

# Museo automatizzato

Giuseppe Parrotta

Elaborato per il corso di  
Ingegneria del Software



UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

Dipartimento di Ingegneria Informatica  
Università degli Studi di Firenze  
28/04/2021

# Indice

<b>I</b>	<b>Progettazione</b>	<b>3</b>
1	Idea di progetto e strumenti per la sua realizzazione	3
2	Definizione del problema	4
3	Manufatti	5
3.1	Class Diagram concettuale del Modello di Dominio . . . . .	5
3.2	Package Diagram . . . . .	6
3.3	Use Case Diagram . . . . .	7
3.4	Use Case Templates . . . . .	8
3.5	Mockups . . . . .	9
3.5.1	Pannello dell'amministratore . . . . .	9
3.5.2	Pannello Impiegato e Organizzatore . . . . .	10
3.5.3	Pannello Visitatori . . . . .	11
3.6	Activity Diagram . . . . .	13
<b>II</b>	<b>Implementazione</b>	<b>14</b>
<b>4</b>	<b>Package museoAndAdmin</b>	<b>15</b>
4.1	Gruppo classi Museo . . . . .	16
4.1.1	Classi SalaFisica e SalaVirtuale. Classe Suggerimento . .	16
4.1.2	Interfaccia Ticket . . . . .	16
4.1.3	Classe Mostra . . . . .	16
4.1.4	Classe Museo . . . . .	17
4.2	Gruppo classi Amministratore . . . . .	22
4.2.1	Classe IncaricoMostra . . . . .	22
4.3	Strategie . . . . .	23
4.3.1	Strategie creazionali . . . . .	23
4.3.2	Strategie non creazionali . . . . .	24
4.4	Classe Amministratore . . . . .	26
4.4.1	Metodi a supporto di update . . . . .	27
4.4.2	Il metodo update . . . . .	30
<b>5</b>	<b>Package organizzazione</b>	<b>34</b>
5.1	Classe Organizzatore . . . . .	35
5.2	Altre classi . . . . .	36
<b>6</b>	<b>Altri packages</b>	<b>38</b>
6.1	Package opera . . . . .	38
6.1.1	Classe Opera . . . . .	38
6.1.2	Classe GestoreOpere . . . . .	39

6.2 Package Visitori . . . . .	41
<b>III Test</b>	<b>42</b>
7 Test noleggio Opere	43
8 Test strategies	44
9 Test suggerimento	45
10 Test di kill e di rimborso	45
11 Test creazione mostre	47
12 Test vendita ticket	47
13 Test Observer	48
14 Conclusione test	50

## Part I

# Progettazione

## 1 Idea di progetto e strumenti per la sua realizzazione

L'elaborato consiste nella realizzazione tramite linguaggio Java di un modello di dominio auto-assegnato.

L'idea è quella di realizzare un'applicazione che permetta la gestione di un museo; in particolare, che permetta al suo amministratore di allestire delle Mostre di opere d'Arte, le quali possono essere, eventualmente, anche visibili tramite un apposito portale web.

Per la progettazione del software sono stati impiegati i seguenti strumenti: Class Diagram, Sequence diagram, Use Case Diagram, Activity diagram e Mock-ups. È stato necessario implementare i design patterns Observer, Strategy e Singleton.

Per scrivere il codice, si è usato IntelliJ IDEA. Il codice è stato testato usando il framework Junit 5.4.

## 2 Definizione del problema

Si vuole realizzare un'applicazione che permetta all'Amministratore di un Museo di organizzare delle Mostre. In particolare, deve essere in grado di prendere le seguenti decisioni anche in maniera automatica: *chiudere* una mostra o *aprirne* una nuova, entrambe in funzione dello stato del museo. L'Amministratore può decidere di noleggiare alcune Opere d'arte da altri musei col fine di inserirle in una Mostra, ammesso che quelle opere non risultino già affittate. Dopodiché ordinerà a un impiegato Organizzatore di realizzare la Mostra, inserendo le Opere d'Arte da lui scelte.

L'Organizzatore avrà il compito di selezionare le Sale necessarie per creare la Mostra, e di scegliere un gruppo di Impiegati che dovranno svolgere i seguenti incarichi: pulizia delle Sale Fisiche, spostare l'Opera d'Arte, creare Pubblicità e nel caso in cui la Mostra sia anche Virtuale, l'allestimento del Portale Web.

La Mostra, oltre a dover essere Fisica, e quindi allestita in alcune Sale del Museo, può essere Virtuale, quindi fruibile anche dal portale online: la sala virtuale ricrea una proiezione online della sala fisica. Il numero di posti in questo caso aumenta, in quanto oltre ai posti previsti dalle varie sale fisiche vanno aggiunti anche quelli previsti dalle Sale Virtuali. È richiesto che per ogni opera d'Arte prevista da una mostra, gli si debba allocare una sala fisica del Museo.

La Mostra deve essere dichiarata conclusa quando raggiunge la quota di incassi prevista dall'Organizzatore, oppure quando l'Amministratore lo richiede. In questo caso, l'Organizzatore rimborserà i Ticket venduti, meno una trattenuta in percentuale. Infine, le eventuali opere noleggiate andranno restituite al proprietario. L'Amministratore può chiedere la chiusura di una Mostra quando nota che le sale sono quasi tutte occupate.

I Visitatori possono essere registrati, o non registrati: questi ultimi, previo pagamento, possono sia entrare nel Museo, che visitare le Mostre, ammesso che siano rimasti posti dentro le Sale Fisiche. I Visitatori Registrati hanno in più la possibilità di acquistare anche posti nelle Sale Virtuali. Inoltre, ogni Visitatore può registrarsi al Museo, diventando un Visitatore Registrato.

Per Impiegati, Organizzatori o Visitatori, è possibile consultare il catalogo delle Opere e lasciare suggerimenti su quali opere portare per la prossima Mostra.

### 3.1 Class Diagram concettuale del Modello di Dominio

### 3.1 Class Diagram concettuale del Modello di Dominio



### 3.2 Package Diagram

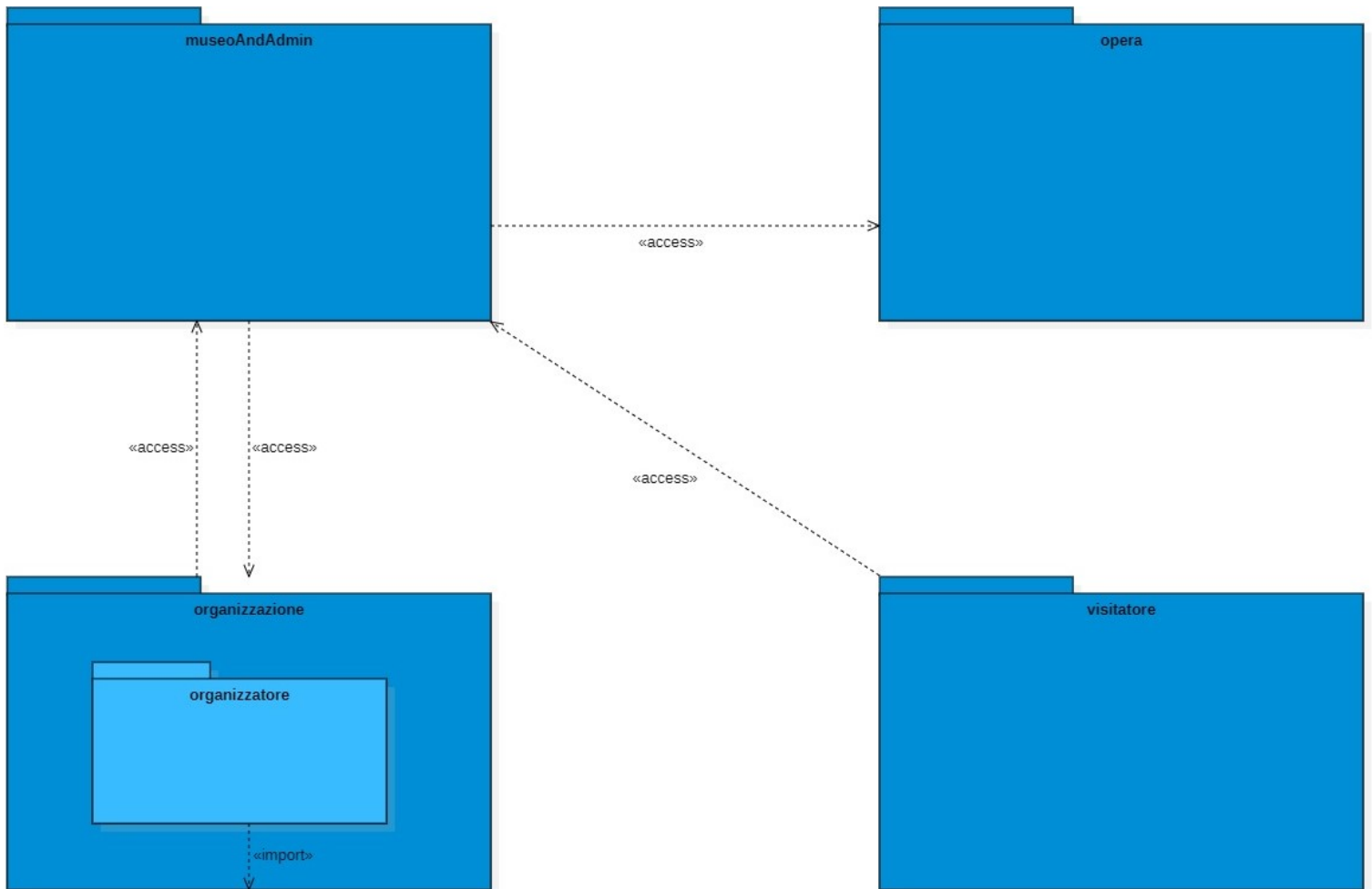


Figure 2: Package diagram

### 3.3 Use Case Diagram

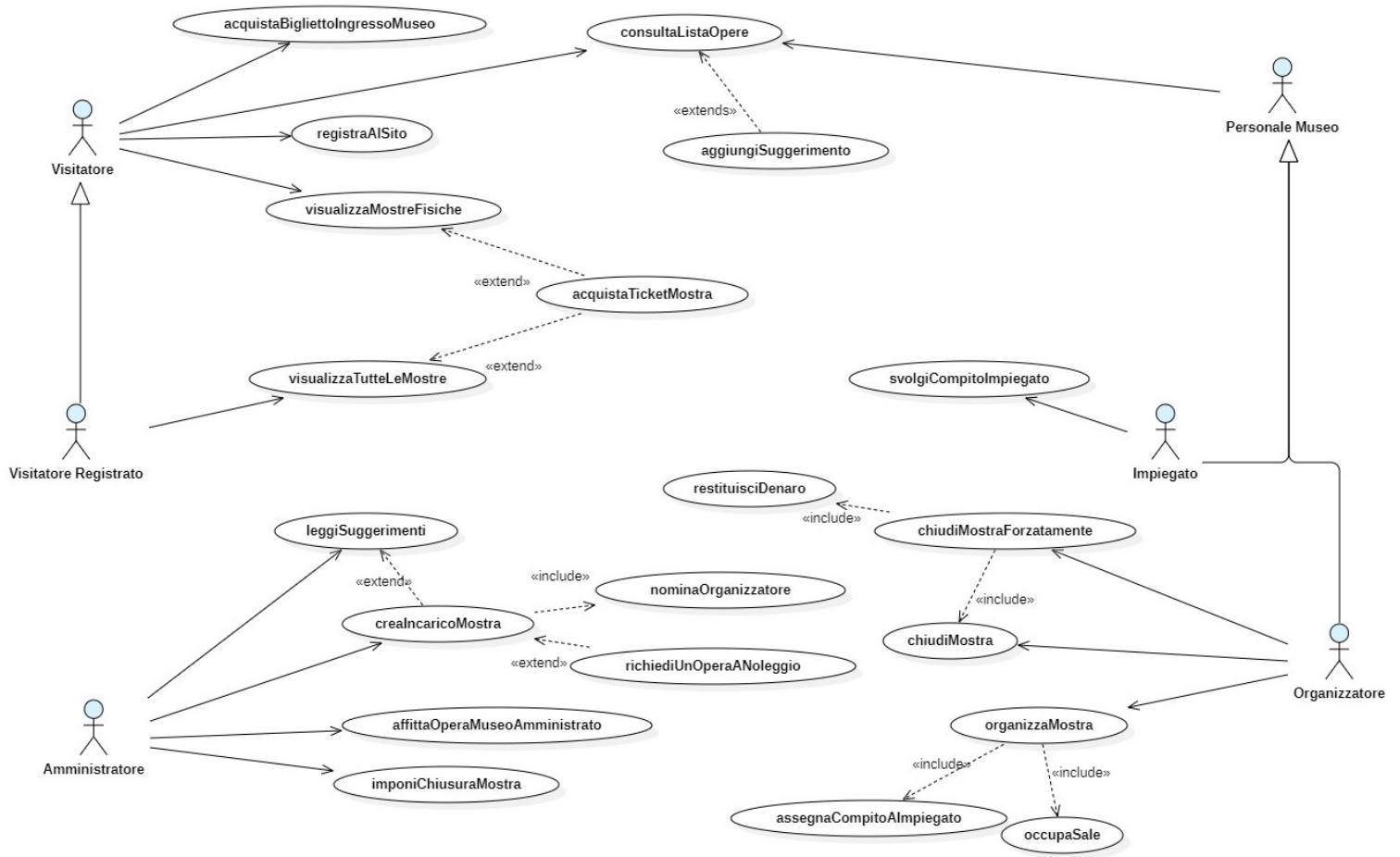


Figure 3: Use Case Diagram



### 3.4 Use Case Templates

<u>Name</u>	UCAdmin#2 - Crea incarico Mostra
<u>History</u>	
<u>Source</u>	
<u>Level</u>	Users goal
<u>Description</u>	L'amministratore del Museo crea un incarico per una mostra
<u>Scope</u>	Creazione Mostra
<u>Actors</u>	Amministratore (Primary), Museo (Secondary)
<u>Pre - Conditions</u>	L'utente ha fatto l'accesso al sistema come amministratore
<u>Post- Conditions</u>	Viene emesso un incarico e fornito a un Organizzatore
<u>Normal flow</u>	0 L'utente accede al pannello amministratore 1 L'utente legge nel pannello la quantità di denaro disponibile 2 Extends: l'utente può scegliere se consultare i suggerimenti 3 L'utente sceglie di avviare una creazione 4 Il sistema mostra il pannello di creazione 5 L'utente seleziona opere e organizzatore e conferma 6 Extends: Il sistema chiede conferma per il noleggio di opere 7 Il sistema manda notifica all'organizzatore nominato
<u>Alternative flows</u>	6b L'utente cambia opere
<u>References</u>	UCAdmin#3 – Nomina Organizzatore
<u>Non functional requirements</u>	

