Museo 2.0

Giuseppe Parrotta

Elaborato per il corso di
Ingegneria del Software

Dipartimento di Ingegneria Informatica
Università degli Studi di Firenze
07/04/2021

Part I
Progettazione

1 Introduzione

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

L'idea è quella di realizzare un software che permetta all'amministratore di un museo di creare delle mostre di opere d'arte, allestendo alcune delle tante sale a disposizione del museo, e gestendo la vendita dei ticket di ingresso.

Questa idea esplode in vari sotto-casi, uno tra questi è permettere alle mostre di essere svolte anche online e di potere essere fruibili a una sola parte di visitatori, cioè quelli che sono registrati.

Un altro sotto-caso è quello di permettere a un museo di realizzare mostre anche usando opere di altri musei, prendendole in affitto dopo aver pagato una certa somma di denaro.

Infine, ho pensato che la creazione di una Mostra possa essere anche fatta in modo automatico, prevedendo che l'amministratore possa essere automatico del sistema e quindi che elabori strategie in base allo stato del museo.

Per la progettazione del software sono stati impiegati i seguenti strumenti: Class Diagrams, Sequence diagrams, Use Case Diagram e Activity diagrams, Mockups. Invece, per quanto riguarda l'implementazione, sono stati fondamentali i design patterns Observer, Strategy e Singleton.

Per testare il codice ho usato la suite JUnit.

2 Definizione del problema

Si vuole realizzare una applicazione che permetta all'Amministratore di un Museo di stanziare delle somme di denaro, per poter organizzare una Mostra di Opere d'Arte.

Volendo, l'Amministratore che sta preparando una Mostra può decidere di noleggiare alcune delle Opere d'arte da altri musei, ammesso che quelle opere non risultino già affittate in qualche altra Mostra. Una volta che le opere sono state scelte ed eventualmente noleggiate, chiederà a un Impiegato Organizzatore di organizzare la Mostra. La Mostra può essere Fisica, quindi fatta in alcune Sale del Museo, o Virtuale e quindi fruibile ANCHE dal portale del Museo. Il numero di posti in questo caso aumenta, avendo a disposizione un certo numero di posti virtuali, proporzionali alla potenza dei server del Museo. L'Organizzatore avrà il compito di selezionare le Sale necessarie per creare la Mostra, e di selezionare un equipe di Impiegati che dovranno svolgere i seguenti compiti: pulizia delle Sale Fisiche, spostare l'Opera d'Arte, creare Pubblicità e nel caso in cui la Mostra sia anche Virtuale, anche l'allestimento del Portale Web.

La Mostra deve essere dichiarata conclusa se richiesto dall'Amministratore, oppure quando raggiunge la quota di incassi prevista dall'Organizzatore. Anche l'Amministratore può chiedere la chiusura di una Mostra: in questo caso, l'Organizzatore farà uno storno dei soldi dei Ticket venduti per quella Mostra meno una trattenuta in percentuale. Alla chiusura delle Mostre, le eventuali opere noleggiate andranno restituite al legittimo proprietario. L'Amministratore chiede la chiusura di una Mostra quando vede che le sale sono quasi tutte occupate.

Gli eventuali Visitatori delle Mostre possono essere di due tipi: Visitatori o Visitatori Registrati. I Visitatori (non Registrati) possono visitare il Museo, oppure possono partecipare a Mostre che quel Museo ha allestito solo se sono rimasti posti dentro le Sale Fisiche dedicate a quella Mostra. I Visitatori Registrati hanno tutte queste possibilità con in più la possibilità di Occupare i posti previsti dalle 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 Manufatti

