PADSOF. Second deliverable.

Pablo Cuesta Sierra, Pablo Fernández Alegre and
Álvaro Zamanillo Sáez

Application: ***ARCH Theater Hall***

Contents

1 Class diagram

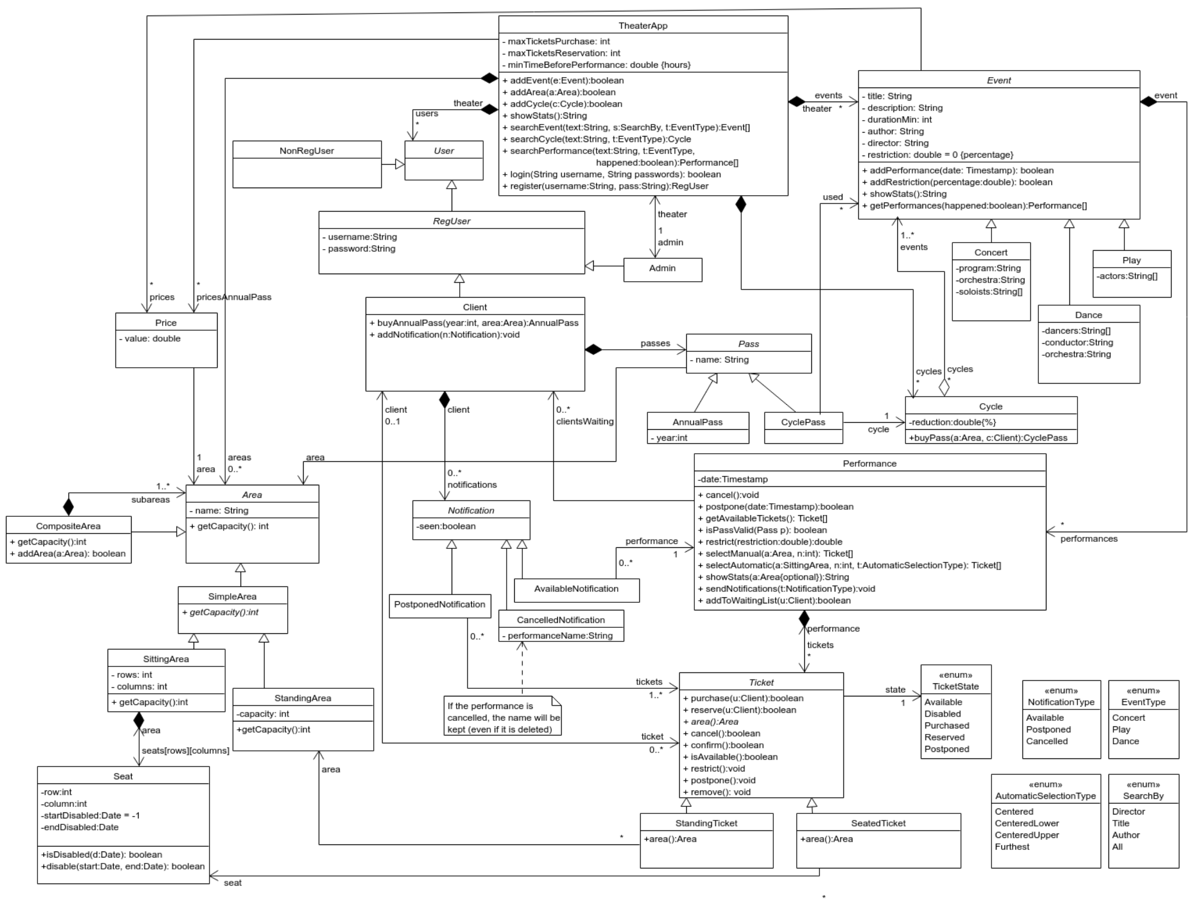


Figure 1: Class diagram

The theater app contains all the information of the system. It is a container of areas (all the areas of the theater), users, cycles and events. It has all the methods for searching (which can be used by any user). Also, for adding events or areas and showing stats of the events, which can be used only by the administrator of the theater. As private attributes, there are parameters that can be configured by the administrator like the maximun number of tickets per reservation, or the prices of the annual passes or each area.