<u>Name</u>	UCOrganiz#3 - Organizza Mostra
<u>History</u>	
<u>Source</u>	
<u>Level</u>	Users goal
<u>Description</u>	L'organizzatore sviluppa l'incarico ricevuto
<u>Scope</u>	Creazione Mostra
<u>Actors</u>	Organizzatore (primary), Museo(secondary)
<u>Pre - Conditions</u>	L'amministratore ha creato l'incarico
<u>Post- Conditions</u>	Il Museo dispone di una mostra in più
<u>Normal flow</u>	0 L'organizzatore accede al pannello organizzatore 1 Il sistema notifica l'incarico 2 L'utente legge la notifica 3 L'utente apre il pannello per la creazione 4 Il sistema mostra quali sono le sale disponibili del museo 5 Il sistema mostra quali sono gli impiegati disponibili 6 L'utente seleziona impiegati e imposta loro un ruolo 7 L'utente sceglie le sale del museo 8 Il sistema chiede conferma 9 Il sistema crea la mostra
<u>Alternative flows</u>	8b L'utente annulla 9b Torna al punto 4
<u>References</u>	UCOrganizz#4 -Richiedi Impiegati UCOrganizz#5 - Occupa Sale
<u>Non functional requirements</u>	

Figure 4: Use Case Templates di due casi d'uso

## 3.5 Mockups

### 3.5.1 Pannello dell'amministratore

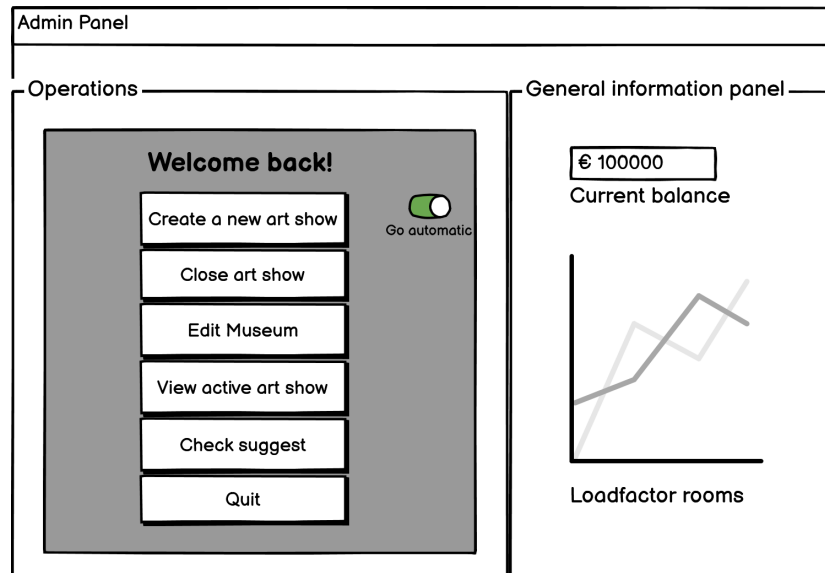


Figure 5: Admin Panel

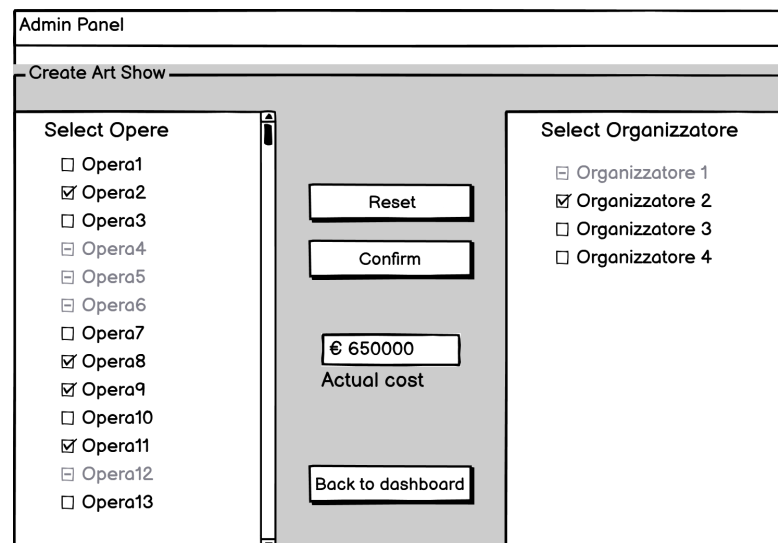


Figure 6: Creazione Mostra

### 3.5.2 Pannello Impiegato e Organizzatore

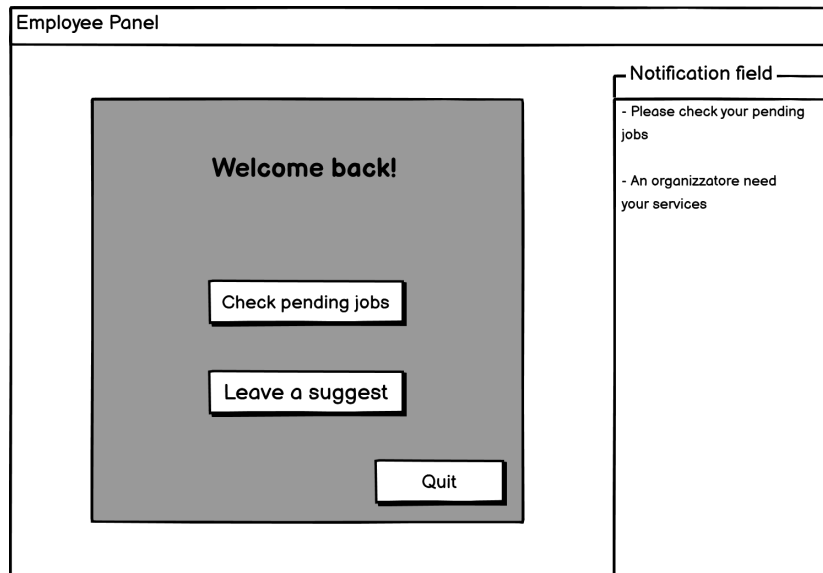


Figure 7: Dashboard Impiegato

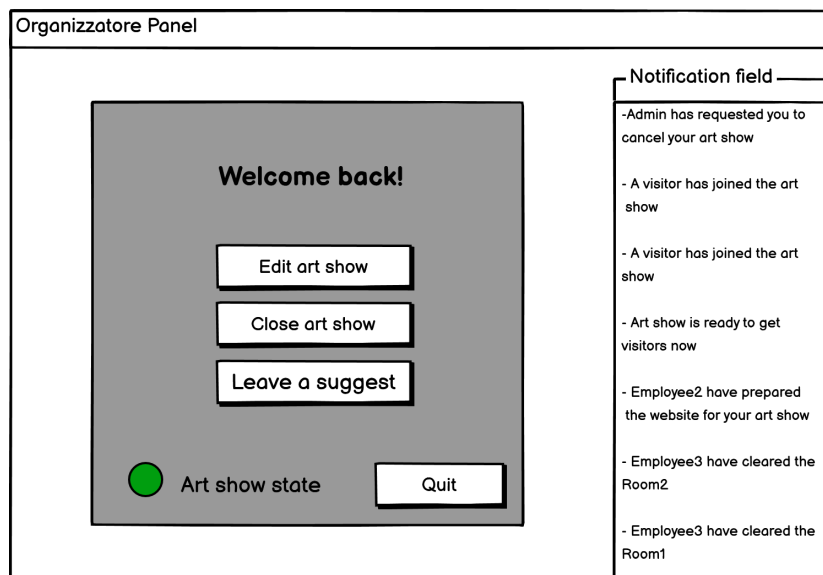


Figure 8: Dashboard Organizzatore

Organizzatore Panel

Create Opera

Select Employees

☒ Employee 1
 

Select role ▼

☒ Employee 2
 

Select role ▼

☒ Employee 3
 

Select role ▼

☐ Employee 4

☐ Employee 5

☒ Employee 6
 

Select role ▼

☐ Employee 7

☐ Employee 8

☐ Employee 9

☐ Employee 10

☐ Employee 11

☐ Employee 12

☐ Employee 13

Reset

Confirm

€ 30000

Allocated funds

Back to dashboard

Select Rooms

☐ Room 1

☒ Room 2

☐ Room 3

☐ Room 4

☒ Room 3

☒ Room 4

☐ Virtual Room 1

☐ Virtual Room 2

☒ Virtual Room 3

☐ Virtual Room 4

☐ Virtual Room 5

☐ Virtual Room 6

☐ Virtual Room 7

Figure 9: Realizzazione Mostra

### 3.5.3 Pannello Visitatori

Visitor Panel

Welcome to Museum 2.0!

Enter as a visitor

Login

Register

Leave a suggest

Figure 10: Pannello visitatori

Visitor Unregistered Panel

Please choose an art show:

☒ Art show 1  
☐ Art show 2  
☐ Art show 3  
☐ Art show 4

Show detail

Confirm

Back

Figure 11: Pannello scelta Mostra

Visitor Registered Panel

Please choose an art show:

☒ Art show 1  
☐ Art show 2  
☐ Art show 3  
☐ Art show 4

Show detail

Confirm

Log out

Figure 12: Pannello scelta Mostra per visitatori registrati

### 3.6 Activity Diagram

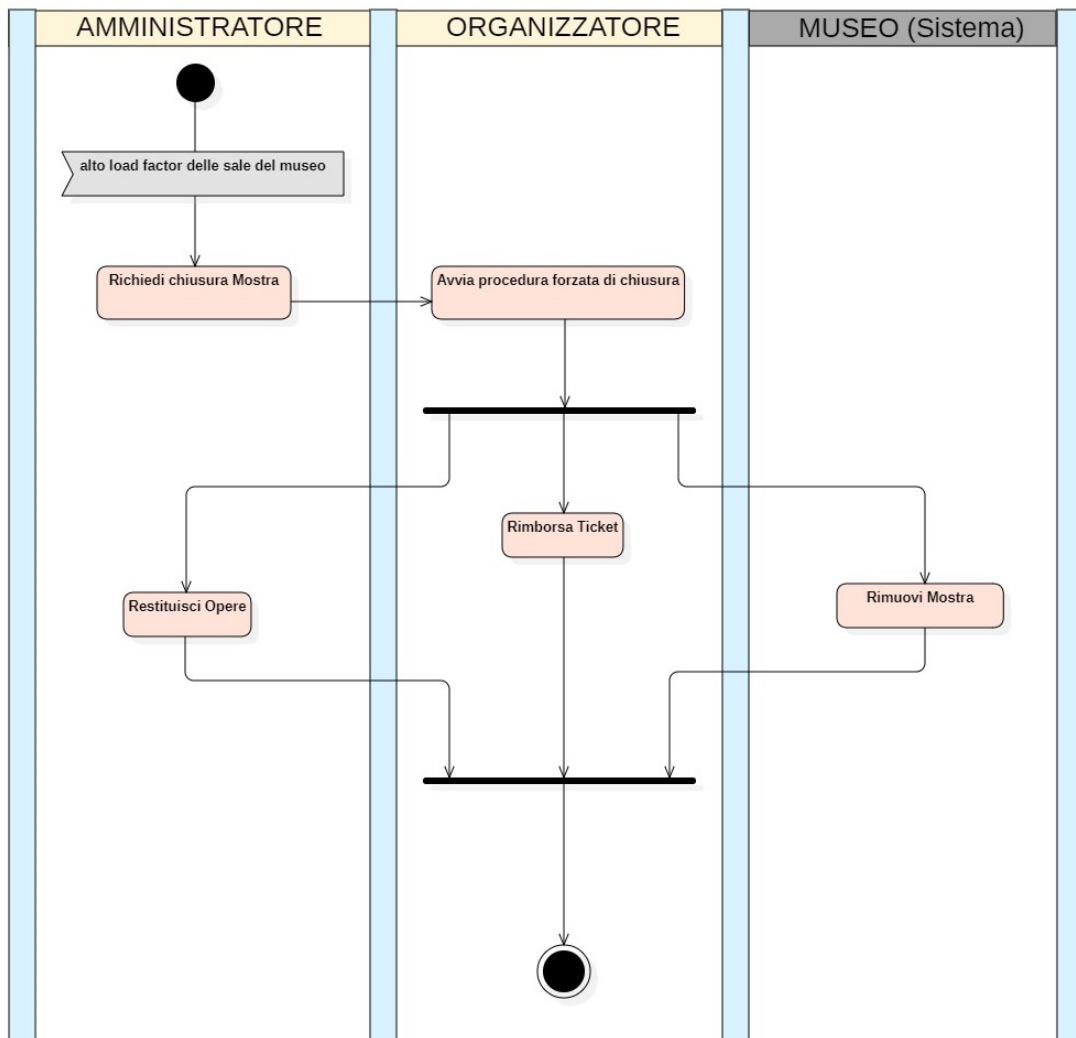


Figure 13: Activity diagram della procedura di chiusura forzata di una mostra

## Part II

# Implementazione

Le classi sono divise in vari package, per come visto nel package diagram. Quello più grande, *strutturaMuseo*, è suddiviso in 2 più piccoli: *organizzazione* e *museoAndAdmin*. Gli altri due *visitatore* e *opera*, sono invece a sé stanti.

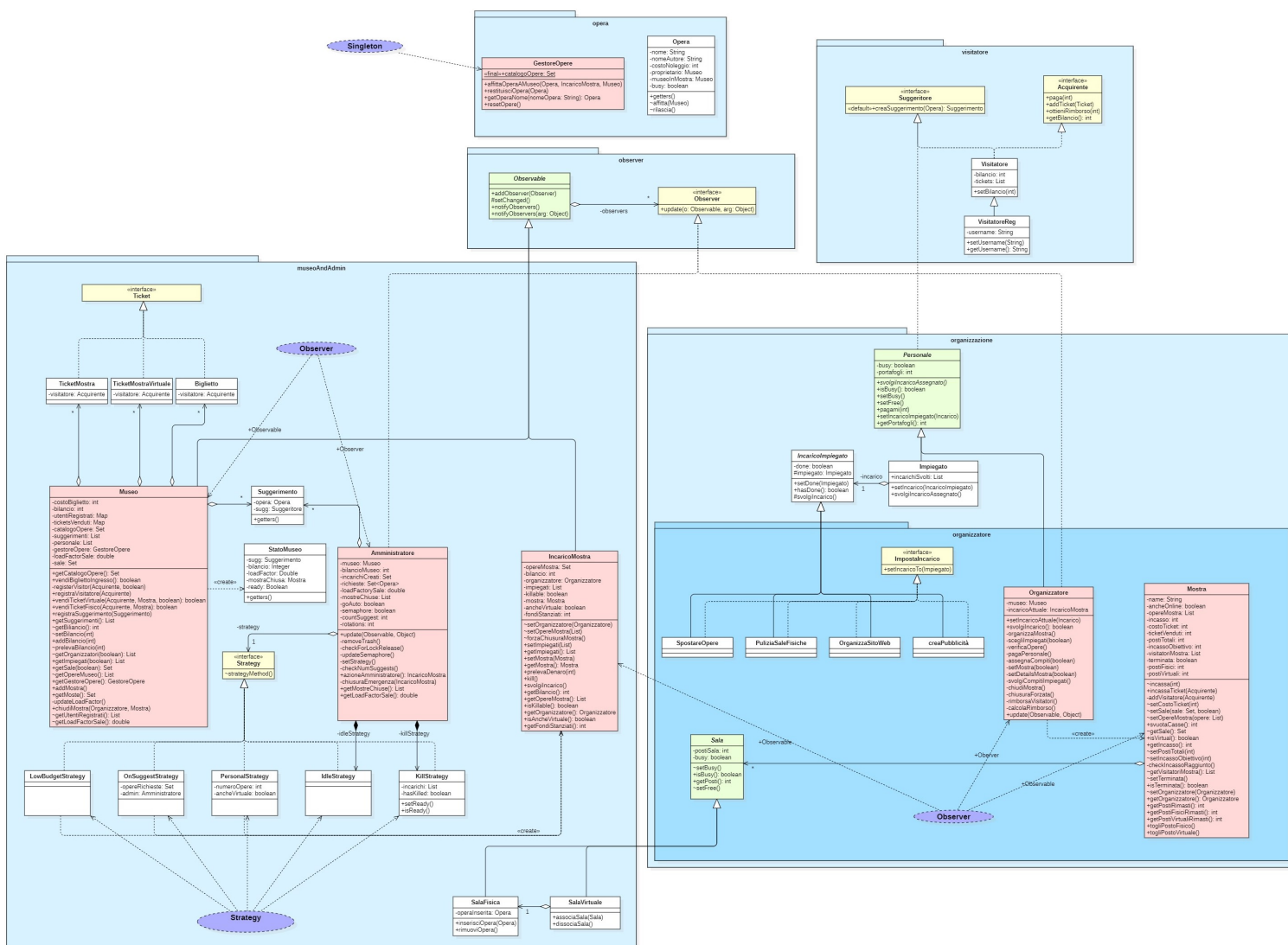


Figure 14: Class Diagram.

## 4 Package museoAndAdmin

Qui dentro sono presenti due gruppi concettuali di classi: quelle relative al **Museo** e quelle relative all'**Amministratore**.

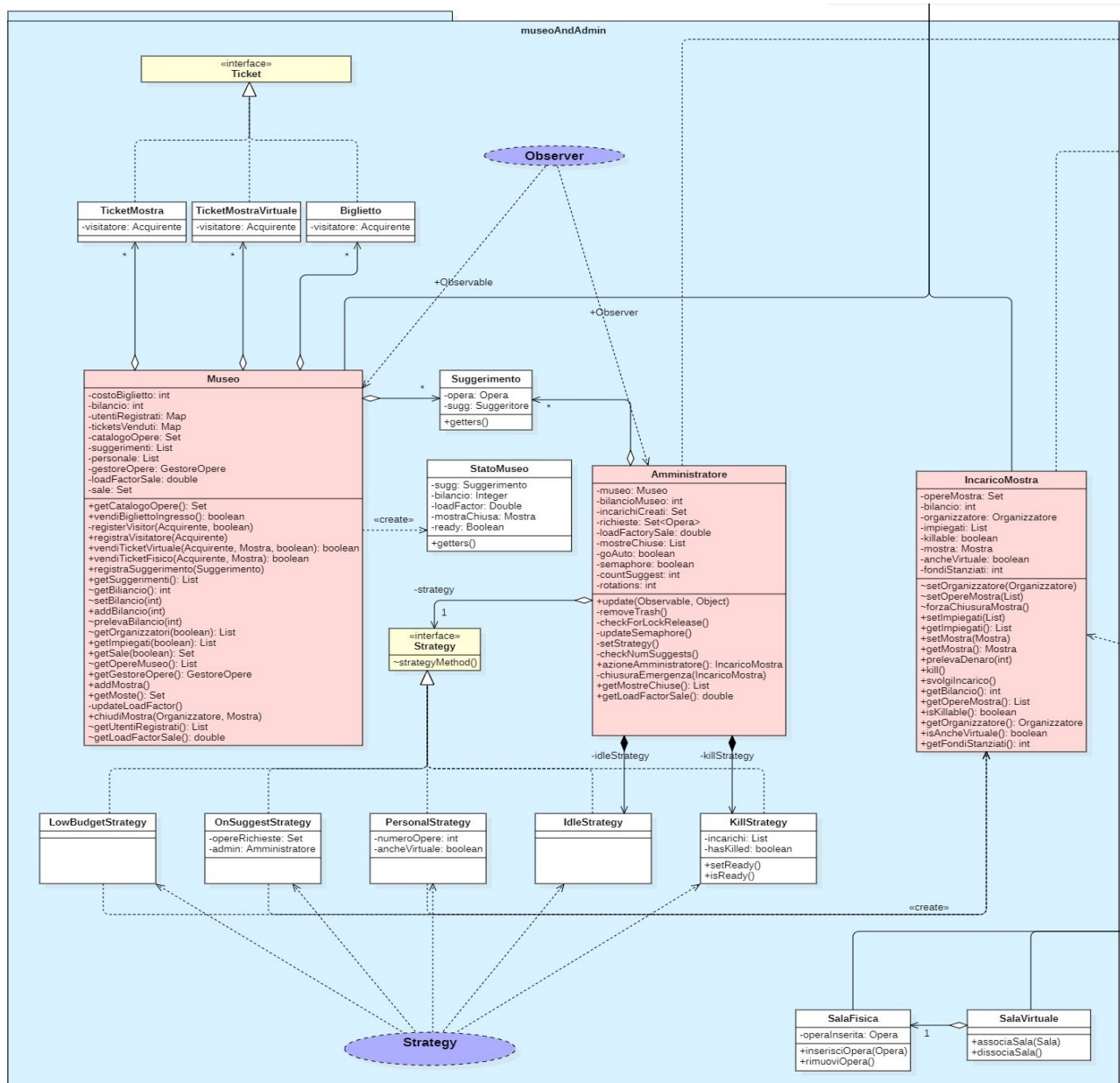


Figure 15: Package museoAndAdmin



## 4.1 Gruppo classi Museo

La classe *Museo* ha un ruolo centrale: tutte le classi in questo gruppo le forniscono un servizio. Infatti oggetti di tipo *Tickets*, *Sale*, *Mostra* e *Suggerimento* hanno uno spazio più o meno di rilievo nella struttura di *Museo*.

### 4.1.1 Classi *SalaFisica* e *SalaVirtuale*. Classe *Suggerimento*

Le classi *SalaVirtuale* e *SalaFisica* sono entrambe specializzazioni della classe astratta *Sala*, la quale è però appartenente al package *Organizzazione* visto che deve essere l'Organizzatore a doverne variare lo stato). Esse vengono create dal costruttore di *Museo* e inserite in un insieme (*Set*) immutabile di oggetti. Delle sale, vengono passati i riferimenti direttamente agli organizzatori, ma solo per cambiarne lo stato da libero a occupato, o vice versa. Infine, i costruttori per i due tipi di sale, hanno visibilità *package-protected*. Riassumendo: *Museo* crea le *Sale*, e l'Organizzatore ne cambia lo stato.

Gli oggetti di classe *Suggerimento* sono degli oggetti che hanno due riferimenti: uno a *Opera*, e l'altro ad un'implementazione di *Suggeritore*. Questi due riferimenti sono passati al costruttore (anch'esso *package-protected*). Una volta creati, vengono inseriti e accumulati dentro una struttura dati di tipo *ArrayList* di *Museo*.

### 4.1.2 Interfaccia *Ticket*

Implementazioni di questa interfaccia sono le classi *TicketMostraFisica*, *TicketMostraFisicaEVirtuale* e *Biglietto*. I costruttori di questi oggetti sono tutti *package-protected*, e l'unico che li usa è sempre *Museo*, il quale dispone di una struttura dati del tipo *LinkedHashMap<String,ArrayList<Ticket>>* per salvarli. Inoltre, il *Visitatore* che ha acquistato il ticket salva il riferimento all'oggetto in una sua struttura dati.

### 4.1.3 Classe *Mostra*

Contiene tutti i metodi getter e setter che servono a *Museo* ed a *Organizzatore*; è composta da field quali l'incasso, i posti rimasti (*int*), una lista di visitatori e il *Set* di *Sale* che gli sono state assegnate. Inoltre estende *Observable*, in quanto deve comunicare all'*Organizzatore* (che implementa *Observer*) quando raggiunge il numero massimo di posti occupati:

```
1 public class Mostra{
2     ...
3     public void incassaTicket(Acquirente v) {
4         ticketVenduti++;
5         this.incasso += this.costoTicket;
6         addVisitatore(v);
7         checkRaggiuntoIncasso();
8     }
9     private void checkRaggiuntoIncasso(){
```

```

10     if(incasso >= incassoObiettivo) {
11         setChanged();
12         notifyObservers();
13     }
14 }
15 ...
16 }

```

Listing 1: Metodi incassaTicket e checkRaggiuntoIncasso

#### 4.1.4 Classe Museo

La classe *Museo* ha tanti compiti: essa deve gestire il pagamento di *Tickets*, fornire i riferimenti alle Sale, alle Opere, agli Impiegati e agli Organizzatori, elencare le Mostre che sono attive, registrare eventuali visitatori e gestire i suggerimenti. Ha inoltre il compito di comunicare con l'*Amministratore* il suo stato: Museo estende la classe astratta *Observable*, e *Amministratore* implementa *Observer*. Questi due oggetti comunicano sia attraverso un Observer di tipo Pull, che con uno di tipo Push.