3.1 Class Diagram concettuale del Modello di Dominio

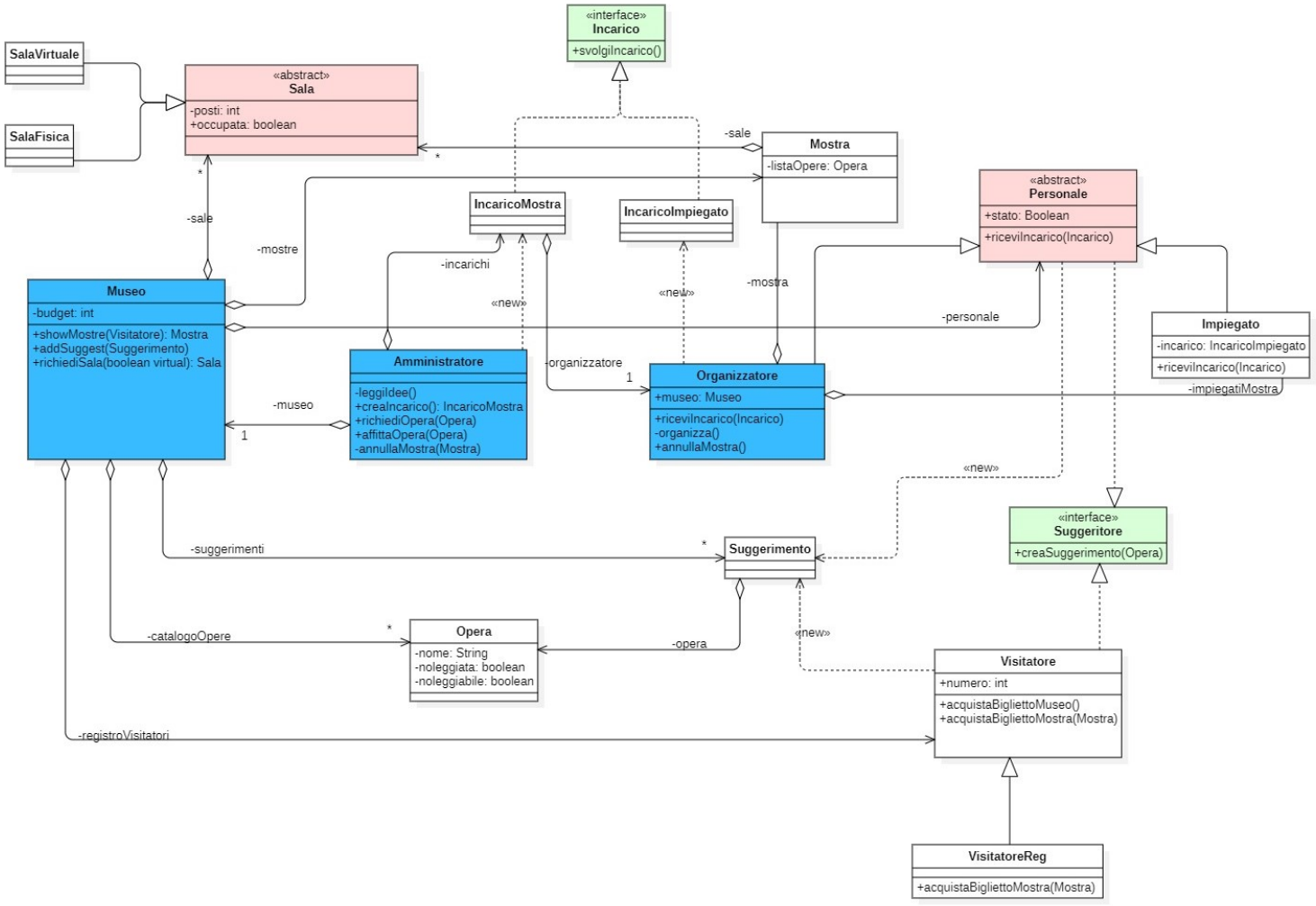

Figure 1: Modello di dominio

3.2 Package Diagram

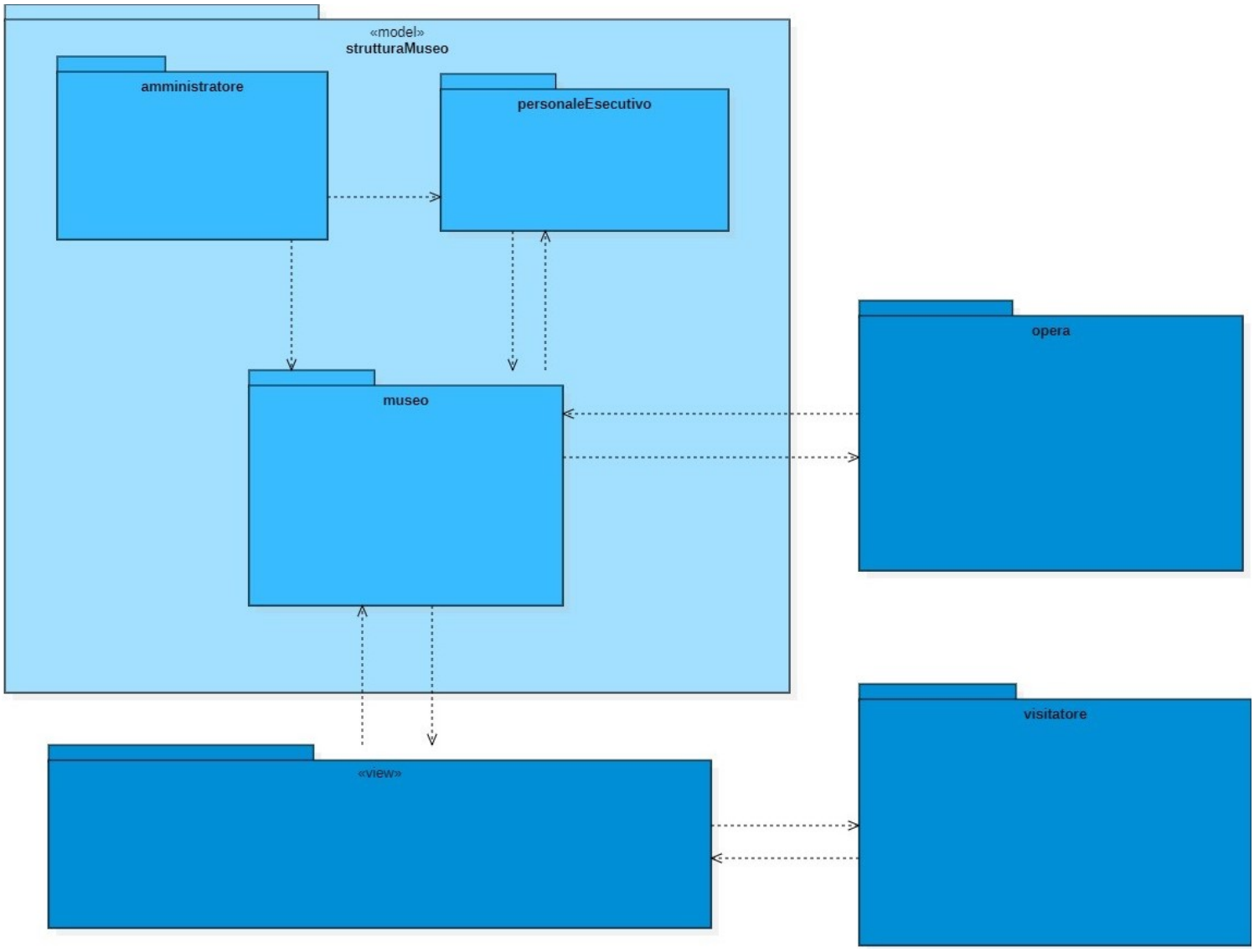

Figure 2: Package diagram

3.3 Use Case Diagram

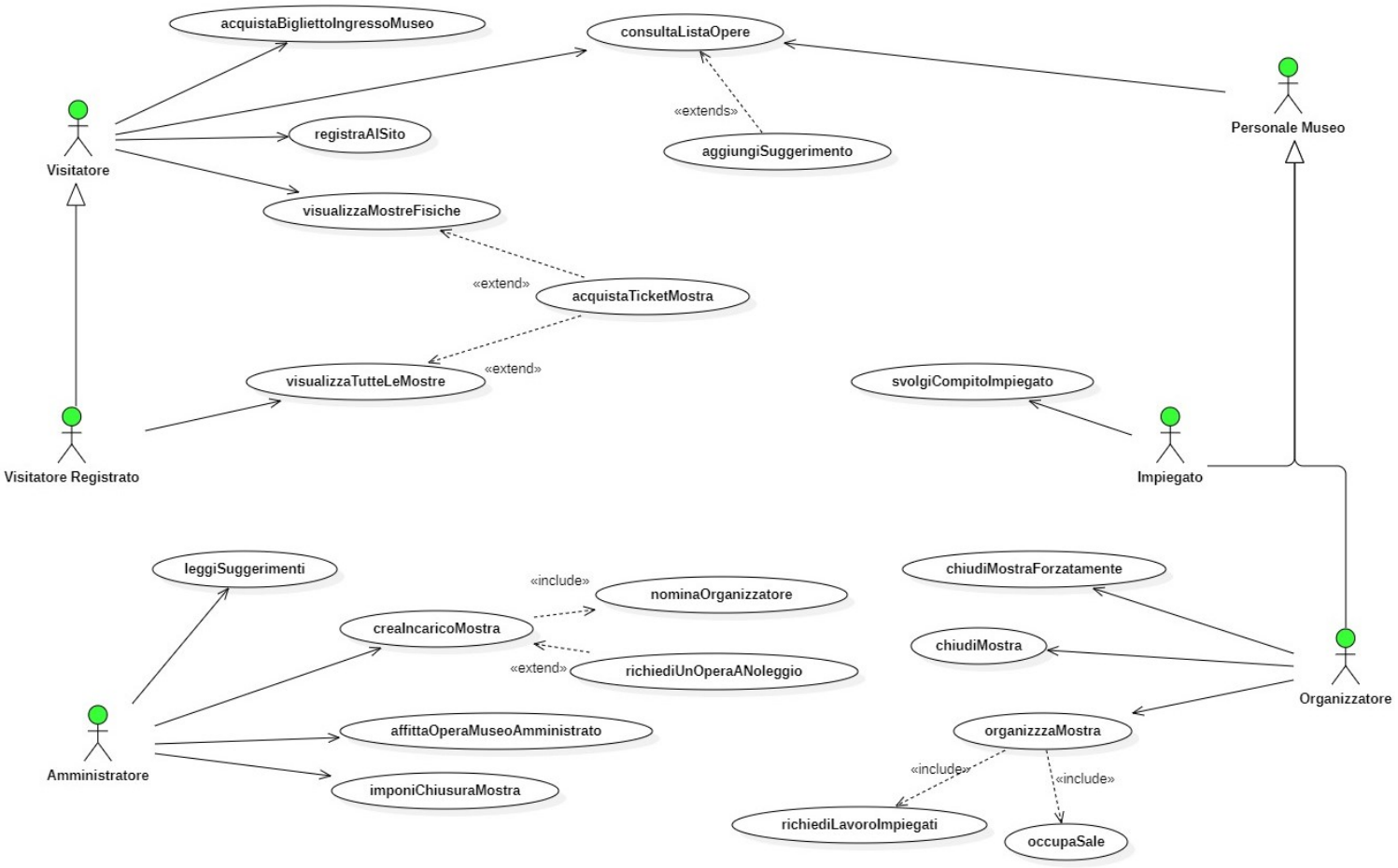

Figure 3: Use Case Diagram

Questi sono i casi d'uso identificati. Alcuni di questi sono dettagliati in degli Use Case templates