Event is an abstract class with subclasses (one for each kind) that have specific attributes. Each event contains all its performances, and performances have a list of all its tickets, each of them with an attribute that indicates its state, depending on its availability. To satisfy the requirements, we have included methods to add new performances to an event, as well as postponing an cancelling them.

It is worth commenting the Ticket class as it is the link between clients and performances. When a performance is created, one ticket for each available spot in the theater (standing or seated) is created. This ticket can be purchased and reserved by clients, but they may also be disabled due to restrictions or because the seat of the ticket is disabled. The different states of this class will be explained with more detail in the state diagram. Moving on to the areas, we have stablished again an abstract class with subclasses. In this case there is an overriden method *getCapacity()* that has a different implementation for each subclass.

Finally, there are several enumeration classes in the diagram: almost all of them are used for restricting the parameter of methods. For instance, there is an enumeration for the different ways of searching for an event (by director, title or author) and another for specifying the criteria when selecting seats in an automatic selection.

2 State diagrams

2.1 State diagram of class: `Performance`

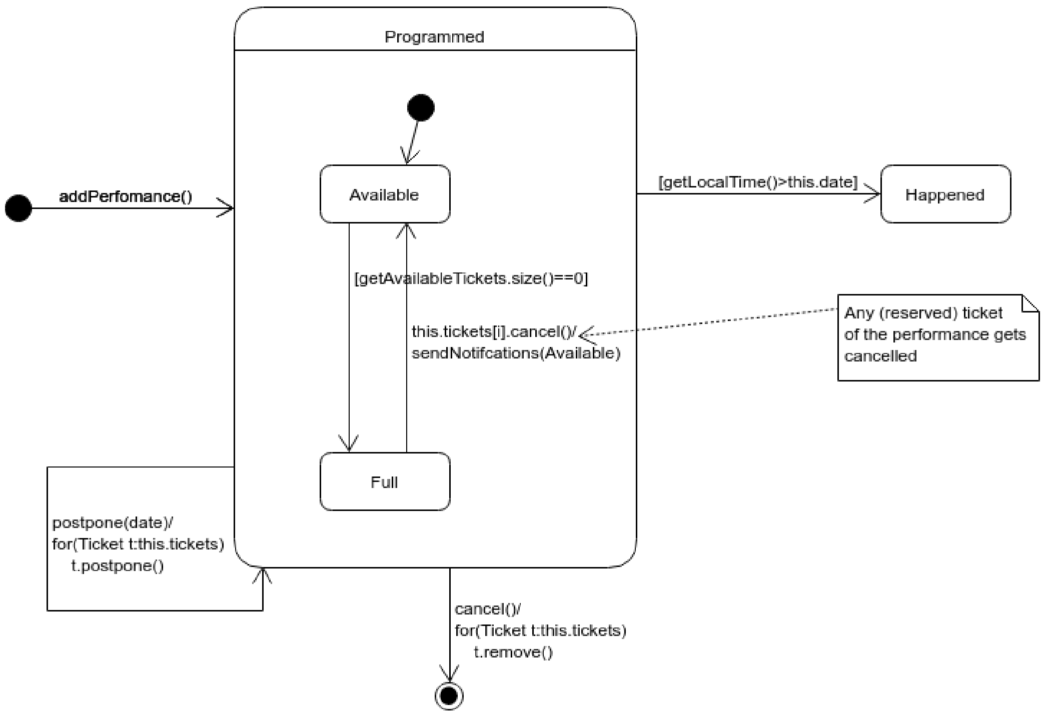

Programmed
addPerfomance()
Available
Full
[getLocalTime()>this.date]
Happened
[getAvailableTickets.size()==0]
this.tickets[i].cancel()/
sendNotifcations(Available)
Any (reserved) ticket of the performance gets cancelled
postpone(date)/
for(Ticket t:this.tickets)
t.postpone()
cancel()/
for(Ticket t:this.tickets)
t.remove()

Figure 2: State diagram: `Performance`

We have defined 2 general states: programmed and happened. In addition, an object of type performance is eliminated when the performance is cancelled (and, as a result, clients who had purchased/reserved a ticket get the appropiate notification-handled by the method `Ticket.remove()`).