Per rendere più chiara la lettura, divido concettualmente il Museo in componenti, intesi come gruppi di metodi della classe.

#### Componente rivolta agli Acquirenti e Suggestori

Ogni *Visitatore* è implementazione delle interfacce *Acquirente* e *Suggestore*, e i vari Impiegati sono implementazioni di *Suggestore*.

Fanno parte di questo gruppo quei metodi che permettono l'**acquisto di Ticket**, quello che permette la **consultazione del Catalogo delle opere**, quelli che permettono la **registrazione degli Acquirenti**, e infine quello che permette di **consultare le mostre attive**.

È stata realizzata l'eccezione *NoMoneyException*, specializzazione di *RuntimeException*, ed è una eccezione che viene lanciata quando non si può effettuare una transazione.

I metodi per l'acquisto sono:

- vendiBigliettoMuseo(Acquirente)
- vendiTicketMostraFisica(Acquirente, Mostra
- vendiTicketMostraVirtuale(Acquirente, Mostra, boolean

Il secondo di questi metodi fa forwarding sul terzo.

```

1 public class Museo{
2     ...
3     public boolean vendiBigliettoMuseo(Acquirente visitatore) {
4         try{
5             visitatore.paga(costoBiglietto);
6             Ticket tmp = new Biglietto(visitatore);
7             ticketMuseoVenduti.get("Biglietto_Base").add(tmp);

```

```

8      visitatore.addTicket(tmp);
9      registerVisitor(visitatore, true);
10     return true;
11 } catch (NoMoneyException e) {
12     System.err.println(e.getMessage());
13     return false;
14 }
15 }
16
17 }

```

Listing 2: Metodo vendiBigliettoMuseo

Nel blocco try-catch, alla terza riga, faccio un `get(String)` sulla `Mappa`, in modo da ottenere l'`ArrayList` che contiene i `Ticket` di quel tipo. Infatti alla *key* `Biglietto Base` corrisponde l'`ArrayList` in cui devo salvare il `Ticket` appena creato; l'operazione di `add()` serve ad aggiungerlo a quest ultimo.

```

1 public class Museo{
2     ...
3     public boolean vendiTicketMostraVirtuale(Acquirente visitatore,
4         Mostra mostra, boolean preferiscoPostoVirtuale){
5         if(mostra.isVirtual()) {
6             try {
7                 visitatore.paga(mostra.getCostoTicket());
8                 if (mostra.getPostiRimasti() > 0) {
9                     if (preferiscoPostoVirtuale) {
10                         if (mostra.getPostiVirtualiRimasti() > 0)
11                             mostra.togliPostoVirtuale();
12                         ...
13                         mostra.incassaTicket(visitatore);
14                         Ticket tmp = new TicketMostraFisicaEVirtuale(visitatore);
15                         ticketMuseoVenduti.get("Ticket_Fisica-Virtuale").add(tmp);
16                         visitatore.addTicket(tmp);
17                         return true;
18                     } else
19                         visitatore.ottieniRimborso(mostra.getCostoTicket());
20                 } catch (NoMoneyException e){return false;}
21             } else{
22                 ...
23             }
24             return false;
25         }
26         ...
27     }

```

Listing 3: Metodo vendiTicketMostraVirtuale

Questo ha lo scopo di prelevare il costo del biglietto dal portafogli dell'Acquirente (usando il metodo `paga(int)`, che può lanciare l'eccezione `NoMoneyException`), e di inserire la somma prelevata nelle casse della Mostra. Le Mostre hanno il field *virtual*, un boolean, che è `true` quando ha almeno una Sala Virtuale.

Se è il metodo `vendiTicketMostraFisica(..)` a essere chiamato, esso fa forwarding su `vendiTicketMostraVirtuale(..)`, mettendo la flag `preferiscoPostoVirtuale` su `false`.

Allo stesso gruppo di classi, ci sono associati i seguenti metodi:

- `registraVisitatore(Acquirente)`
- `registerVisitor(Acquirente)`
- `getCatalogoOpere()`
- `registraSuggerimento(Suggerimento)`

`registraVisitatore(..)` chiama in forward il metodo `registerVisitor(..)`, che fa la registrazione di un *Acquirente*. Gli utenti registrati sono salvati dentro `LinkedHashMap<String, VisitatoreReg>`, dove `String` generata in modo automatico solo ai fini di velocizzare la fase di test. Di seguito l'implementazione di `registerVisitor(..)`.

```
1 public class Museo{
2     ...
3     private void registerVisitor(Acquirente visitatore){
4         int vecchioBilancio = visitatore.getBilancio();
5         try{
6             visitatore.paga(vecchioBilancio);
7         } catch (NoMoneyException e){};
8         Visitatore vreg = new VisitatoreReg(vecchioBilancio);
9         byte[] array = new byte[12];
10        new Random().nextBytes(array);
11        String randomString = new String(array, Charset.forName("ISO-8859-1"));
12        ((VisitatoreReg)vreg).setUsername(randomString);
13        utentiRegistrati.put(randomString, (VisitatoreReg)vreg);
14    }
15 }
16 public void registraSuggerimento(Suggerimento suggerimento){
17     suggerimenti.add(suggerimento);
18     setChanged();
19     notifyObservers(new StatoMuseo(suggerimento, null, null, null, null));
20 }
21 ...
22 }
```

Listing 4: Metodo `registerVisitor` e `registraSuggerimento`

Il metodo `registraSuggerimento(..)` effettua la chiamata di *Observer*: quando viene aggiunto un nuovo suggerimento, l'amministratore (cioè l'Observer di Museo) riceve un oggetto di tipo *StatoMuseo*, che ha tutti i parametri `null` eccetto il nuovo *Suggerimento*. Questo oggetto *StatoMuseo* serve per differenziare i flussi di informazione di cui l'Amministratore ha bisogno per funzionare in modo automatico.

### Componente rivolta a Amministratore e Organizzatore

In questa componente identifico i metodi per interfacciare il museo con l'amministratore o l'organizzatore. I metodi:

- `getSuggerimenti()`
- `getBilancio()`
- `setBilancio(int)`
- `addBilancio(int)`
- `prelevaBilancio()`
- `getOrganizzatori(boolean)`
- `getImpiegati(boolean)`
- `getSale(boolean)`
- `getOpereMuseo()`
- `getGestoreOpere()`
- `addMostra(Mostra)`
- `getMostre()`
- `updateLoadFactorSale()`
- `chiudiMostra(Mostra)`
- `getUtentiRegistrati()`
- `getLoadFactorSale()`

Nei metodi `getOrganizzatori(boolean)`, `getImpiegati(boolean)` e `getSale(boolean)`, il loro parametro serve per ottenere tutti gli *Organizzatori*, o tutti gli *Impiegati*, o tutte le *Sale*, oppure solo quelle libere (liberi). Il metodo `updateLoadFactorSale()` è un metodo privato che serve per ricalcolare il fattore di carico, un campo dell'oggetto Museo che indica quante sale sono occupate rispetto al numero totale. Di seguito un pezzo che spiega questo meccanismo.

```

1  public class Museo{
2      ...
3      private void updateLoadFactorSale(){
4          int saleOccupate = 0;
5          for(Sala sala:sale)
6              if(sala.isBusy())
7                  saleOccupate++;
8          loadFactorSale = (((double) saleOccupate)/(((double) sale.size()));
9      }
10     public void addMostra(Mostra mostra){
11         this.mostre.add(mostra);
12         updateLoadFactorSale();
13         setChanged();
14         notifyObservers(new StatoMuseo(null, null, loadFactorSale, null, true));
15     }
16     public void chiudiMostra(Organizzatore organizzatore, Mostra mostra){
17         if(organizzatore == mostra.getOrganizzatore()){
18             StatoMuseo sm = new StatoMuseo(null, null,
19                 loadFactorSale, mostra, true);
20             this.bilancio += mostra.svuotaCasse();
21             this.mostre.remove(mostra);
22             updateLoadFactorSale();
23             sm.setLoadFactorSale(loadFactorSale);
24             setChanged();
25             notifyObservers(sm);
26         }
27     }
28     ...
29 }

```

Listing 5: Metodi updateLoadFactorSale, addMostra e chiudiMostra

Queste due funzioni sono le uniche che chiamano il metodo updateLoadFactorSale, perché sono gli unici due casi in cui ha senso calcolarlo. Sia `addMostra(Mostra)` che `chiudiMostra(Mostra)` chiamano il metodo `notifyObservers(StatoMuseo)`, cioè chiamano l'Observer di tipo Push. Di seguito spiego quali sono i campi di `StatoMuseo`.

Anche altri tra quei metodi chiamano gli Observers in ascolto, `setBilancio(int)` e `addBilancio(int)`, però questi metodi mandano una notifica di tipo Pull.

La classe **StatoMuseo** ha i seguenti campi:

- suggerimento
- bilancioMuseo
- loadFactorSale
- mostraChiusa

- `museoIsReady`

`mostraChiusa` è l'oggetto di tipo `Mostra` che viene mandato all'Amministratore quando una mostra viene chiusa. `museoIsReady` è una flag che comunica quando il Museo è pronto, ovvero quando l'Amministratore può valutare se creare altre mostre: se è `true`, l'Amministratore può fare un'azione `Amministratore()`.

## 4.2 Gruppo classi Amministratore

Si trovano le classi delle varie *strategie*, la classe *Amministratore* e la classe *IncaricoMostra*. Le strategie servono all'amministratore per creare un `IncaricoMostra` in conseguenza a un cambio di stato del Museo.

### 4.2.1 Classe IncaricoMostra

Ha un costruttore *package-protected*, estende *Observable* e implementa *Incarico*, ovvero un'interfaccia che espone il metodo `svolgiIncarico()`. Quando l'Organizzatore riceve l'incarico, esso diventa un Observer di questo oggetto: questo sistema permette all'Amministratore di chiudere una mostra. A tal proposito, riporto un pezzo di codice:

```

1 public class IncaricoMostra{
2     ...
3     void forzaChiusuraMostra(){
4         setChanged();
5         notifyObservers();
6     }
7     ...
8 }
```

Listing 6: Metodo `forzaChiusuraMostra`

Questo metodo serve a chiamare l'organizzatore con lo scopo di eseguire una chiusura forzata.

Il metodo è *package-protected* garantisce che sia solo l'amministratore a poter chiamare questo metodo. Altri metodi dedicati all'Amministratore sono `setOpere(Set <Opere>)` e `setOrganizzatore(Organizzatore)`.

Quando viene creato un `IncaricoMostra`, gli viene fornito un certo quantitativo di denaro, preventivamente trasferito dalla cassa del museo per merito dell'amministratore; l'organizzatore lo ha poi a disposizione, e per prelevare lo usa il metodo `prelevaDenaro(int)`

I campi di questa classe:

- `ArrayList<Opere> opere`
- `int bilancio`
- `ArrayList<Personale> impiegati`

- boolean killable
- Mostra m
- boolean ancheVirtuale

Quando `killable` è `false`, vuol dire che l'incarico è già stato interrotto, e che quindi la Mostra o non è stata creata, oppure è stata chiusa.

Quando `ancheVirtuale` è `true`, vuol dire che si prevede l'utilizzo anche di sale virtuali.

```

1 public class IncaricoMostra{
2     ...
3     IncaricoMostra(int bilancioMostra, boolean ancheVirtuale){
4         opereMostra = new ArrayList<>();
5         bilancio = bilancioMostra;
6         fondiStanziati = bilancioMostra;
7         this.ancheVirtuale = ancheVirtuale;
8     }
9     ...
10 }
```

Listing 7: costruttore IncaricoMostra

## 4.3 Strategie

Ogni classe strategia implementa l'interfaccia *Strategy*, la quale espone il metodo `strategyMethod()`. Questo viene chiamato direttamente dall'amministratore usando `azioneAmministratore()`.

Le strategie possono essere di due tipi:

- Strategie creazionali
- Strategie non creazionali

### 4.3.1 Strategie creazionali

Ogni strategia creazionale a un certo punto costruisce un oggetto di tipo `IncaricoMostra` e gli fornisce tutte le informazioni che servono, cioè:

- il bilancio
- le Opere richieste

Per quanto riguarda le Opere richieste, è proprio in queste strategie che avviene il loro noleggio, attraverso i metodi di un oggetto di tipo *GestoreOpere*. Anche nel caso in cui l'opera la stia richiedendo il museo proprietario, si deve passare attraverso il *GestoreOpere* per noleggiarla. Per le strategie creazionali lo `strategyMethod()` è composto da 2 parti: inizializzazione e decorazione. Quest



ultima serve per inserire le Opere previste e per noleggiarle. Alla fine, si fa il return dell'IncaricoMostra completo.