3.4 Use Case Templates

Name	UCAdmin#2 - Crea incarico Mostra
History	
Source	
Level	Users goal
Description	L'amministratore del Museo crea un incarico per una mostra
Scope	Creazione Mostra
Actors	Amministratore
Pre - Conditions	L'utente ha fatto l'accesso al sistema come amministratore
Post- Conditions	Viene emesso un incarico e fornito a un Organizzatore
Normal flow	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
Alternative flows	6b L'utente cambia opere
References	UCAdmin#3 – Nomina Organizzatore
Non functional requirements	

Name	UCOrganiz#3 - Organizza Mostra
History	
Source	
Level	Users goal
Description	L'organizzatore sviluppa l'incarico ricevuto
Scope	Creazione Mostra
Actors	Organizzatore (primary), Museo(secondary)
Pre - Conditions	L'amministratore ha creato l'incarico
Post- Conditions	Il Museo dispone di una mostra in più
Normal flow	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
Alternative flows	8b L'utente annulla 9b Torna al punto 4
References	UCOrganizz#4 -Richiedi Impiegati UCOrganizz#5 - Occupa Sale
Non functional requirements	

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

3.5 Mockups

3.5.1 Pannello dell'amministratore

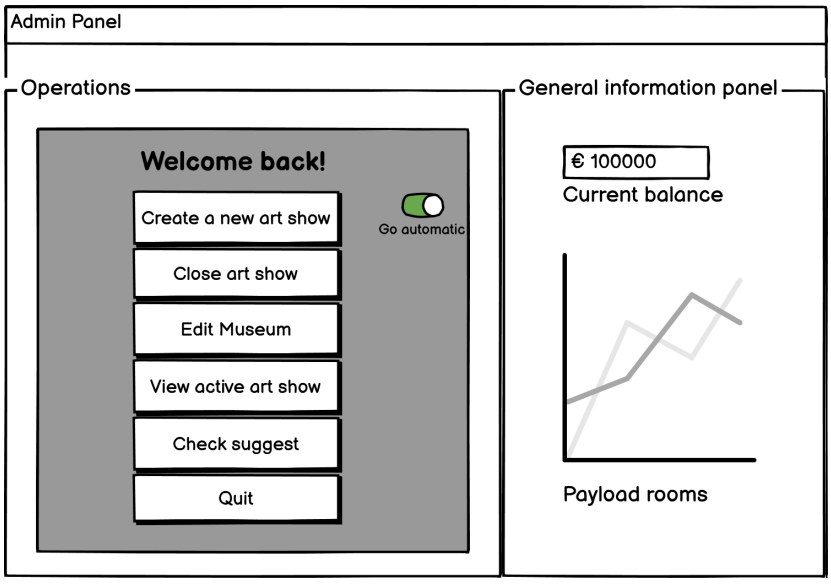

Figure 5: Admin Panel

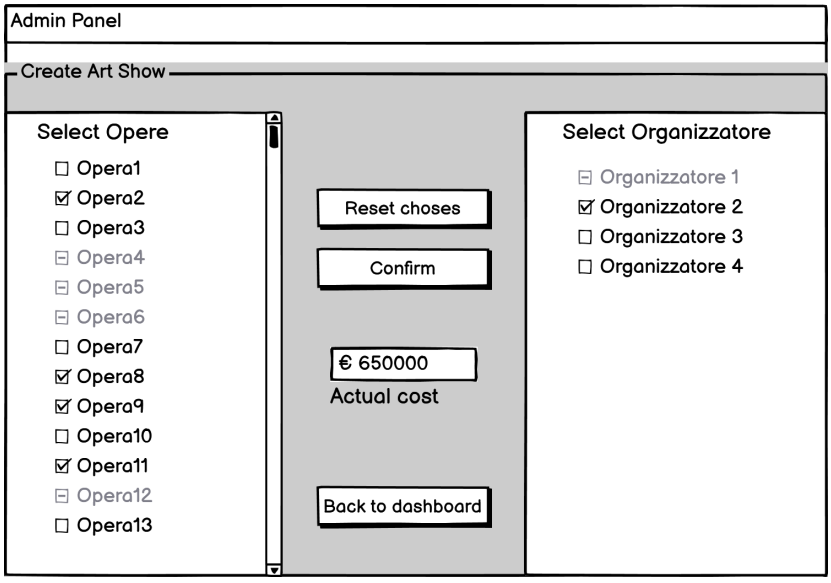

Figure 6: Creazione Mostra

3.5.2 Pannello Impiegato e Organizzatore

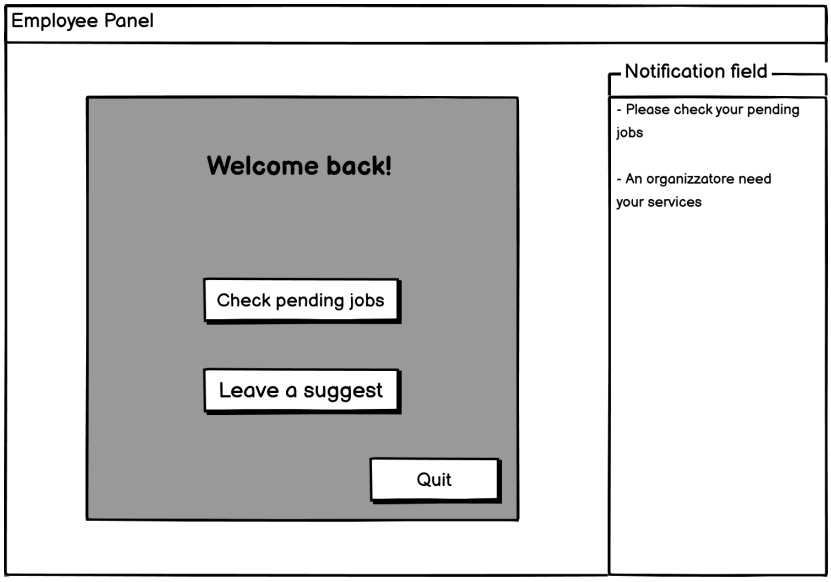