- **Programmed**: This is a composite state that shows that the performanced has not taken place yet. The transition to `Happened` is the same for the two atomic states inside, and implies that the scheduled date of the performance already belongs to the past. On the other hand, programmed events can be postponed, which does not result in a change of state; the method `Ticket.postpone()` is invoked over every ticket of the performance, so each client receives a notification.
  - **Available**: This is the initial state of any performance that has been created (as the diagram shows). The transition to `Full` only happens when the guard "no empty spots" is met (which in our design of the class performance, it can be translated into the condition that the list of available tickets is empty).
  - **Full**: It is the opposite state of *Available*, and its transition implies calling the method *sendNotification(Available)* (the argument in this method specifies the type of notification that must be sent and to whom-clients in the waiting list).
- **Happened**: This state is needed so it makes sense to search for "upcoming performances" (the complementary set), as well as being able to retrieve the stats of past performances.

2.2 State diagram of class: `Ticket`

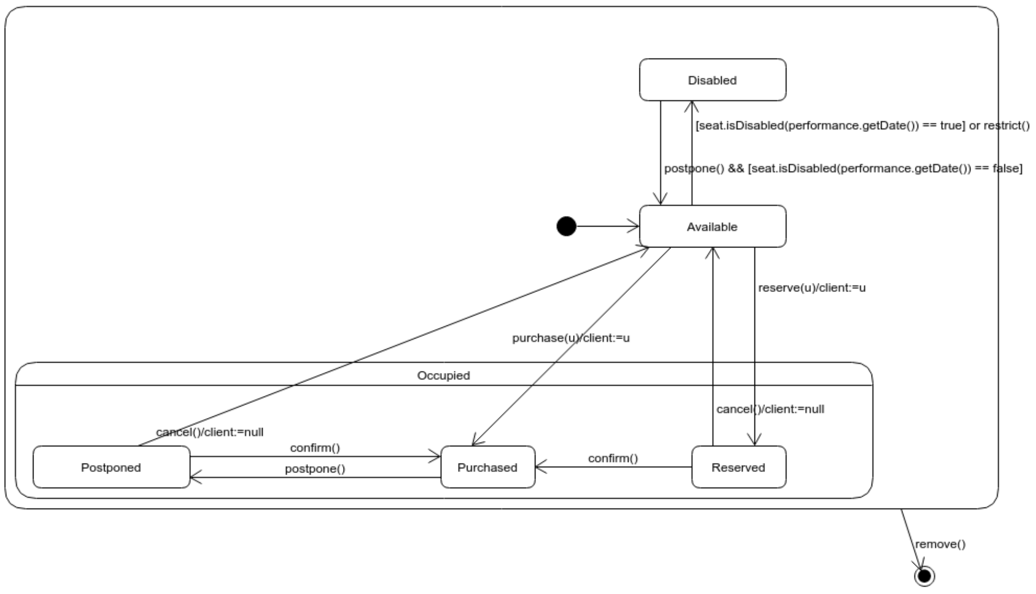


Figure 3: State diagram: `Ticket`

This class has 5 possible states:

- **Available**: The default state, when a performance is created all tickets are available (unless they correspond to a seat that is disabled). If a ticket is cancelled (from Reserved or Postponed), it becomes available.
- **Disabled**: This is the state of the disabled seats. Also, if the admin has applied a restriction to an event, the restricted seats (calculated automatically) will also be in this state.
- **Occupied**: the ticket is associated with a user (composite state):
  - **Reserved**: When the ticket has been reserved by a user.
  - **Purchased**: When the ticket has been bought by a user, or a reservation (or postponed ticket) is confirmed.
  - **Postponed**: This state is reached when the performance is postponed and the ticket was purchased. In this state, the client can confirm the ticket or cancel it (getting their money back). If the client does none of this things, the ticket counts as Purchased when the expiration time is reached.

Note on `postpone()`: This method appears in two of the transitions of the diagram (3). It means that, when the performance is postponed, it will invoke this method over all of its tickets and modify their state if necessary: disabled tickets could become available if they are not disabled in the new date, and purchased tickets become *postponed*. If necessary, notificatios to the clients will also be sent.

`remove()` is a method that is invoked over all of the tickets when the performace is cancelled, after this method, the ticket will be removed. Depending on its state, this method will send notifications to the client, or give the money back.