Questo incarico va a finire all'Amministratore, il quale si occupa di assegnargli un Organizzatore.

**LowBudgetStrategy** Questa strategy è scelta quando il budget è basso e allora si preferisce creare una mostra usando le opere di proprietà del museo, così da risparmiare sul noleggio.

```
1  class LowBudgetStrategy{
2      ...
3      public IncaricoMostra strategyMethod(Museo museo){
4          int numeroOpereLowStrategy = 3;
5          IncaricoMostra incarico = new IncaricoMostra(50, false);
6          ArrayList<Opera> opere = new ArrayList<>();
7          for(Opera o:museo.getOpereMuseo())
8              if(opere.size() <= numeroOpereLowStrategy)
9                  opere.add(o);
10         incarico.setOpereMostra(opere);
11         return incarico;
12     }
13 }
```

Listing 8: strategyMethod di LowBudgetStrategy

**OnSuggestStrategy** è la strategy che viene scelta quando il numero di suggerimenti è abbastanza alto. Le opere vengono prese da quelle suggerite, e si fa un incarico partendo da quelle.

**PersonalStrategy** Questa classe permette di sviluppare strategie personalizzate grazie al fatto che il costruttore accetta in ingresso più parametri:

```
1  class PersonalStrategy{
2      ...
3      PersonalStrategy(int numeroOpere, boolean ancheVirtuale){
4          this.numeroOpere = numeroOpere;
5          this.ancheVirtuale = ancheVirtuale;
6      }
7      ...
8  }
```

Listing 9: Costruttore di PersonalStrategy

Il metodo `setStrategy()` della classe Amministratore creerà varie PersonalStrategy a seconda dello stato dell'Amministratore.

#### 4.3.2 Strategie non creazionali

- IdleStrategy

- KillStrategy

Entrambe le strategie non creazionali hanno in comune la restituzione di `null`, invece dell'`IncaricoMostra` come succedeva per le strategie creazionali.

**IdleStrategy** non fa nulla. L'amministratore è in stato di quiete e viene impostata in caso non dovesse entrare in nessuna strategia.

```

1 class IdleStrategy{
2     ...
3     public IncaricoMostra strategyMethod(Museo museo) {
4         return null;
5     }
6 }
```

Listing 10: Class IdleStrategy

**KillMostreStrategy** Questa strategia serve per chiudere le mostre. In pratica chiama in forward il metodo `forzaChiusuraMostra()` di *IncaricoMostra*.

```

1 class KillMostreStrategy{
2     private Set<IncaricoMostra> incarichiMostre;
3     private boolean strategyHasKilled = false;
4
5     KillMostreStrategy(Set<IncaricoMostra> incarichiMostre,
6         Amministratore amministratore){
7         this.incarichiMostre = incarichiMostre;
8         this.amministratore = amministratore;
9     }
10    public IncaricoMostra strategyMethod(Museo museo) {
11        Iterator<IncaricoMostra> iterator = incarichiMostre.iterator();
12        while(!strategyHasKilled && iterator.hasNext()){
13            IncaricoMostra temp = iterator.next();
14            if(temp.isKillable()) {
15                this.strategyHasKilled = true;
16                temp.kill();
17                temp.forzaChiusuraMostra();
18                break;
19            }
20        }
21        return null;
22    }
23    public void setReady(){
24        this.strategyHasKilled = false;
25    }
26
27    public boolean isReady(){
28        return !strategyHasKilled;
29    }
29 }
```

```

29     }
30 }

```

Listing 11: Classe KillMostreStrategy

La flag `strategyHasKilled` mi dice se la strategia è pronta, cioè se è possibile usarla per chiudere una Mostra. Una volta che viene attivata dallo `strategyMethod()`, bisogna usare il metodo `setReady()` per sbloccarla.

Quello che fa questa strategia è: legge gli incarichi che sono stati creati, trova il primo che risulta *unkilled* e chiama `forzaChiusuraMostra()` su quel incarico.

## 4.4 Classe Amministratore

La classe Amministratore ha il ruolo di gestire il Museo. Per fare questo, implementa l'interfaccia *Observer*: grazie al metodo `update(Observable o, Object arg)` esposto dall'interfaccia, aggiorna il suo stato e successivamente esegue un'azione in modo automatico grazie all'implementazione del pattern *Strategy*. Sono presenti un certo numero di flag, e un certo numero di attributi, che vengono aggiornati quando Museo manda una notifica:

- `goAuto`: permette di prendere decisioni automatiche
- `semaphoreCreational`: se true, permette strategie di creazione Mostre automatiche; altrimenti userà o la *IdleStrategy*, o la *KillStrategy*.
- `lock`: l'amministratore deve poter creare solo una mostra per volta: quando va in lock, allora aspetta la flag *museoIsReady* per poter essere sbloccato.

Come fields:

- `LinkedHashMap<Opera,Integer> suggPerOpera`: contiene il numero di suggerimenti per ogni opera.
- `LinkedHashSet<IncaricoMostra> incarichiCreati`: contiene gli incarichi fino ad ora creati dall'Amministratore.
- `LinkedHashSet<Opera> richieste`: quando le opere raggiungono un certo numero di suggerimenti, vengono messe qui dentro.
- `ArrayList<Mostra> mostreConcluse`: le mostre che vengono rimosse dal Museo vengono salvate qui dentro.
- `Strategy actualStrategy`
- `Strategy killStrategy`: visto che la *KillStrategy* deve mantenere uno stato, creo una *KillStrategy* dal costruttore di Amministratore e ne mantengo il riferimento.

Il metodo `update(..)`, usa i seguenti metodi di supporto:

- updateSemaforo()
- checkNumSuggerimenti()
- setStrategy()
- checkForLockRelease()
- azioneAmministratore()
- chiusuraEmergenza()

#### 4.4.1 Metodi a supporto di update

**updateSemaforo()** serve per aggiornare la flag `semaphoreCreational`.

```

1 public class Amministratore{
2     ...
3     private void updateSemaforo(){
4         int incarichiAttivi = museo.getMostre().size();
5
6         if(incarichiAttivi < 2 && loadFactorSale < 0.65) // semaforo verde
7             semaphoreForCreationalStrategies = true;
8         else
9             semaphoreForCreationalStrategies = false;
10    }
11    ...
12 }
```

Listing 12: Metodo updateSemaforo

Controlla se il `loadFactor` è al di sotto del 65% e se il numero di *incarichiAttivi* è al di sotto di 2. In questo caso l'Amministratore può usare una strategia di creazione.

**checkNumSuggerimenti()** serve per contare quanti suggerimenti ci sono per ogni opera e, se sono uguali o maggiori a 5, le *Opere* suggerite vengono messe in *richieste*.

#### **setStrategy()**

```

1 public class Amministratore{
2     ...
3     private void setStrategy(){
4         if(!semaphoreForCreationalStrategies){ // semaforo ROSSO
5             if (loadFactorSale > 0.9)
6                 strategy = killMostreStrategy;
7             else // semaforo GIALLO
8                 strategy = idleStrategy;
9         } else { // semaforo VERDE
```

```

10         if (accumuloRichieste.size() >= 2 && bilancioMuseo >= 400) {
11             strategy = new OnSuggestStrategy(accumuloRichieste, this);
12         } else {
13             if (bilancioMuseo < 1000)
14                 strategy = idleStrategy;
15             else if (1000 <= bilancioMuseo && bilancioMuseo < 2500)
16                 strategy = new LowBudgetStrategy();
17             else if (2500 <= bilancioMuseo && bilancioMuseo < 5000)
18                 strategy = new PersonalStrategy(5, true);
19             else if (5000 <= bilancioMuseo && bilancioMuseo < 10000)
20                 strategy = new PersonalStrategy(7, false);
21             else
22                 strategy = idleStrategy;
23         }
24     }
25 }
26 ...
27 }

```

Listing 13: Metodo setStrategy

Qui si vede come è applicata la flag `semaphoreCreational`. Infatti, se è false, allora non creerà mai un nuovo *IncaricoMostra*, ma mi limiterò ad aspettare oppure a chiudere alcune mostre attive.

```

        checkForLockRelease()
1 public class Amministratore{
2     ...
3     private void checkForLockRelease(){
4         int expectedMostreVive = 0;
5         int expectedMostreMorte = 0;
6         for(IncaricoMostra incaricoMostra : incarichiCreati){
7             if(incaricoMostra.isKillable())
8                 expectedMostreVive++;
9             else
10                 expectedMostreMorte++;
11         }
12         if(museo.getMostre().size() == expectedMostreVive
13             && mostreChiuse.size() == expectedMostreMorte) {
14             lockCreationIncaricoMostre = false;
15             ((KillMostreStrategy)killMostreStrategy).setReady();
16         }
17     }
18     ...
19 }

```

Listing 14: Metodo checkForLockRelease

Capisce se si può rilasciare la flag `lock`. Mi assicuro che Museo e Amministratore siano sincronizzati, verificando che ci siano lo stesso numero di Mostre attive nel Museo quanti sono gli incarichi *unkilled*, e che gli incarichi *killed* siano in numero uguale alle Mostre chiuse.

#### azioneAmministratore()

```

1 public class Amministratore{
2     ...
3     public IncaricoMostra azioneAmministratore(){
4         IncaricoMostra incaricoMostra = strategy.strategyMethod(museo);
5         if (incaricoMostra != null && !lockCreationIncaricoMostre) {
6             lockCreationIncaricoMostre = true;
7             Organizzatore organizzatore =
8                 (Organizzatore) museo.getOrganizzatori(true).get(0);
9             incaricoMostra.setOrganizzatore(organizzatore);
10            incarichiCreati.add(incaricoMostra);
11            organizzatore.setIncaricoAttuale(incaricoMostra);
12            organizzatore.svolgiIncaricoAssegnato();
13        }
14        else if (incaricoMostra != null){
15            chiusuraEmergenza(incaricoMostra);
16        }
17        return incaricoMostra;
18    }
19    ...
20 }
```

Listing 15: Metodo azioneAmministratore

E' in questo metodo che viene chiamato lo `strategyMethod()`, ed eventualmente, se la strategia è di tipo creazionale, allora avrò creato un `IncaricoMostra`. Nel caso in cui l'Incarico sia stato creato ma il lock era azionato, allora bisogna cancellare l'incarico, e lo si fa attraverso la `chiusuraEmergenza()`.

#### chiusuraEmergenza(IncaricoMostra)

```

1 public class Amministratore{
2     ...
3     private void chiusuraEmergenza(IncaricoMostra im){
4         GestoreOpere go = museo.getGestoreOpere();
5         for (Opera o:im.getOpereMostra()){
6             go.restituisceOperaAffittata(o);
7         }
8         try {
9             museo.addBilancio(this, im.getBilancio());
10        } catch (Exception e){}
11    }
```

```

12    ...
13 }

```

Listing 16: Metodo chiusuraEmergenza

#### 4.4.2 Il metodo update

Questo metodo lo possiamo dividere in due parti: la prima, di *aggiornamento*, e la seconda, di *esecuzione*. Nella prima ci sono le dichiarazioni delle seguenti flag:

- `needAdminOp = false`
- `strongRestart = false`

Sono entrambe inizializzate su false, e se alla fine del processo di aggiornamento saranno diventate true, sarà necessaria un'azione dell'amministratore. La fase di aggiornamento può essere fatta in 3 modi:

- usando uno *StatoMuseo*, che come si è visto è l'oggetto mandato da Museo e viene ricevuto in modo push. A seconda di quale sia il dato ricevuto, `needAdminOp` può diventare true o restare false.
- usando `notifyObserver()`, senza parametri e da Museo, abbiamo stavolta un pull, e l'amministratore a questo punto aggiornerà solo il fattore di carico e il bilancio usando i metodi getter forniti dal museo. In questo caso `needAdminOp` verrà impostata su true.
- chiamando `update(null,null)`. In questo caso si avvia una procedura forzata. `strongRestart` viene impostato su true.

```

1  public class Amministratore{
2      ...
3      public void update(Observable o, Object arg) {
4          boolean needAdminOp = false;
5          boolean strongRestart = false;
6
7          // UPDATE STATO MUSEO
8          if (o == museo && arg != null) {
9              StatoMuseo sm = (StatoMuseo) arg;
10             Suggerimento suggerimento = sm.getOperaSuggerita();
11             Integer bilancioMuseoState = sm.getBilancioMuseo();
12             Double loadFactor = sm.getLoadFactorSale();
13             Mostra mostra = sm.getMostraChiusa();
14             Boolean museoIsReady = sm.getMuseoIsReady();
15
16             if (suggerimento != null) {
17                 Opera opera = suggerimento.getSuggerimento();