Figure 7: Dashboard Impiegato

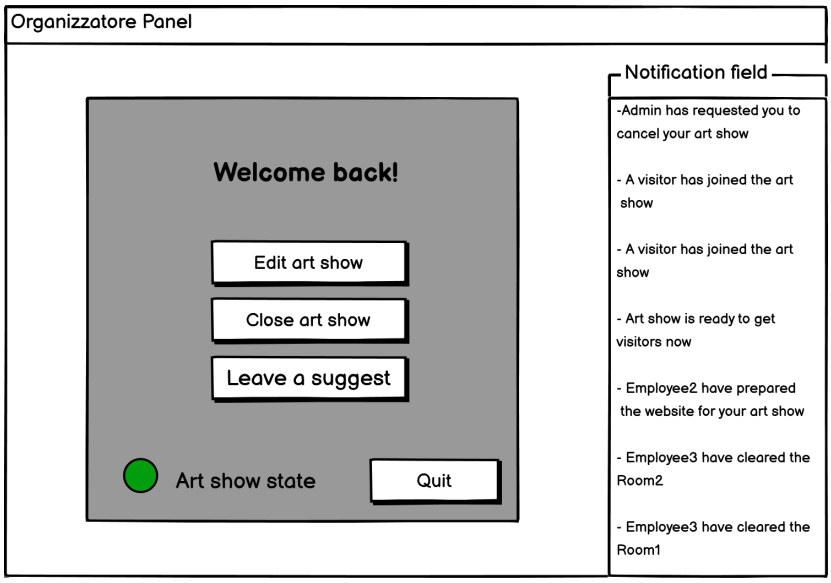

Figure 8: Dashboard Organizzatore

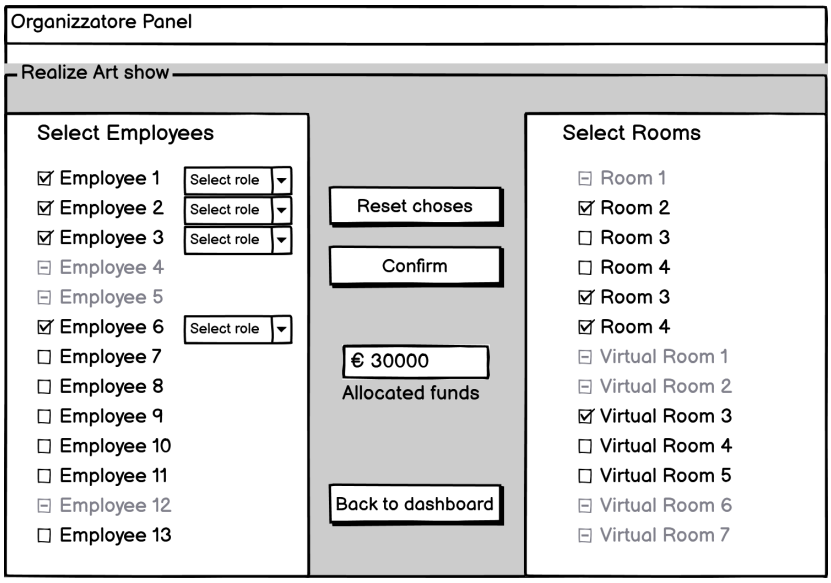

Figure 9: Realizzazione Mostra

3.5.3 Pannello Visitatori

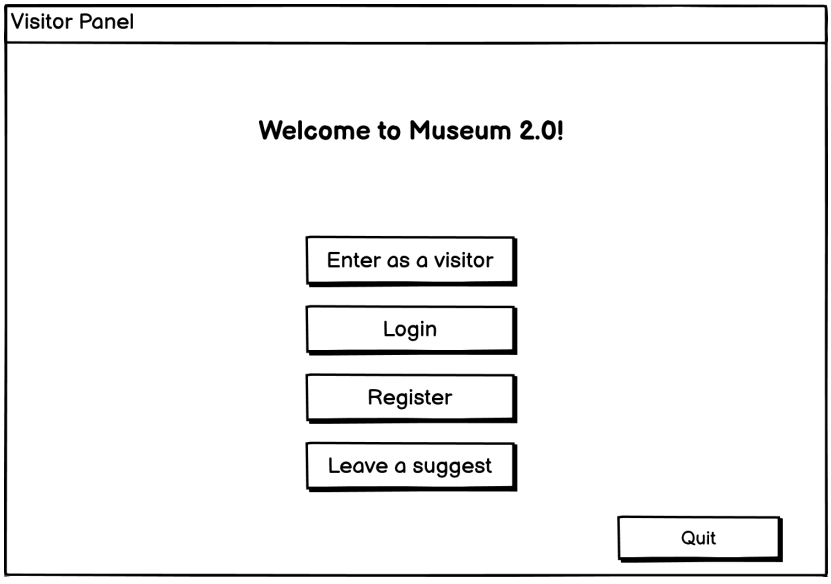

Figure 10: Pannello visitatori

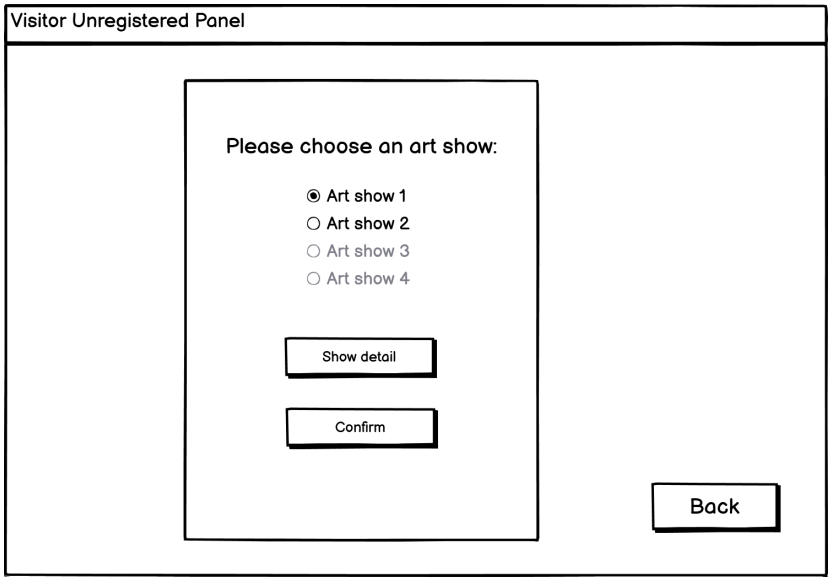

Figure 11: Pannello scelta Mostra

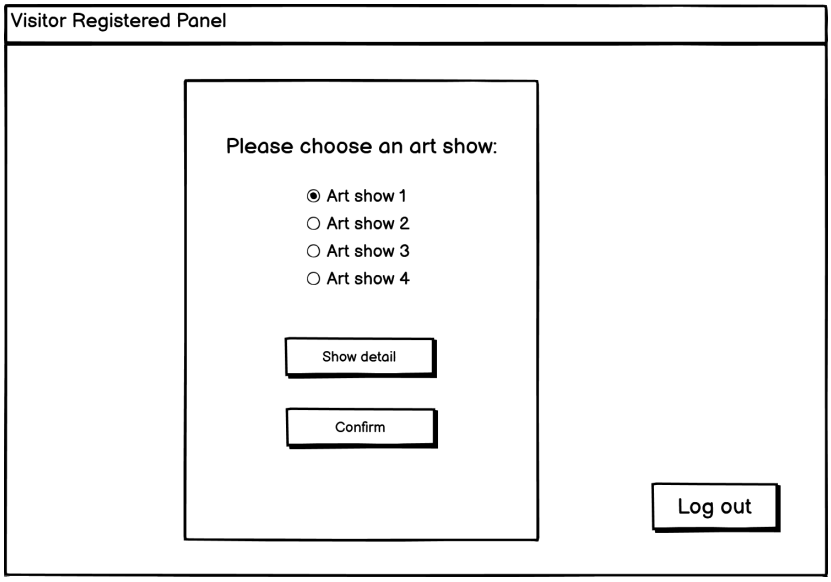

Figure 12: Pannello scelta Mostra per visitatori registrati

3.6 Activity Diagram

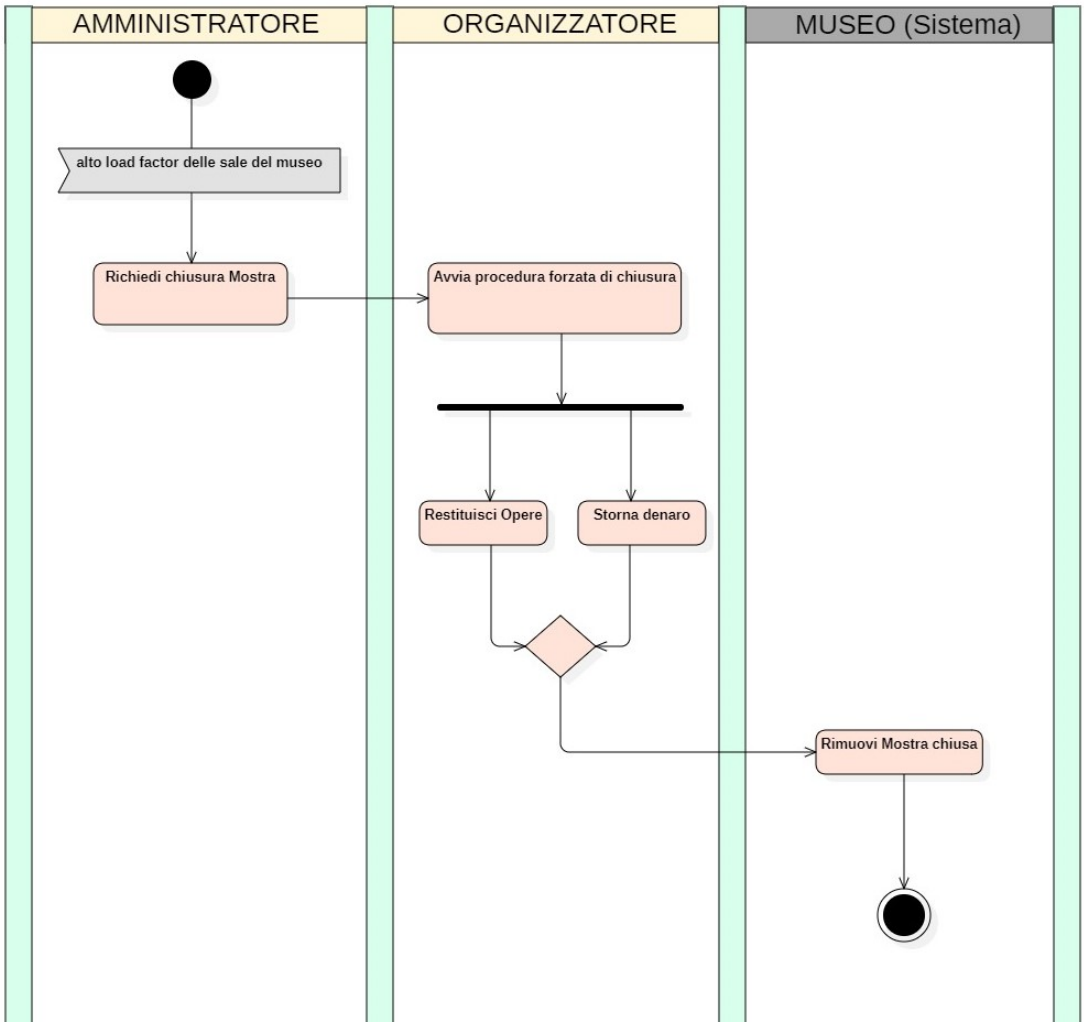

Figure 13: Activity diagram della procedura di chisura forzata di una mostra

Part II
Implementazione

Le classi che ho scritto sono state tutte accorpate in vari package, per come visto nel package diagram. Quello più grande, *strutturaMuseo*, è suddiviso in 3 più piccoli: *personale*, *amministratore* e *museo*. Gli altri due (tre se si considera il package view) sono invece a sè stanti. e verranno descritti dopo questi 3.

4 Package museo

Qui dentro sono contenute tutte quelle classi che chiamerei di struttura. Oltre alle varie classi di supporto, quale la classe $Mostra$ e la classe $Suggerimento$, c'è la classe $Museo$ che ha un ruolo centrale: tutte le classi forniscono un servizio alla classe Museo. Infatti oggetti quali $Tickets, Sale$, e le sopra citate Mostre e Suggerimenti, hanno tutte uno spazio più o meno di rilievo nella struttura del $Museo$. Inizio a definire le classi di contorno per poi concludere con la classe Museo. Tutte le classi, eccezion fatta per Museo, Sala e per l'interfaccia Ticket, sono final per evitare che vengano estese.

4.1 Classi Sala, SalaFisica e SalaVirtuale. Classe Suggerimento

Le classi $SalaVirtuale$ e $SalaFisica$ sono entrambe specializzazioni della classe astratta $Sala$. Sono tutti oggetti che definirei inerti. Vengono create dal costruttore di Museo e inserite in un insieme (Set) immutabile di oggetti. Vengono passati i riferimenti agli organizzatori solo per cambiare il loro stato, da libero a occupato e vice versa. Infine, il costruttore per i due tipi di sale ha visibilità $package - protected$.

Gli oggetti di classe $Suggerimento$ sono degli oggettini che hanno giusto due riferimenti, uno a un oggetto di tipo $Opera$ (si vedrà nella sezione dedicata di cosa si tratta) e uno a un generico oggetto $Suggeritore$, cioè una interfaccia che implementano tutti quegli oggetti che hanno la possibilità di creare suggerimenti. Questi due riferimenti sono passati al costruttore ($package - protected$) e che una volta creati sono sostanzialmente immutabili, per via dell'assenza di metodi setter e della classe Suggerimento che è dichiarata final, quindi inestensibile. Una volta creati, vengono inseriti dentro una struttura dati di tipo ArrayList di Museo.

4.2 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.