3 Sequence diagrams

3.1 Sequence diagram: Create area (configure areas)

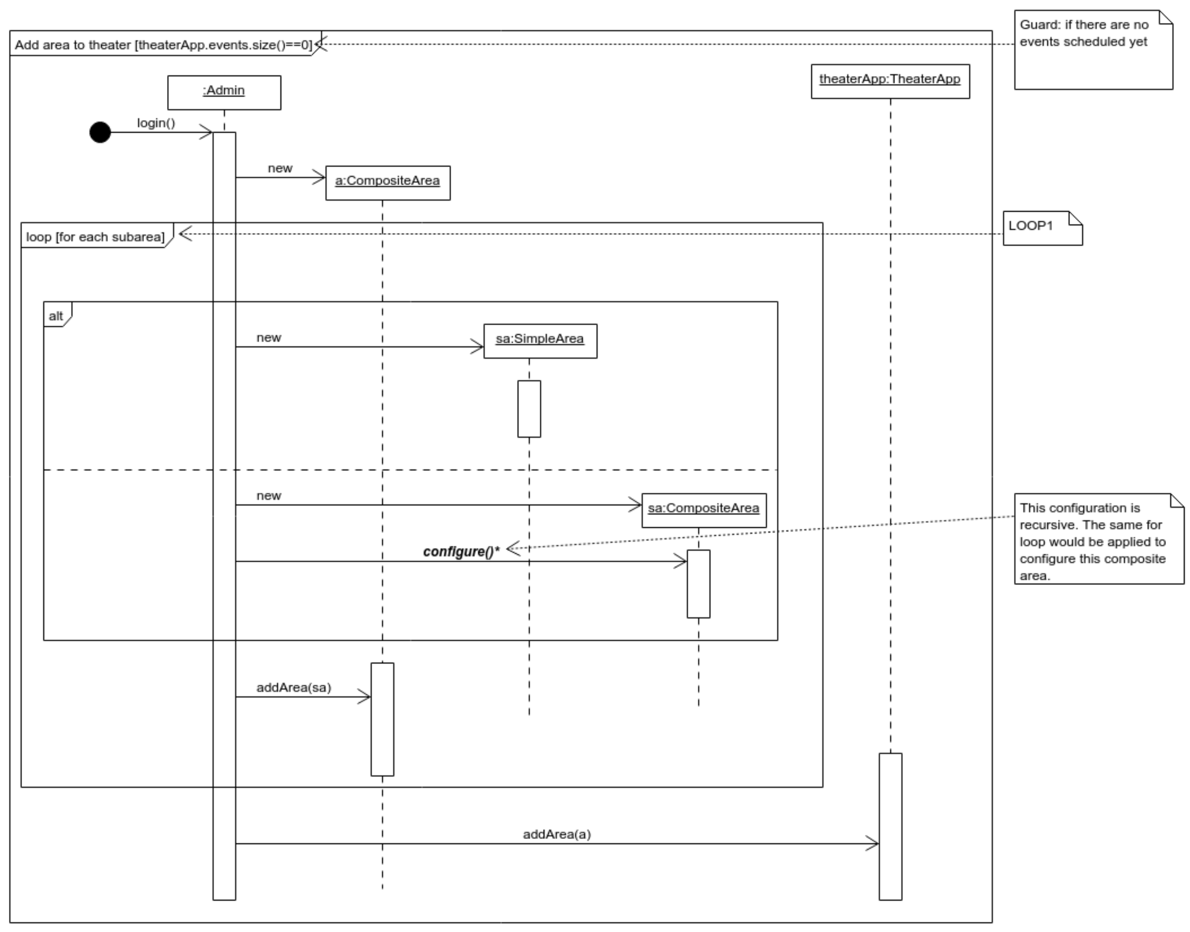

Add area to theater [theaterApp.events.size()==0]
Guard: if there are no events scheduled yet
:Admin
theaterApp:TheaterApp
login()
new
a:CompositeArea
loop [for each subarea]
LOOP1
alt
new
sa:SimpleArea
new
sa:CompositeArea
configure()*
This configuration is recursive. The same for loop would be applied to configure this composite area.
addArea(sa)
addArea(a)

Figure 4: State diagram: `Ticket`

The administrator, after logging in, can create a composite area inside the theater. Then there is a loop (the configuration of that area) where all of the subareas are added. The subareas can be of two types (alternate path): composite or simple. The simple subareas are just created. The composite subareas have to be configured with a loop (the configuration of the area) just like `LOOP1` in Figure 4. Each created subarea has to be added to the composite area.