```

```

18         int prevInt = suggerimentiPerOpera.get(opera);
19         suggerimentiPerOpera.put(opera, ++prevInt);
20         countSuggest++;
21     }
22     if (bilancioMuseoState != null) {
23         bilancioMuseo = bilancioMuseoState;
24         needAdminOp = false;
25     }
26     if (loadFactor != null) {
27         loadFactorSale = loadFactor;
28         needAdminOp = false;
29     }
30     if (mostra != null) {
31         mostreChiuse.add(mostra);
32         checkForLockRelease();
33         needAdminOp = true;
34     }
35     if (museoIsReady != null) {
36         checkForLockRelease();
37         needAdminOp = true;
38     }
39     bilancioMuseo = museo.getBilancio(this);
40 }
41 else if (o == null && arg == null) {
42     strongRestart = true;
43 }else{
44     bilancioMuseo = museo.getBilancio(this);
45     loadFactorSale = museo.getLoadFactorSale(this);
46     needAdminOp = true;
47 }
48 ...
49 }

```

Listing 17: Update - fase di aggiornamento

Alla fine di questa fase, avremo aggiornato lo stato dell'amministratore e avremo le seguenti flag impostate per prendere decisioni nella seconda parte del metodo:

- needAdminOp
- strongRestart
- lock
- strategyHasKilled
- goAuto



Lock è una flag dell'oggetto amministratore, come anche goAuto. Lock serve per bloccare la creazione di Mostre. Invece strategyHasKilled è la flag che viene presa dalla strategia *KillStrategy*.

```

1 public class Amministratore{
2     ...
3     public void update(Observable o, Object arg){
4         ...
5         if(strongRestart){
6             updateSemaforo();
7             setStrategy();
8             azioneAmministratore();
9         }
10        else {
11            // AMMINISTRATORE AUTOMATICO
12            updateSemaforo();
13            if (goAutomatico && needAdminOp) {
14                if (!semaphoreCreational) {
15                    setStrategy();
16                    boolean killerIsReady =
17                        ((KillMostreStrategy) killMostreStrategy).isReady();
18                    if (strategy instanceof KillMostreStrategy && !killerIsReady) {
19                        ;
20                    } else
21                        azioneAmministratore();
22                }
23            } else {
24                if (!lockCreationIncaricoMostre) {
25                    setStrategy();
26                    azioneAmministratore();
27                }
28            }
29        }
30    }
31    ...
32 }

```

Listing 18: Update - fase esecuzione

Se **strongRestart** è true, salto tutte le verifiche, e aggiorno direttamente la flag **semaphore**, seleziono la strategy usando **setStrategy()** ed eseguo **azioneAmministratore()**.

Se invece non è così, allora per agire faccio l'**updateSemaforo()**, e poi se ho il **goAutomatico** e **needAdminOp** a true, posso forse fare l'azione.

Qui si biforcano due casi, a seconda della flag **semaphoreCreational**: se false, allora devo adottare una strategia non creazionale, e se il **setStrategy** mi ha impostato la **killStrategy**, devo vedere se lei è pronta; se lo è, allora chiamo **azioneAmministratore()**.

Se invece la strategia è una *IdleStrategy*, allora eseguo senza alcun blocco `azioneAmministratore()`.

Se invece il `semaphoreCreational` è `true`, vuol dire che voglio creare una mostra. Per crearla devo giusto controllare la flag `lock` e a quel punto posso eseguire `azioneAmministratore()`.

Al fine di aumentare la chiarezza, ho riportato un sequence diagram di come si crea una mostra in modo automatico:

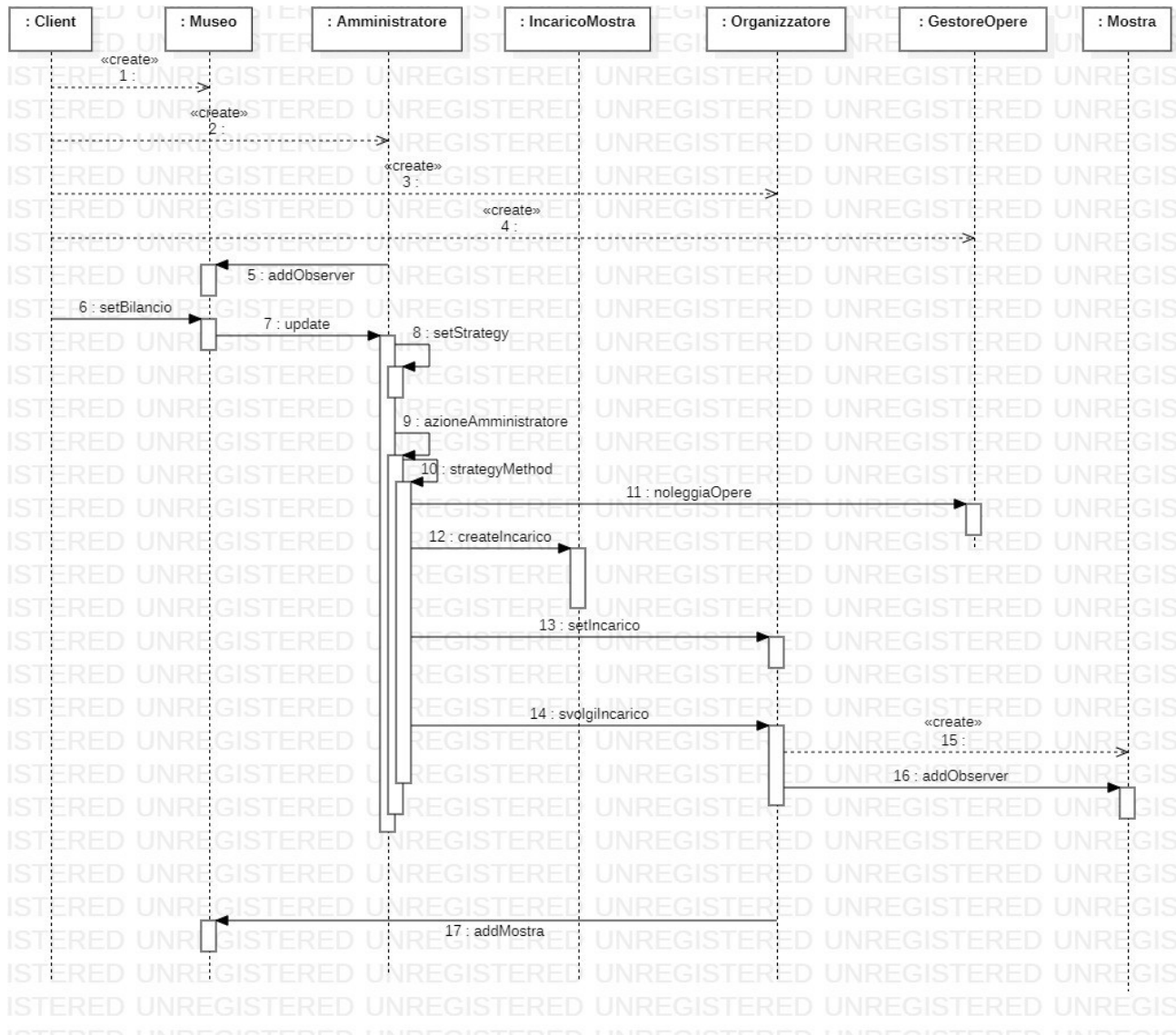


Figure 16: Sequence diagram creazione mostra

## 5 Package organizzazione

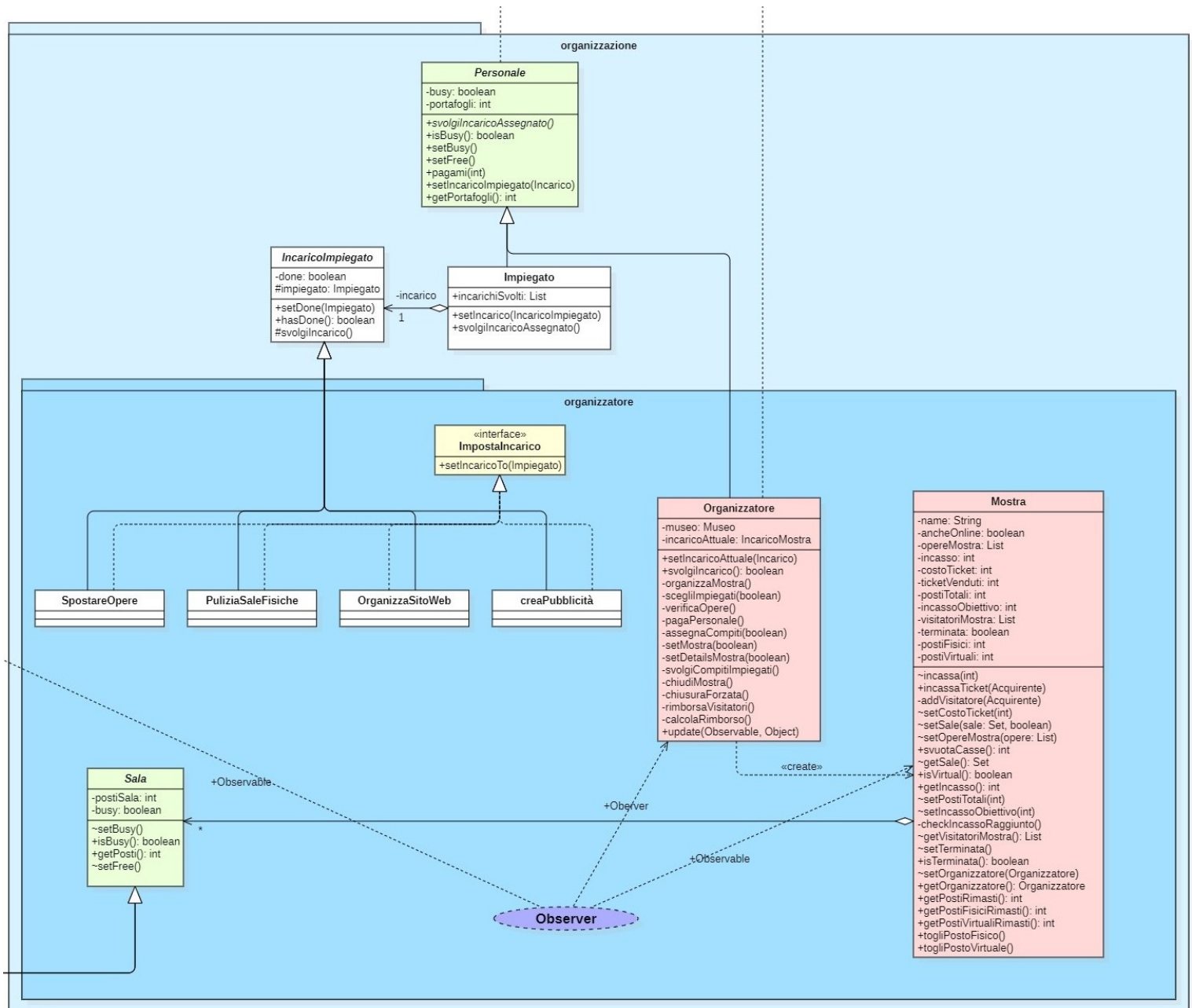


Figure 17: Package organizzazione

Sono presenti tutte quelle classe *Impiegato* e la classe *Organizzatore* estendono la classe *Personale*. La classe più strutturata di questo package è la classe *Organizzatore*

## 5.1 Classe Organizzatore

Come già accennato questa classe implementa *Observer*, e ha un metodo `update(Observable o, Object arg)`, che viene chiamato dai due oggetti che osserva: *IncaricoMostra* e *Mostra*.

```
1 public class Organizzatore{
2     ...
3     public void update(Observable o, Object arg) {
4         Mostra m = incaricoAttuale.getMostra();
5         if(o != m)
6             chiusuraForzata();
7         else
8             chiudiMostra();
9     }
10    ...
11 }
```

Listing 19: Metodo update di Organizzatore

Se l'Observable non è la *Mostra*, allora è automaticamente l'*IncaricoMostra* che ha mandato una notifica, e la manda solo in caso l'amministratore voglia una chiusura forzata.

```
1 public class Organizzatore{
2     ...
3     private void chiusuraForzata(){
4         stornaDenaroVisitatori();
5         chiudiMostra();
6     }
7     private void chiudiMostra(){
8         Mostra mostra = incaricoAttuale.getMostra();
9         GestoreOpere gestoreOpere = museo.getGestoreOpere();
10        for(Opera o: incaricoAttuale.getOpereMostra())
11            gestoreOpere.restituisceOperaAffittata(o);
12        for(Personale pp:incaricoAttuale.getImpiegati()) {
13            pp.setFree();
14            try {
15                pp.setIncaricoImpiegato(null);
16            } catch (Exception e) { }
17        }
18
19        Set<Sala> saleMostra = mostra.getSale();
20        for(Sala sala: saleMostra)
```

```

21     sala.freeSala();
22     this.setFree();
23     mostra.setTerminata();
24     mostra.deleteObserver(this);
25     museo.chiudiMostra(this, mostra);
26     incaricoAttuale.kill();
27 }
28 ...
29 }

```

Listing 20: Metodo chiudiMostra e chiusuraForzata

chiudiMostra() si occupa di liberare gli spazi occupati, ritornare le opere e dire al museo di chiudere la mostra.