4.3 Classe Mostra

Preciso che la mostra è vista più come un evento, e che venga chiusa una volta che ha raggiunto il numero di biglietti richiesti. La classe *Mostra* è una classe che descrive una mostra di qualsiasi tipo. Contiene tutti i metodi getter e setter che servono per la descrizione, composta da field quali l'incasso, cioè i soldi che ha incassato, i posti rimasti, una lista di visitatori e un Set di sale che ha allocate. Inoltre, Mostra estende *Observable* in quanto è l'*Organizzatore* (che implementa *Observer*) che ha la responsabilità di chiuderla. Per come è concepita in questo momento, la mostra viene chiusa quando raggiunge l'incasso, quindi mostro un pezzetto di codice che può essere importante per capire questo meccanismo.

```
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   public otherFunctions();
17 }
```

Listing 1: Metodi incassaTicket e checkRaggiuntoIncasso

Quando l'Organizzatore viene chiamato dal metodo `notifyObservers()`, ha il compito di eseguire un'azione per la Mostra. Per come è attualmente implementato, ha solo il compito di chiudere la mostra, ma questo è un meccanismo che può essere facilmente riadattato.

Voglio specificare che non è questo il metodo che vende il ticket, bensì `incassaTicket()` è chiamato in forward dal metodo che gestisce la transazione di denaro, che è un metodo della classe Museo.

4.4 Classe Museo

La classe *Museo* ha tanti compiti: essa infatti deve gestire il pagamento di *Tickets*, fornire le Sale, le Opere, gli Impiegati e gli Organizzatori, mostrare le

Mostre che sono attive, registrare eventuali visitatori. Ha inoltre il compito di comunicare con l'*Amministratore* il suo stato: Museo estende la classe astratta *Observable*, e *Amministratore* implementa *Observable*. Questi due oggetti comunicano sia attraverso un un Observer pull che un Observer Push. Questo meccanismo verrà descritto successivamente.

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

4.4.1 Componente rivolta ai Acquirenti e Suggeritori

Ricordo che ogni *Visitatore* è implementazione delle interfacce *Acquirente* e *Suggeritore*, e i vari Impiegati sono implementazioni di *Suggeritore*.

Tra i field che servono per questo scopo, di particolare interesse è la mappa che accumula Ticket venduti. Un esempio di come vengono aggiunte è mostrata di seguito. Fanno parte di questo gruppo quei metodi che permettono **l'acquisto di Ticket**, quelli che permettono la **consultazione del Catalogo opere**, quelli che permettono la **registrazione degli Acquirenti** e quelli che permettono di **consultare le mostre attive**.

Per quanto riguarda l'acquisto, ho creato la classe *NoMoneyException* che specializza *RuntimeException* ed è un'eccezione che viene lanciata quando non si può effettuare una transazione. I metodi per l'acquisto:

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

Di questi metodi, riporto le implementazioni del primo e del terzo; il secondo 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` 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            else
13              mostra.togliPostoFisico();
14           } else
15            if (mostra.getPostiFisiciRimasti() > 0)
16              mostra.togliPostoFisico();
17            else
18              mostra.togliPostoVirtuale();
19          mostra.incassaTicket(visitatore);
20          Ticket tmp = new TicketMostraFisicaEVirtuale(visitatore);
21          ticketMuseoVenduti.get("Ticket␣Fisica-Virtuale").add(tmp);
22          visitatore.addTicket(tmp);
23          return true;
24        } else
25          visitatore.ottieniRimborso(mostra.getCostoTicket());
26       }catch(NoMoneyException e){return false;}
27     } else{
28       try {
29         visitatore.paga(mostra.getCostoTicket());
30         if (mostra.getPostiRimasti() > 0) {
31           mostra.togliPostoFisico();
32           mostra.incassaTicket(visitatore);
33           Ticket tmp = new TicketMostraFisica(visitatore);
34           ticketMuseoVenduti.get("Ticket␣Fisica").add(tmp);
35           visitatore.addTicket(tmp);
36           return true;
37         }
```

```
38          else
39            visitatore.ottieniRimborso(mostra.getCostoTicket());
40        }catch (NoMoneyException e){return false;}
41      }
42      return false;
43    }
44    ...
45  }
```

Listing 3: Metodo vendiTicketMostraVirtuale

Questo ha lo scopo di prelevare il costo del biglietto dal portafogli dell'Acquirente, chiamando da quest ultimo il metodo `paga` che può lanciare l'eccezione `NoMoneyException` la quale deve essere gestita; inoltre, si occupa di mettere la somma prelevata dal portafogli dell'Acquirente nella cassa della Mostra; infine si occupa di aggiornare lo stato della Mostra togliendo un posto da quelli acquistabili. La flag `preferiscoPostoVirtuale` serve per i *VisitatoriRegistrati* per scegliere una preferenza. 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 chiama il metodo appena descritto mettendo la flag su `false`.

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

- `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 è una string generata in modo automatico solo ai fini di velocizzare la fase ti 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

Una serie di caratteri casuali formano la chiave per mappare il *VisitatoreRegistrato*; l'oggetto mappato è un nuovo oggetto con gli stessi soldi che aveva da non registrato. Un fatto interessante è, nel secondo metodo, la chiamata di *Observer*: infatti all'Observer viene mandato un oggetto di tipo *StatoMuseo*, che ha tutti i parametri null eccetto il nuovo *Suggerimento*. Questo oggetto serve per differenziare i flussi di informazione di cui l'Amministratore ha bisogno per funzionare.

4.4.2 Componente rivolta agli utenti interni

In questa componente identifico i metodi per il funzionamento della struttura stessa del Museo, quindi tutta una serie di getter-setter per impostare alcuni parametri del Museo, o per riceverne. 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()`

Molti di questi metodi sono autoesplicativi. Nei metodi `getOrganizzatori(boolean)` `getImpiegati(boolean)` e `getSale(boolean)`, il loro parametro serve per chiedere se ottenere tutti gli *Organizzatori*, tutti gli *Impiegati* e tutte le *Sale*, oppure solo quelle libere (liberi). Il metodo `updateLoadFactorSale()` è un metodo privato, e serve per ricalcolare il fattore di carico, un campo dell'oggetto Museo. 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

Queste due funzioni sono le uniche che chiamano il metodo updateLoadFactorSale, e ha senso calcolarlo appunto solo nei casi in cui effettivamente varia. Entrambi questi due metodi preparano e mandano agli *Observer* che sono aggiunti questi oggetti StatoMuseo, i quali verranno formalizzati successivamente.

Anche altri tra quei metodi chiamano gli Observers in ascolto: parlo dei metodi `setBilancio`,`addBilancio` e `prelevaBilancio`, ma questi metodi mandano una notifica di tipo pull: infatti sarà poi l'Amministratore a prendersi i dati di cui ha bisogno.

La classe **StatoMuseo** è un oggetto creato solo per comunicare con gli Observers. Ha i seguenti campi:

- Suggerimento
- bilancioMuseo
- loadFactorSale
- mostraChiusa
- museoIsReady

Dei primi tre si è già parlato. Gli ultimi due campi, sono mostraChiusa e museoIsReady. Il primo tra questi è un oggetto di tipo Mostra, che viene mandato all'Amministratore quando una mostra viene chiusa e quindi cancellata dal Museo. La seconda è una flag, che comunica quando il Museo è pronto, ovvero quando l'Amministratore può valutare se creare altre mostre: infatti se questa flag è `true` l'Amministratore può fare un'*azioneAmministratore*, che verrà illustrata nella sezione apposita.

5 Package amministratore

Nel package amministratore si trovano le classi delle varie *strategie*, la classe *Amministratore* e la classe $IncaricoMostra$. Le strategie servono all'amministratore per creare un IncaricoMostra a seconda dello stato del museo. IncaricoMostra ha un costruttore $package - protected$, e solo l'amministratore ha la possibilità di crearli. Anche le strategies hanno la stessa visibilità per il costruttore.

5.1 Classe IncaricoMostra

Questa class estende *Observable* e implementa *Incarico*, che è un'interfaccia che offre il metodo `svolgiIncarico`. Quando l'Organizzatore riceve l'incarico, diventa un Observer anche di questo oggetto. In pratica, l'Organizzatore osserva sia la Mostra che ha creato, che l'IncaricoMostra che gli è stato affidato. Questo fa sì che l'Amministratore, avendo tutte le istanze di IncaricoMostra, possa scegliere effettivamente quando dire all'Organizzatore di chiudere una mostra. A tale proposito riporto un pezzo di codice:

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

