicitamente richiesto nel testo del modello, che riguarda la gestione delle piste di decollo da parte dell'aereoporto.

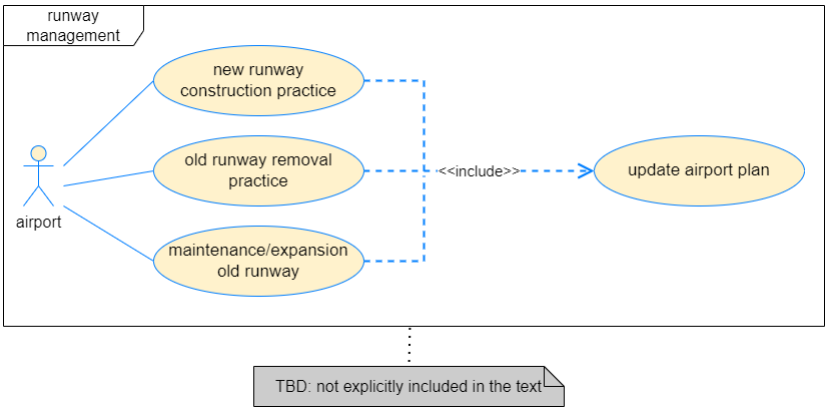

Figure 8: Use case management runways, high summary abstraction level

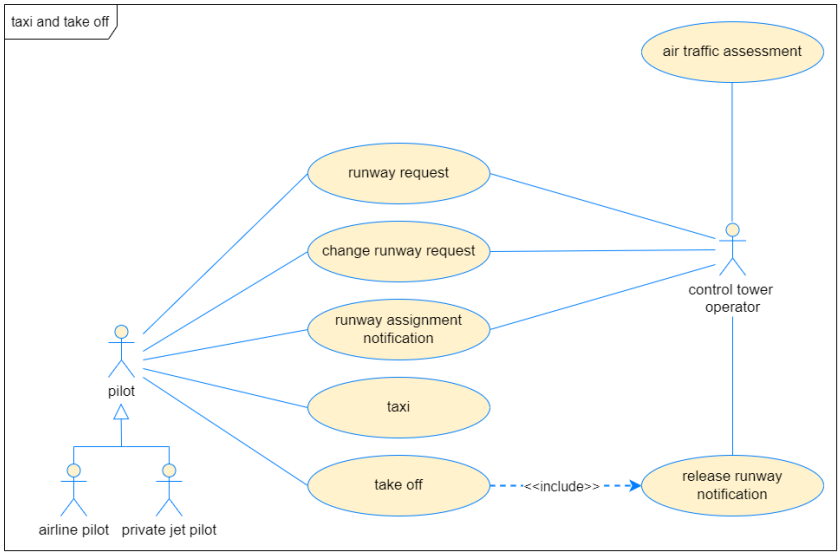

Figure 9: Use case taxi and take off, high summary abstraction level

L'*Activity Diagram* che segue è riferito solo al caso d'uso rappresentato in **Figure 9**.