Finally the composite area that has been created is added to the areas of the Theater.

Of course, this action can only be performed when there are no events in the system.

3.2 Sequence diagram: Purchase/Reserve tickets

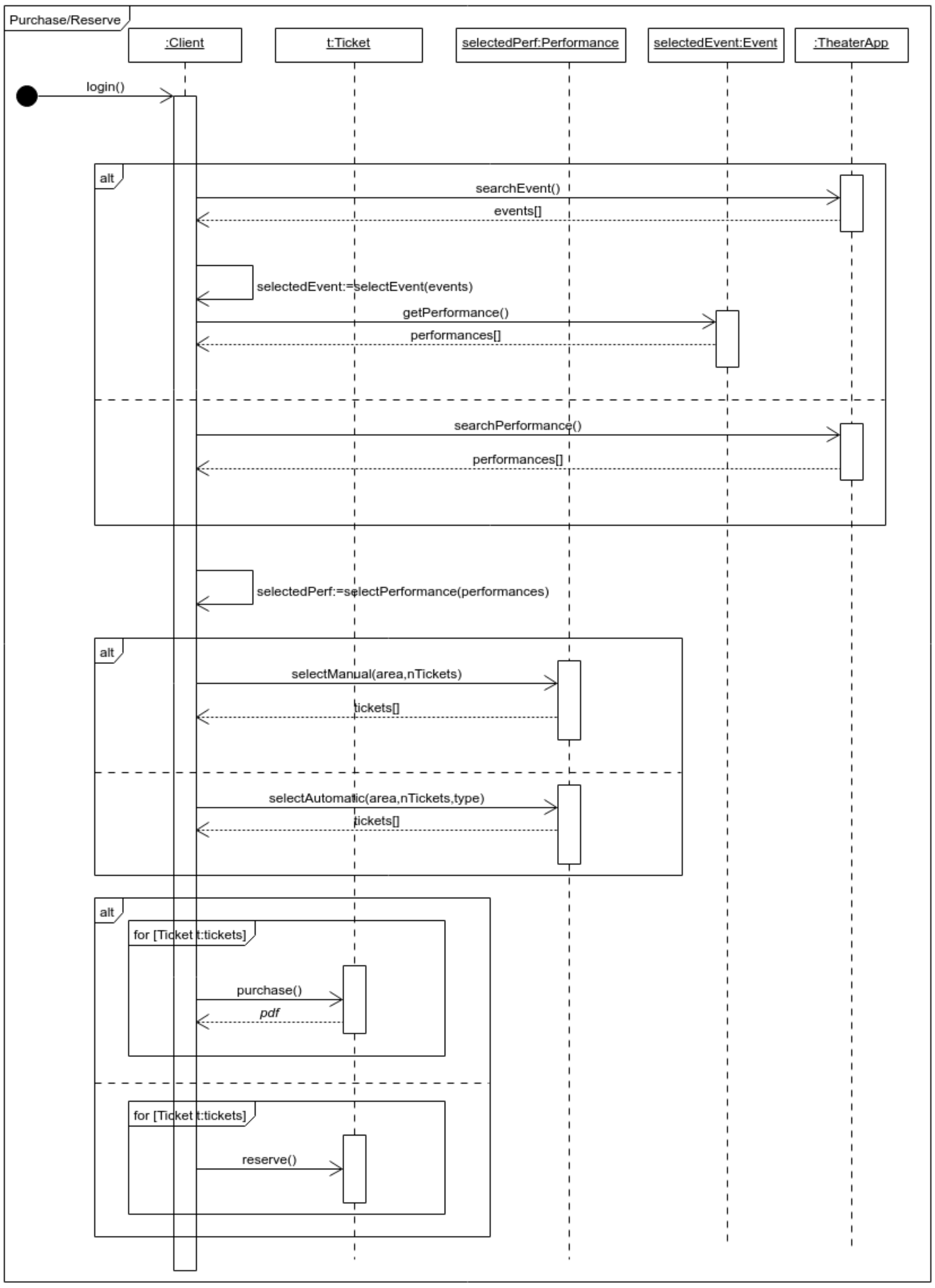
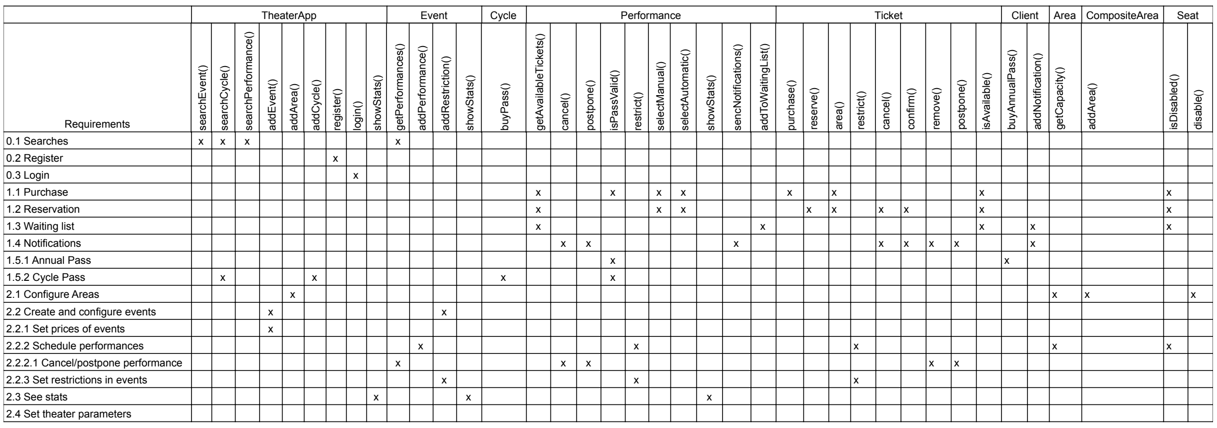