```
6     }
7     ...
8   }
```

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

Voglio fare notare che il metodo è *package − protected*, come il costruttore. Questo garantisce che sia solo l'amministratore a chiamare questo metodo. Altri metodi dedicati solo all'Amministratore sono `setOpere(Set <Opere>` e `setOrganizzatore(Organizzatore)`.

Quando viene stanziato un incarico, esso viene fornito di un certo quantitativo di denaro, che viene preventivamente trasferito dalla cassa del museo a quella dell'Incarico Di fatto, è come se venisse stanziato. Quindi l'organizzatore ha a disposizione il denaro presente nell'IncaricoMostra per organizzare la Mostra. Per prendere quel denaro si avvale del metodo `prelevaDenaro(int)`, che prevede semplicemente di togliere, se possibile, la quantità mandata come parametro, dalla cassa dell'Incarico.

I field più importanti della class sono:

- `ArrayList<Opera> 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, o è stata chiusa. In ogni caso, se `killable` è false, la Mostra sicuramente non è fruibile.

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

Per chiarezza, riporto anche il costruttore:

```
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 }
```

5.2 Strategie

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

Le strategie possono essere di due tipi:

- **Strategie creazionali**
- **Strategie non creazionali**

Riporto tutte le implementazioni dello strategyMethod.

5.2.1 Strategie creazionali

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

- il bilancio
- le Opere richieste

Per quanto riguarda le Opere: è proprio in queste strategie che avviene il noleggio. A questo fine, viene richiesto al museo l'oggetto *GestoreOpere*, che si occupa di aggiornare lo stato delle Opere. Voglio aggiungere che anche nel caso in cui l'opera la stia richiedendo il museo proprietario, si debba comunque passare attraverso il GestoreOpere per cambiarne lo stato. Per le strategie creazionali: lo `strategyMethod` è composto da 3 parti: inizializzazione, dove si creano le liste vuota e l'IncaricoMostra non ancora decorato; dopo vengono prese le Opere attraverso i vari cicli while; infine si fanno eventualmente alcune operazioni di chiusura (come nella OnSuggestStrategy) e si fa il return di questo Incarico, completo di Opere d'Arte e budget.

Questo incarico va a finire all'Amministratore, il quale si occupa di assegnargli un Organizzatore e infine di passarlo a quest ultimo. **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 6: strategyMethod di LowBudgetStrategy

OnSuggestStrategy è la strategy che viene scelta quando il numero di suggerimenti è abbastanza alto.

```
class OnSuggestStrategy{
  ...
  public IncaricoMostra strategyMethod(Museo museo) {
    ArrayList<Opera> opereResetSuggerimenti = new ArrayList<>();
    GestoreOpere gestoreOpere = museo.getGestoreOpere();
    int numeroOpereAutoStrategy = 5;
    IncaricoMostra incarico = new IncaricoMostra(200, true);
    ArrayList<Opera> opereNuovoIncarico = new ArrayList<>();
    ArrayList<Opera> opereMuseo =
         (ArrayList<Opera>) museo.getOpereMuseo();
    Iterator<Opera> iterator = opereMuseo.iterator();
    while(opereNuovoIncarico.size() < 3
          && iterator.hasNext()){
      Opera operaMuseo = iterator.next();
      if(!operaMuseo.isBusy())
        gestoreOpere.affittaOperaAMuseo(operaMuseo, incarico, museo);
        opereNuovoIncarico.add(operaMuseo);
    }
    iterator = opereRichieste.iterator();
    while(opereNuovoIncarico.size() < numeroOpereAutoStrategy
          && iterator.hasNext()){
      Opera opera = iterator.next();
        if(opera.getProprietario() != museo)
          if(!opera.isBusy()) {
            try {
              museo.prelevaBilancio(this, opera.getCostoNoleggio());
              gestoreOpere.affittaOperaAMuseo(opera, incarico, museo);
              opereNuovoIncarico.add(opera);
              opereResetSuggerimenti.add(opera);
            } catch (NoMoneyException e) {
            System.err.println(e.getMessage());}
          }
    }
    amministratore.resetSuggerimentiPerOpere(opereResetSuggerimenti);

    incarico.setOpereMostra(opereNuovoIncarico);
    return incarico;
  }
}
```

Listing 7: strategyMethod di OnSuggestStrategy

PersonalStrategy

```
class PersonalStrategy{
```

```
2       ...
3       GestoreOpere gestoreOpere = museo.getGestoreOpere();
4       ArrayList<Opera> opereNuovoIncarico = new ArrayList<>();
5       ArrayList<Opera> opereMuseo = (ArrayList<Opera>) museo.getOpereMuseo();
6       int media = 0;
7       for(Opera opera:opereMuseo)
8         media += opera.getCostoNoleggio();
9       media = media/opereMuseo.size(); //  costo opere + operai + sicurezza
10      IncaricoMostra incarico =
11                new IncaricoMostra(numeroOpere*media + 60 + numeroOpere*10,
12                ancheVirtuale);
13      // Voglio solo un'opera del Museo proprietario
14      Iterator<Opera> iterator = opereMuseo.iterator();
15      while(opereNuovoIncarico.size() == 0 && iterator.hasNext()){
16        Opera opera = iterator.next();
17        if(!opera.isBusy()) {
18          gestoreOpere.affittaOperaAMuseo(opera, incarico, museo);
19          opereNuovoIncarico.add(opera);
20        }
21      }
22      iterator = museo.getCatalogoOpere().iterator();
23      while(iterator.hasNext() && opereNuovoIncarico.size() < numeroOpere) {
24        Opera opera = iterator.next();
25        if (!opera.isBusy() && opera.getProprietario() != museo) {
26          try {
27            museo.prelevaBilancio(this, opera.getCostoNoleggio());
28            gestoreOpere.affittaOperaAMuseo(opera, incarico, museo);
29            opereNuovoIncarico.add(opera);
30          } catch (NoMoneyException e) {
31            System.err.println(e.getMessage());
32          }
33        }
34      }
35      incarico.setOpereMostra(opereNuovoIncarico);
36      return incarico;
37    }
38  }
```

Listing 8: strategyMethod di PersonaStrategy

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` creerà le PersonalStrategy a seconda dello stato dell'Amministratore.

5.2.2 Strategie non creazionali

Queste due strategie sono due casi particolari, e in comune hanno che tutte e due restituiscono `null` Servono entrambe a svolgere funzioni diverse dalla creazione. **IdleStrategy** non fa assolutamente nulla. L'amministratore è in stato di quiete e viene impostata in caso non dovessi 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. Sostanzialmente chiama il metodo di *IncaricoMostre* visto precedentemente, `forzaChiusuraMostra()`.

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