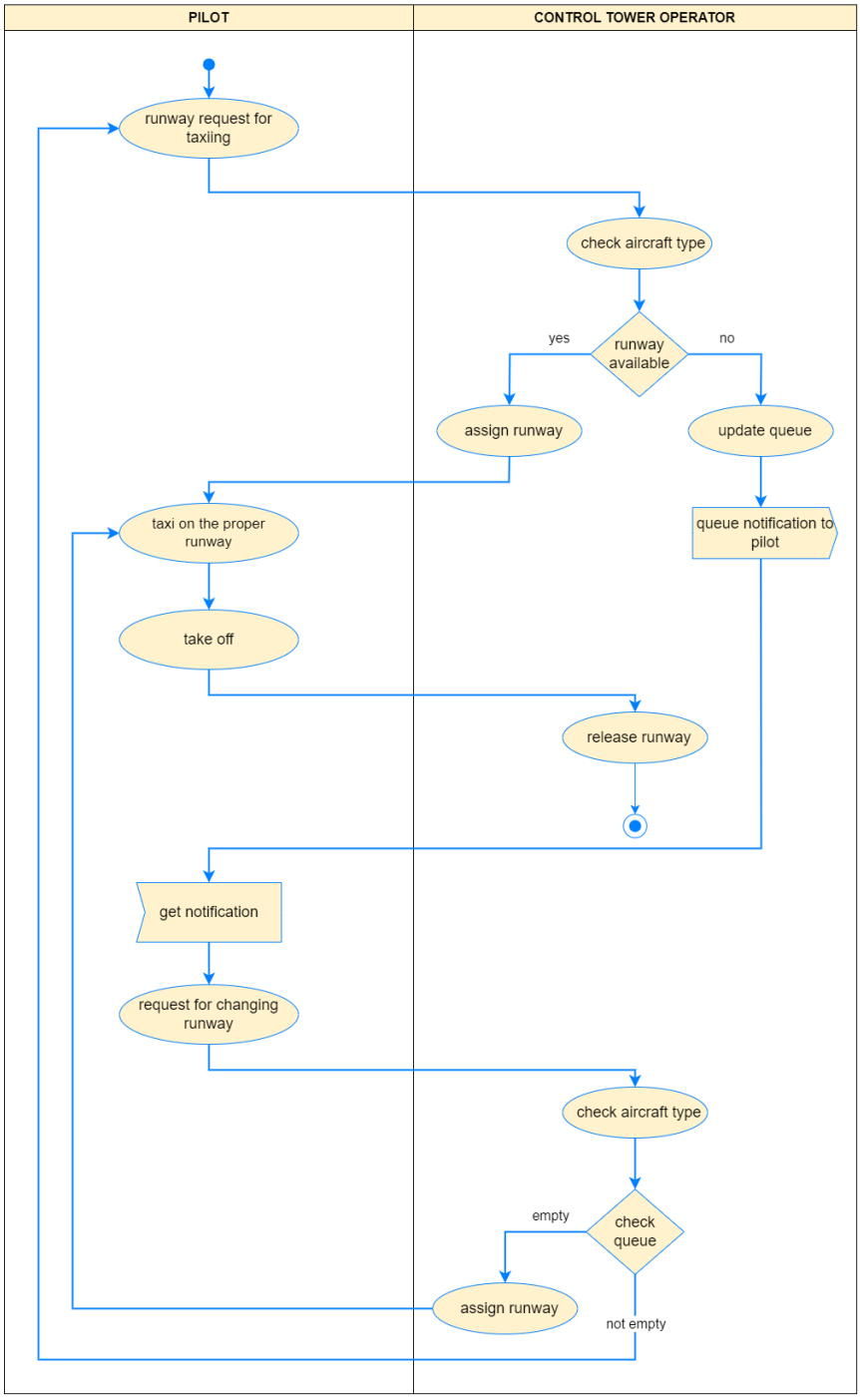

Figure 10: Activity Diagram

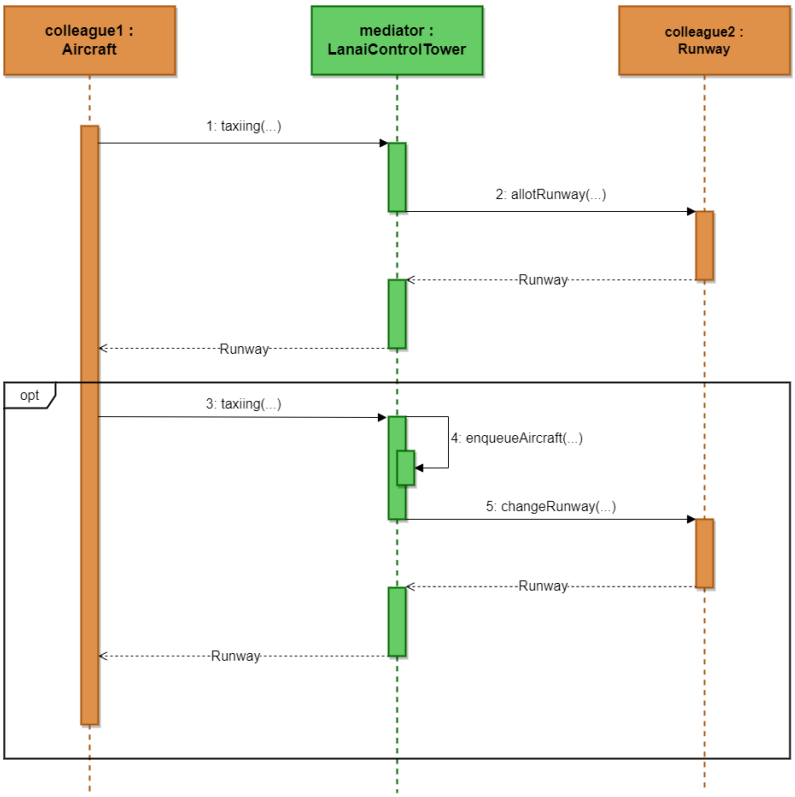

Figure 11: Sequence Diagram

*Note: Il sequence diagram riportato in **Figure 11** presenta un frame denominato "opt" la quale funzione è quella di indicare una particolare sequenza che si verificherà solo sotto certe condizioni. Modella in un certo senso la logica "se poi...".*