L'organizzatore svolge il suo lavoro quando viene chiamato il suo metodo svolgiIncarico() dall'esterno; questo metodo è chiamato in forward dal metodo azioneAmministratore() di amministratore.

svolgiIncarico() si appoggia sul metodo privato organizzaMostra:

```

1 public class Organizzatore{
2     ...
3     private void organizzaMostra(){
4         boolean ancheVirtuale = incaricoAttuale.isAncheVirtuale();
5         scegliImpiegati(ancheVirtuale);
6         assegnaCompiti(ancheVirtuale);
7         pagaPersonale();
8         try {
9             affittaOpere();
10        } catch (Exception e){System.err.println(e.getMessage());}
11        setMostra(ancheVirtuale);
12        svolgiCompitiImpiegati();
13        museo.addMostra(incaricoAttuale.getMostra());
14    }
15    ...
16 }

```

## 5.2 Altre classi

Le altre classi sono sostanzialmente molto poco complesse. Le elenco:

- *IncaricoImpiegato(abstract)*
- *Sala(abstract)*
- *Personale(abstract)*
- *Impiegato*
- *CreaPubblicita'*

- *OrganizzaSitoWeb*
- *PulizieSaleFisiche*
- *SpostareOpera*

IncaricoImpiegato è una classe astratta che fa da superclasse per CreaPubblicità, OrganizzaSitoWeb, PulizieSaleFisiche e SpostareOpera e che implementa l'interfaccia *Incarico*. La classe astratta *Personale* ha un `int` che indica il bilancio, e un metodo `pagami(int)`, che serve per ricevere denaro.

La class *Sala* espone i metodi *package-protected* `setBusy()` e `setFree()`, che devono essere usati dall'organizzatore.

## 6 Altri packages

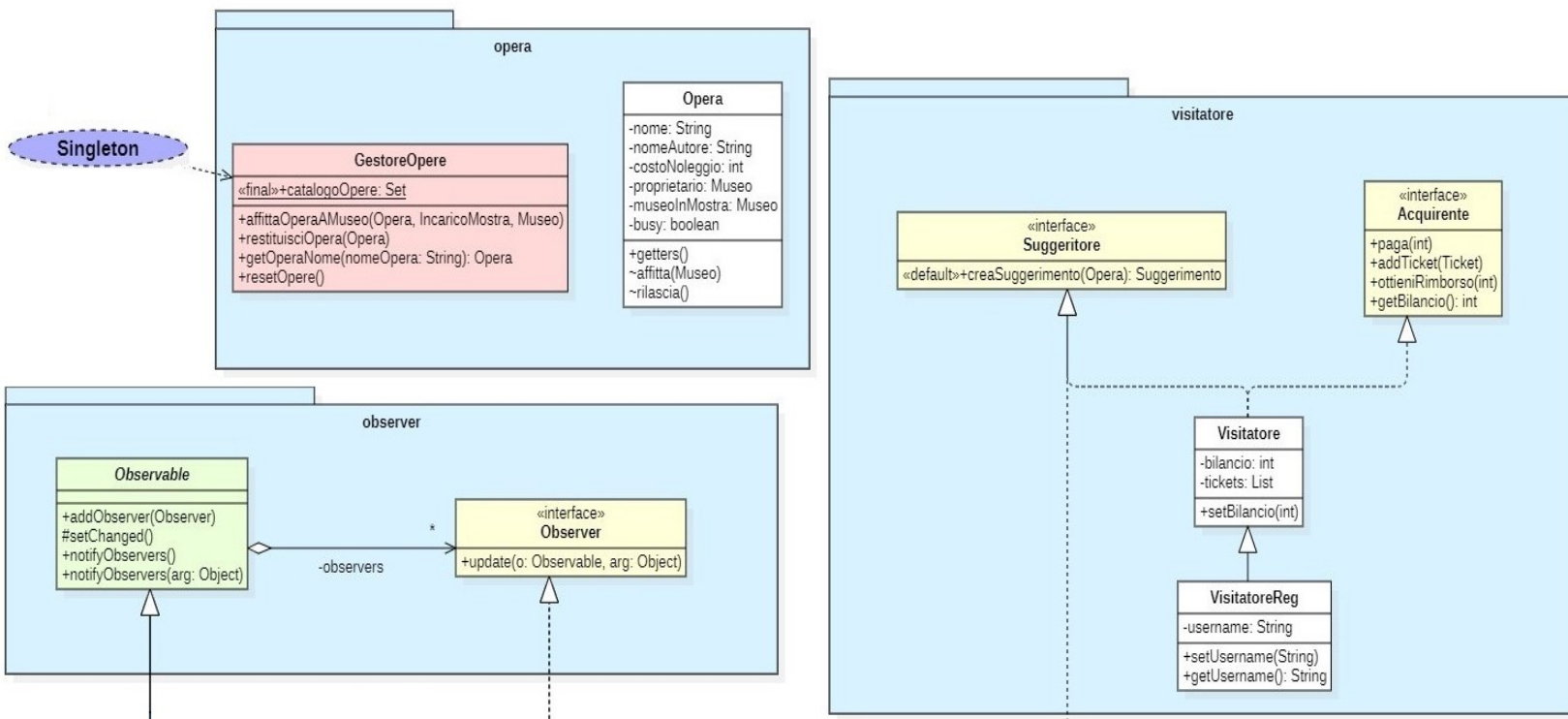


Figure 18: observer, visitatore, opera

### 6.1 Package opera

Vi sono due classi in questo package:

- *Opera*
- *GestoreOpere*

#### 6.1.1 Classe Opera

Opera è un oggetto semplice, parzialmente immutabile se non nel field dell'utilizzatore (cioè il Museo che lo sta noleggiando) e nella flag **busy** se occupata in qualche mostra. E' una classe final, quindi non può essere estesa. Gli unici metodi che gli cambiano lo stato sono **affitta(Museo)** e **rilasciaOpera()**, che hanno visibilità *package-protected*. Questi metodi vengono utilizzati solamente dal *GestoreOpere*.

```

1 public class Opera{
2     ...
3     Opera(String nome, String autore, Museo proprietario, int rentCost){
4         ...
5     }
6     ...
7 }

```

Listing 21: Costruttore Opera

### 6.1.2 Classe GestoreOpere

In questa classe vi sono i metodi da usare per affittare un'opera, e quindi renderla occupata, e ripristinare lo stato originale dell'opera. GestoreOpere ha un field statico, che è final, ed è del tipo:

```

1 public final static Set<Opera> catalogoOpere = Set.of(
2     new Opera("La_Gioconda", "Leonardo_Da_Vinci", null, 50),
3         ...
4         ...
5 );

```

Listing 22: Il field catalogoOpere

Il metodo della classe Set, *Set.of(...)*, costruisce un insieme immutabile, e il field **public static** mi garantisce visibilità in tutto il sistema e la sicurezza che non possa venire copiato o modificato. Riporto il metodo `affittaOpere(Opera, IncaricoMostra, Museo)`

```

1 public class GestoreOpere{
2     ...
3     public void affittaOpera(Opera opera, IncaricoMostra im, Museo richiedente){
4         try {
5             if (!opera.isBusy()) {
6                 if (opera.getProprietario() != richiedente) {
7                     int costoOperaDaAffittare = opera.getCostoNoleggio();
8                     try {
9                         im.prelevaDenaro(this, costoOperaDaAffittare);
10                        opera.affitta(museoRichiedente);
11                        opera.getProprietario().addBilancio(this,
12                            opera.getCostoNoleggio());
13                    } catch (NullPointerException e) {
14
15                        System.err.println("Trovato_proprietario_null");
16                    } catch (Exception e) {
17                        System.err.println(e.getMessage());
18                    }
19                }
20            } else {

```



```

21         opera.affitta(museoRichiedente);
22     }
23     }else
24         throw new Exception("Opera□gia□occupata");
25     }catch (Exception e) {System.err.println(e.getMessage());
26     }
27 }
28 ...
29 }

```

Listing 23: Metodo affittaOpere

Questo metodo si occupa anche di fare la transazione di denaro, e annulla lanciando l'eccezione nel caso non riesca a prelevare il denaro necessario. Il denaro lo prende direttamente dall'IncaricoMostra passato come parametro.

## 6.2 Package Visitatori

Entrambe le classi presenti implementano l'interfaccia *Acquirente*, e queste sono:

- *Visitatore*
- *VisitatoreRegistrato*

*VisitatoreRegistrato* eredita direttamente *Visitatore*, offrendo in più i metodi `setUsername(String)` e `getUsername()`. *Visitatore*, per come prevede l'interfaccia *Acquirente*, offre il metodo `paga(int)`, che serve per pagare e che può lanciare un'eccezione del tipo *NoMoneyException* se il visitatore non ha una cifra maggiore o uguale a quella passata come parametro.

## Part III

# Test

Le varie classi di Test sono tutte contenute nello stesso package. Ogni classe ha sempre i seguenti field:

- *Museo* museo
- *Amministratore* amministratore

Inoltre, dopo ogni test vengono ripristinate le opere ai proprietari, usando la funzione `resetAllOpere()` della classe `GestoreOpere`. È ricorrente l'uso di `forceStrategyExecution(int, int, boolean)` che imposta in una variabile locale di tipo *Strategy*, la strategia scelta tramite il primo `int`. Il secondo è il numero di opere che si vogliono inserire e il terzo è `true` se si vogliono usare anche le sale virtuali. Dopo, senza alcun check sulle flags, chiama una sua versione del metodo `l'azioneAmministratore()`, vista in classe `Amministratore`.

```
1 public class Amministratore{
2     ...
3     public IncaricoMostra forceStrategyExecution(int nStrategy,
4     int nOpere, boolean ancheVirtuale){
5         Strategy tempStrategy = idleStrategy;
6         if(nStrategy == 0) {    // Num 0 = strategia Killer
7             tempStrategy = new KillMostreStrategy(incarichiCreati);
8         }else if(nStrategy == 1){ // Num 1 = strategia Lowbudget
9             tempStrategy = new LowBudgetStrategy();
10        }else if(nStrategy == 2){ // Num 2 = strategia Personale
11            tempStrategy = new PersonalStrategy(nOpere, ancheVirtuale);
12        }else if(nStrategy == 3){ // Num 3 = strategia suggerimento
13            tempStrategy = new OnSuggestStrategy(accumuloRichieste, this);
14        }
15        IncaricoMostra incaricoMostra = tempStrategy.strategyMethod(museo);
16        if(incaricoMostra != null) {
17            Organizzatore organizzatore =
18                (Organizzatore) museo.getOrganizzatori(true).get(0);
19            incaricoMostra.setOrganizzatore(organizzatore);
20            incarichiCreati.add(incaricoMostra);
21            organizzatore.setIncaricoAttuale(incaricoMostra);
22            organizzatore.svolgiIncaricoAssegnato();
23        }
24        return incaricoMostra;
25    }
26    ...
27 }
```

Un altro metodo chiamato spesso è `setAmministratoreAutomatico(boolean)` che imposta la flag `goAutomatic` della classe `Amministratore`.

## 7 Test noleggio Opere

In questa classe, si verifica il funzionamento del meccanismo di noleggio, e il lancio delle eccezioni in caso qualcosa vada storto. Per inciso: è stato impostato che il costo del noleggio di ogni opera è pari a 50, ma non è un fatto assolutamente vincolante che tutte le opere abbiano lo stesso prezzo, o che il prezzo sia 50

La classe ha riferimenti a un *museo*, al suo *amministratore* e a due *opere*. Il primo test verifica che le opere affittate finiscano al proprietario.

```
1 public class TestNoleggio{
2     ...
3     public void testNoleggiaConSoldi(){
4         amministratore.setAmministratoreAutomatico(false);
5         museo.setBilancio(amministratore, 50000);
6         incaricoMostra = amministratore.forceStrategyExecution(2,14,false);
7         for(Opera o: incaricoMostra.getOpereMostra())
8             assertTrue(o.getAffittuario() == museo);
9     }
10    ...
11 }
```

Listing 24: Test noleggio con Soldi

Il secondo verifica che almeno una delle opere che si è richiesto, non sia stata affittata. Infatti, impostando il valore del bilancio del museo a 50, è possibile affittare al massimo un'opera.

```
1 public class TestNoleggio{
2     ...
3     public void testNoleggiaSenzaSoldi(){
4         amministratore.setAmministratoreAutomatico(false);
5         museo.setBilancio(amministratore, 50);
6         int numOpere = 5;
7         incaricoMostra =
8             amministratore.forceStrategyExecution(2,numOpere,false);
9         int count = 0;
10        for(Opera o:incaricoMostra.getOpereMostra())
11            if(o.getAffittuario() == museo)
12                count++;
13        assertFalse(count == numeroOperePreviste);
14    }
15    ...
16 }
```

Listing 25: Test noleggio senza Soldi

## 8 Test strategies

Fa il testing delle 5 strategie che ho creato. Si usano i metodi di stop di amministratore e di forzatura della strategia; quest ultimo è comodo perché mi ritorna l'*IncaricoMostra* che ha costruito usando una specifica strategia, e posso valutare quindi che sia conforme con quello che mi aspetto. Questa classe ha il field *incaricoMostra*, che serve nel test della *KillStrategy*.