Figure 5: Sequence diagram: Purchase/Reserve tickets

This action (purchase or reserve) can only be made by Clients. Firstly, the client selects a performance. There are two ways of selecting a performance: by searching the event and then selecting the performance, or by directly searching performances.

In second place, the client has to select the tickets automatically or manually. Finally, the client selects whether to buy the tickets or reserve them. If the tickets are purchased, a pdf is generated with an authentication code.

4 Requirements traceability matrix



	TheaterApp									Event				Cycle	Performance										Ticket									Client		Area	CompositeArea	Seat	
Requirements	searchEvent()	searchCycle()	searchPerformance()	addEvent()	addArea()	addCycle()	register()	login()	showStats()	getPerformances()	addPerformance()	addRestriction()	showStats()	buyPass()	getAvailableTickets()	cancel()	postpone()	isPassValid()	restrict()	selectManual()	selectAutomatic()	showStats()	sencNotifications()	addToWaitingList()	purchase()	reserve()	area()	restrict()	cancel()	confirm()	remove()	postpone()	isAvailable()	buyAnnualPass()	addNotification()	getCapacity()	addArea()	isDisabled()	disable()
0.1 Searches	x	x	x							x																													
0.2 Register							x																																
0.3 Login								x																															
1.1 Purchase															x			x		x	x				x		x						x					x	
1.2 Reservation															x					x	x					x	x		x	x			x					x	
1.3 Waiting list															x									x									x		x			x	
1.4 Notifications																x	x						x						x	x	x	x			x				
1.5.1 Annual Pass																		x																x					
1.5.2 Cycle Pass		x				x								x				x																					
2.1 Configure Areas					x																															x	x		x
2.2 Create and configure events				x								x																											
2.2.1 Set prices of events				x																																			
2.2.2 Schedule performances											x								x									x								x		x	
2.2.2.1 Cancel/postpone performance										x						x	x														x	x							
2.2.3 Set restrictions in events												x							x									x											
2.3 See stats									x				x									x																	
2.4 Set theater parameters																																							

Figure 6: Requirements traceability matrix

The matrix above shows how the different funcionalities specified in the RAD (Requirement Analysis Document) are solved by the methods of each class. For simplycity sake, most getters and setters have not been included. Finally, we would like to mention that the funcionality "Set theater parameters" (2.4), is met mainly with getters of the *TheaterApp* class which are not in the matrix.