```
21    public void setReady(){
22      this.strategyHasKilled = false;
23    }
24
25    public boolean isReady(){
26      return !strategyHasKilled;
27    }
28  }
```

Listing 11: Classe KillMostreStrategy

Vista la sua brevità, l'ho riportata completamente. E' importante notare la flag `strategyHasKilled`, che mi dice se la strategia ha già "ucciso", cioè se il processo è stato chiuso. Questa funziona tipo un lock, cioè viene attivata, e poi solo dall'esterno può essere sbloccato, e questo avviene quando l'amministratore chiama il metodo `setReady()`.

Piccola annotazione: la strategia è pronta quando non ha ancora ucciso. Concettualmente, legge gli incarichi che sono stati creati, tova il primo che risulta *unkilled* e avvia la procedura di chiusura forzata.

5.3 Classe Amministratore

La classe Amministratore ha il ruolo di gestire il Museo. Per fare questo, implementa l'interfaccia *Observer*: grazie al metodo `update` dell'interfaccia, aggiorna il suo stato e successivamente esegue un'azione in modo automatico grazie all'implementazione del pattern *Strategy*. Quindi: `update` aggiorna, e `strategyMethod` esegue.

Sono presenti un certo numero di flag, e un certo numero di attributi, che vengono aggiornate insieme allo stato del Museo:

- `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`: questa ultima deve mantenere uno stato, e quindi ha bisogno di un campo suo.

Prima di parlare direttamente del metodo `update`, parlo dei suoi metodi si supporto:

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

5.3.1 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 sopra di un certo numero, le *Opere* suggerite vengono messe in *richieste*. E' un metodo molto pesante da calcolare a ogni aggiornamento di update, quindi ho creato un counter che attiva questo metodo dopo

che si è arrivato a un certo numero di suggerimenti.

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 creerò 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 && mostreChiuse.size()
13        lockCreationIncaricoMostre = false;
14        ((KillMostreStrategy)killMostreStrategy).setReady();
15      }
16    }
17    ...
18  }
```

Listing 14: Metodo checkForLockRelease

Con questa funzione faccio un check per capire se posso rilasciare il 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 egual numero delle 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 = (Organizzatore) museo.getOrganizzatori(t
8        incaricoMostra.setOrganizzatore(organizzatore);
9        incarichiCreati.add(incaricoMostra);
10       organizzatore.setIncaricoAttuale(incaricoMostra);
11       organizzatore.svolgiIncaricoAssegnato();
12     }
13     else if(incaricoMostra != null){
14       chiusuraEmergenza(incaricoMostra);
15     }
16     return incaricoMostra;
17   }
18   ...
19 }
```

Listing 15: Metodo azioneAmministratore

Il return che fa è solo a fini di test.

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.restituisciOperaAffittata(o);
7       }
8       try {
9         museo.addBilancio(this, im.getBilancio());
10      }catch (Exception e){}
11    }
12    ...
13  }
```

Listing 16: Metodo chiusuraEmergenza

5.3.2 Il metodo update

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

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

Sono entrambe inizalizzate su false, e se alla fine del processo di aggiornamento saranno diventate true, sarà necessaria un'azione dell'amministratore. La fase di aggiornamento è 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 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 = true`
- chiamando update da fuori al museo, con parametri `null, null`; in questo caso si avvia una procedura forzata. `strongRestart = 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 le flag per prendere decisioni nella seconda parte della funzione, che riprendo ora:

- `needAdminOp`

- `strongRestart`
- `lock`
- `strategyHasKilled`
- `goAuto`

Lock è una flag dell'oggetto amministratore, come goAuto. Invece strategyHasKilled è quella flag che viene dalla strategia *KillMostreStrategy*.

```
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
12       // AMMINISTRATORE AUTOMATICO
13       updateSemaforo();
14
15       if (goAutomatico && needAdminOp) {
16         if (!semaphoreCreational) {
17           setStrategy();
18           boolean killerIsReady = ((KillMostreStrategy) killMostreStrategy).
19           if (strategy instanceof KillMostreStrategy && !killerIsReady) {
20             ;
21           } else
22             azioneAmministratore();
23
24         } else {
25           if (!lockCreationIncaricoMostre) {
26             setStrategy();
27             azioneAmministratore();
28           }
29         }
30       }
31     }
32   }
33   ...
34 }
```

Listing 18: Update - fase esecuzione

Nella parte esecutiva si vede come se faccio uno `strongRestart`, eseguo `azioneAmministratore()` dopo aver calcolato la strategia. 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()`.

6 Package personaleEsecutivo

In questo package, la classe *Impiegato* e la classe *Organizzatore* estendono la classe *Personale*. La classe più strutturata di questo package è la classe *Organizzatore*

6.1 Classe Organizzatore

Come già accennato questa classe implementa *Observer*, e ha un metodo `update(Observable o, Object arg)` 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 m, cioè la Mostra che ha creato, allora è automaticamente l'IncaricoMostra che ha mandato una notifica, e sappiamo che 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.restituisciOperaAffittata(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.

Nota: quando chiudi la mostra, Museo manda un `notifyObservers()`.

L'organizzatore organizza quando viene chiamato il metodo `svolgiIncarico()` dall'esterno: sarà proprio l'amministratore a chiamare quel metodo in `azioneAmministratore()`

`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  }
```

6.2 Altre classi

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

- *IncaricoImpiegato*(*abstract*)
- *Personale*(*abstract*)
- *Impiegato*
- *CreaPubblicità*
- *OrganizzaSitoWeb*
- *PulizieSaleFisiche*
- *SpostareOpera*

IncaricoImpiegato è una classe astratta che fa da superclasse per CreaPubblicità, OrganizzaSitoWeb, PulizieSaleFisiche e SpostareOpera e che implementa l'interfaccia *Incarico*.

7 Package Opera

Vi sono due classi in questo package:

- *Opera*
- *GestoreOpere*

7.1 Classe Opera

Opera è un oggetto semplice, parzialmente immutabile se non nel field dell'utilizzatore (cioè il Museo che lo sta noleggiando) e lo stato free oppure busy se occupata in qualche mostra. E' comunque 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

7.2 Classe GestoreOpere

In questa classe vi sono i metodi da usare per affittare un'opera 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));
```

con ovviamente non solo questa opera. Il costruttore Set.of, costruisce un insieme immutabile, e il field static public 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 22: 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.

8 Package Visitatori

Le classi che fanno parte di questo package sono oggetti molto semplici. 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 lancia un'eccezione se il visitatore non ha una cifra maggiore o uguale a quella passata come parametro.

9 Tests