5 Conclusioni

In conclusione, l'elaborato ha fornito un'approfondita analisi e progettazione di un modello di dominio basato su un pattern specificamente scelto per affrontare uno scenario reale rappresentativo. Durante l'analisi, sono stati evidenziati chiaramente i vantaggi del pattern selezionato in relazione al contesto da modellare, sottolineando l'importanza degli strumenti di analisi utilizzati in fase di progettazione.

La progettazione della struttura del modello è stata eseguita con la massima attenzione, mettendo in luce i partecipanti chiave e le relazioni che sussistono tra di essi. Questa fase di progettazione ha gettato le basi per l'implementazione pratica del modello.

Nella sezione dedicata all'implementazione, sono stati forniti frammenti di codice che illustrano in modo chiaro e dettagliato gli aspetti salienti dello schema. Inoltre, sono stati definiti casi di test utilizzando JUnit per garantire che il modello funzioni correttamente in uno scenario che ne caratterizza l'intento.

In sintesi, questo elaborato ha dimostrato una solida comprensione del modello di dominio scelto, evidenziando come esso sia applicabile in un contesto reale. L'approfondita analisi, la meticolosa progettazione e l'implementazione accurata con casi di test forniscono una base solida per la futura applicazione di questo modello, contribuendo al progresso e alla comprensione del dominio trattato. Inoltre, l'elaborato sottolinea l'importanza di un approccio strutturato e ben pianificato nella progettazione di modelli di dominio complessi. La chiara identificazione dei partecipanti e delle relazioni, insieme all'utilizzo di strumenti come JUnit per la verifica, dimostra un impegno verso la qualità e l'affidabilità del modello.

Da un punto di vista più ampio, questo lavoro fornisce un contributo significativo alla comprensione del dominio specifico trattato. Gli aspetti analitici evidenziati nell'analisi complessiva suggeriscono che il pattern selezionato è appropriato per risolvere le sfide specifiche del contesto, aprendo la strada a futuri sviluppi e applicazioni nel settore.

In conclusione, l'elaborato rappresenta un esempio eccellente di come l'analisi, la progettazione e l'implementazione accurata di un modello di dominio possano portare a soluzioni robuste e efficaci per problemi complessi. Questo lavoro costituirà una risorsa preziosa per chiunque sia interessato a esplorare ulteriormente il dominio e implementare soluzioni basate su questo pattern, contribuendo così al progresso nella comprensione e nell'applicazione di questo importante campo.

6 Note e osservazioni del docente

* La struttura UML del programma non ricalca dettagliatamente quella del pattern Mediator presentata. In particolare la dipendenza tra la classe **Aircraft** e **LanaiAirPolicie** rende "inutile" l'utilizzo dell'interfaccia. Decisamente più interessante sarebbe stato qualora ci fosse una correlazione tra **Aircraft** e **AirTrafficControllable**. Non è stato dunque implementato correttamente il pattern Mediator, quanto una sua revisione modificata.

* La classe astratta **ControlTower** può essere sostituita con una default interface.

* I test dovrebbero essere efficaci per collaudare la strutturalità del contesto da modellare, oltre che le funzionalità offerte. Quanto meno devono essere maggiormente dettagliati nella stesura dell'elaborato.

* L'elaborato ha funzione di documentare il lavoro svolto: consiglio quindi l'utilizzo di template, mock-up, commenti in fase di trascrizione del codice, oltre che tutti gli altri diagrammi correttamente definiti.

* Estensione del modello: sottolineando che per l'esame di SWE il progetto si presenta coerente con le linee guida, maggior interesse avrebbe lo studio della comunicazione dei diversi software che compongono il sistema (quindi le relazioni che sussistono tra i codici degli oggetti di tipo *Aircraft*, *Runway*, *ControlTower*, ecc...).