**Test IdleStrategy** Verifico se dal *forceAdminStrategy* ottengo un null, e se è null, siamo in *IdleStrategy*.

**Test LowBalanceStrategy** Verifico che ogni opera dell'*IncaricoMostra* inserita, sia di appartenenza del museo associato all'amministratore.

**Test PersonalStrategy** Questa strategia è impostata affinché scelga in modo automatico un'opera del museo, e che la restante parte delle opere previste non vi appartengano.

Per questa strategia, che prevede quindi il noleggio di Opere, verifico che tra le Opere aggiunte, ce ne sia qualcuna che non è del museo.

**Test OnSuggestStrategy** Questo metodo aggiunge al museo un certo numero di suggerimenti per opere. Alla fine, l'*IncaricoMostra* creato deve contenere alcune delle opere suggerite.

```
1 public class TestStrategies{
2     ....
3     public void testStrategySuggerimenti(){
4         amministratore.setAmministratoreAutomatico(false);
5         museo.setBilancio(amministratore, 500);
6         Visitatore v = new Visitatore(100);
7         Opera opera1 = gestoreOpere.getOperaNome("TestY");
8         Opera opera2 = gestoreOpere.getOperaNome("TestZ");
9         for(int i=0;i<5;i++) {
10             museo.registraSuggerimento(new Suggerimento(opera1, v));
11             museo.registraSuggerimento(new Suggerimento(opera2, v));
12         }
13         incaricoMostra = amministratore.forceStrategyExecution(3);
14         assertTrue(incaricoMostra.getOpereMostra().contains(opera1));
15         assertTrue(incaricoMostra.getOpereMostra().contains(opera2));
16     }
17     ...
18 }
```

Listing 26: Test per strategia suggerimenti

### Test KillStrategy

```
1 public class TestStrategies{
```

```

2    ...
3    public void testKillStrategy(){
4        testStrategySuggerimenti();
5        Mostra m = incaricoMostra.getMostra();
6        assertFalse(m.isTerminata());
7        amministratore.forceStrategyExecution(0);
8        assertTrue(m.isTerminata());
9    }
10   ...
11 }

```

Listing 27: Test sulla KillStrategy

In questo caso ricicliamo la mostra creata su suggerimento dal test precedente, e poi forziamo la chiusura usando il metodo `forceStrategyExecution(0)`, che forza l'avvio della KillStrategy.

## 9 Test suggerimento

In questa classe ci sono due test:

**Test aggiunta suggerimento** Qui verifico che un suggerimento inserito risulti presente nella lista dei suggerimenti del museo.

**Test update amministratore** Quando aggiungo un suggerimento al museo, voglio che anche l'amministratore ne venga a conoscenza, e che lo salvi nella sua struttura dati corrispondente.

```

1    public class TestSuggerimenti{
2        ...
3        public void testUpdateAmministratore() {
4            LinkedHashMap<Opera, Integer> hashMapAdmin =
5                amministratore.getSuggerimentiPerOpera();
6            assertTrue(hashMapAdmin.get(suggerimento.getSuggerimento()) == 0);
7            museo.registraSuggerimento(suggerimento);
8            assertTrue(hashMapAdmin.get(suggerimento.getSuggerimento()) == 1);
9        }
10       ...
11    }

```

Listing 28: Test aggiunta suggerimeto

## 10 Test di kill e di rimborso

Sto testando la chiusura forzata di una mostra, e la chiusura automatizzata. Nel primo caso, forzo il museo ad avere un alto fattore di carico dopo aver disattivato l'amministratore automatico. Quindi chiamo un update forzato,

usando il comando `amministratore.update(null, null)` . Verifico che a quel punto il fattore di carico delle sale sia sceso.

```
1 public class TestKillRimborso{
2     ...
3     public void testKillMostreAperteHihLoadFactor(){
4         amministratore.setAmministratoreAutomatico(false);
5         museo.setBilancio(amministratore, 6000);
6         incaricoMostra = amministratore.forceStrategyExecution(2, 20, true);
7         incaricoMostra = amministratore.forceStrategyExecution(2, 20, true);
8         incaricoMostra = amministratore.forceStrategyExecution(2, 20, true);
9         incaricoMostra = amministratore.forceStrategyExecution(2, 20, true);
10        assertTrue(amministratore.getLoadFactorSale() > 0.9);
11        mostra = incaricoMostra.getMostra();
12        amministratore.update(null, null);
13        assertTrue(amministratore.getLoadFactorSale() <= 0.9);
14    }
15    ...
16 }
```

Listing 29: Test chiusura forzata mostre in caso di alto fattore di carico

**Test con rimborso ai visitatori** Il rimborso avviene quando è l'amministratore a chiedere l'annullamento della mostra e quando ci sono dei visitatori che hanno pagato il ticket.

In questo test ho creato una mostra, dei visitatori, e ho fatto entrare i visitatori in quella mostra. Dopodiché forzo l'amministratore a chiudere la mostra. Ho scelto che, in caso di rimborso, l'Organizzatore della mostra ritorni ai compratori il prezzo del biglietto meno il 40%, e quindi verifico che i visitatori abbiano esattamente quella somma.

```
1 public class TestKillRimborso{
2     ...
3     public void testRimborso(){
4         ...
5         assertEquals(0, visitatori[0].getBilancio());
6         amministratore.update(null, null);
7         assertEquals(costoMenoTrattenuta, visitatori[0].getBilancio());
8     }
9     ...
10 }
```

Listing 30: Parte finale test rimborso denaro

*visitatori* è un array di **Visitatore**: ognuno viene creato con la stessa quantità di denaro, pari al costo del biglietto di una certa mostra.

Questo metodo verifica che prima della chiusura della mostra, il bilancio del visitatore è a 0, e che dopo la chiusura forzata, il bilancio è esattamente il costo del biglietto meno la trattenuta.

## 11 Test creazione mostre

Sono due test che verificano che, dopo lo svolgimento dell'IncaricoMostra, il museo dispone di esattamente il numero di mostre previste. Non sono test particolarmente strutturati quelli di questa classe, in quanto si limitano a verificare che la dimensione del *Set* che contiene le mostre attive dentro il museo sia uguale al numero di IncarichiMostra svolti.

## 12 Test vendita ticket

In questa classe abbiamo dei metodi che vogliono testare il corretto funzionamento del sistema di acquisti dei ticket. Le funzioni di vendita ticket ritornano **true** se il biglietto viene venduto

In **testVenditaNoBudget** verifico che, in caso non ci siano soldi, si lanci l'eccezione *NoMoneyException*

```
1 public class TestVendite{
2     ...
3     public void testVendiBigliettoNoBudget(){
4         try {
5             museo.vendiBigliettoMuseo(new Visitatore());
6         } catch (NoMoneyException e) {
7             assertTrue(true);
8         }
9     }
10    ...
11 }
```

Listing 31: Test vendita biglietto museo

**venditaBigliettoIngresso** Verifica che sia **true** il return della funzione di acquisto di un biglietto.

**venditaTicketFisici e testSaleMiste** Questo metodo testa le funzioni di vendita di ticket della classe museo. Per farlo, si crea un array di acquirenti dotati di denaro, e poi prova a fargli acquistare i ticket per una mostra. Come verifica del test, mi accerto che la mostra sia chiusa. Infatti, se chiusa, lo è perché ha raggiunto il massimo di posti, e quindi l'Organizzatore ha potuto chiuderla. Anche l'altra funzione di test, ovvero **testSaleMiste()** segue lo stesso principio di verifica. Mostro quest ultimo:

```
1 public class TestVendite{
2     ...
3     public void testSaleMiste(){
4         museo.setBilancio(amministratore, 6000);
5         Visitatore[] visitatori = new Visitatore[60];
6         for(int i=0;i< visitatori.length;i++){
```



```

7      visitatori[i] = new Visitatore(200);
8      museo.vendiBigliettoMuseo(visitatori[i]);
9  }
10     assertTrue(museo.getMostre().size() != 0);
11     Object mostre[] = museo.getMostre().toArray();
12     Mostra m = ((Mostra)mostre[0]);
13     for(Map.Entry<String, VisitatoreReg> entry:
14         museo.getUtentiRegistrati().entrySet()) {
15         museo.vendiTicketMostraVirtuale(entry.getValue(), m, true);
16     }
17     assertTrue(m.isTerminata());
18 }
19 ...
20 }

```

Listing 32: Test vendita ticket

## 13 Test Observer

Infine, questa classe prova tutta la procedura che riguarda entrambi gli Observer (Amministratore e Organizzatore).

**Test final** Questo test si può semplificare come test *Create–Sell–Destroy–Repeat*, in cui a essere create e distrutte sono le mostre, e a essere venduti sono i biglietti per le mostre. In pratica, faccio 2 volte il test sul meccanismo dell'observer, chiamando prima le strategie creazionali, poi quella distruttiva.

Di tutte le mostre che vengono create, vengono venduti tutti i biglietti grazie a un array di visitatori inizializzato all'inizio del test.

Per la prima parte del test, i visitatori sono non registrati; Questi vengono fatti entrare nella mostra, *ma non fino a riempirla*: infatti dopo averne fatti entrare solo 6, creo forzatamente una mostra che sovraccarica le sale del museo, costringendo l'amministratore a chiudere la prima mostra che ha creato. *Si vuole verificare che la prima mostra, quella creata in modo automatico, è stata chiusa.*

```

1  public class TestObserver{
2      ...
3      public void testFinal(){
4          museo.addBilancio(amministratore, 5000);
5          assertEquals(2, museo.getMostre().size());
6          Mostra m = museo.getMostre().iterator().next();
7          int postiPre = m.getPostiRimasti();
8          Visitatore[] visitors = new Visitatore[500];
9          for(int j = 0; j<visitors.length; j++)
10             visitors[j] = new Visitatore(1000);
11         for (int i = 0; i < 6; i++)

```

```

12     museo.vendiTicketMostraFisica(visitors[i], m);
13     assertTrue(postiPre != m.getPostiRimasti());
14     IncaricoMostra im = amministratore.forceStrategyExecution(2, 20, true);
15     assertTrue(m.isTerminata());
16     assertEquals(2, museo.getMostre().size());
17     ...
18 }
19 }

```

Listing 33: Prima parte test observer

Questo prova che la mostra iniziale è terminata. Quindi in questo momento ci sono attive due Mostre: quella creata forzatamente (e di cui abbiamo il riferimento tramite l'*IncaricoMostra*), e la seconda creata in modo automatico dall'amministratore.

Nella seconda parte del test, dopo aver *registrato tutti i visitatori dell'array*, prendo la prima mostra disponibile, coincidente con la prima per ordine di creazione (a garanzia di questo ho il fatto che gli incarichi sono salvati in un `LinkedHashSet<IncaricoMostra>`). In pratica prendo la seconda mostra che aveva creato in automatico l'amministratore, e di questa acquisto tutti i biglietti, usando tutti i *VisitatoriRegistrati* necessari a riempirla. Come risultato di questo, avrò la chiusura di quella mostra, che verificherò tramite un'asserzione.

```

1 public class TestObserver{
2     ...
3     public void testFinal(){
4         ...
5         m = museo.getMostre().iterator().next();
6         for(int i = 0; i < visitors.length; i++)
7             museo.registraVisitatore(visitors[i]);
8         for(int j=0; j < visitors.length; j++)
9             museo.vendiTicketMostraVirtuale(visitors[j], m, false);
10        assertTrue(m.isTerminata());
11        assertEquals(2, museo.getMostre().size()); //piRITARE
12        ...
13    }
14 }

```

Listing 34: Seconda parte test observer

Nella ultima parte, verifico di chiudere anche quella creata forzatamente.

```

1 public class TestObserver{
2     ...
3     public void testFinal(){
4         ...
5         m = im.getMostra();
6         for(int i=0; i < visitors.length; i++)
7             museo.vendiTicketMostraVirtuale(visitors[i], m, true);

```

```
8      assertTrue(m.isTerminata());
9      assertEquals(2, museo.getMostre().size());
10   }
11 }
```

Listing 35: Terza parte test observer

L'asserzione `assertEquals(2, museo.getMostre().size())` mi dice che ci sono sempre 2 mostre attive dentro il museo. Questo è vero solamente se l'Observer fa il suo lavoro, dimostrando che il sistema funziona e che le implementazioni sono corrette.

## 14 Conclusione test

L'esecuzione di tutti i test e la loro verifica di funzionamento, suggerisce che si possano implementare le classi descritte.