CO567
Object-Oriented Systems Development
Coursework 1

The BUCKS Centre for the Performing Arts
Design & Implementation

Alex Paulo da Costa

21611431

Jack Standen

21204638

James Harris

21606555

Qasim Maruf

21610356

15th December 2017

Computing & Web Development
Buckinghamshire New University

Summary

A Design **4**

1 Use Case Models . . . **5**
- A The Consumer . . . 5
  - A.1 Register . . . 5
  - A.2 Login . . . 5
  - A.3 Update User Profile . . . 5
  - A.4 View Upcoming Events . . . 6
  - A.5 View Shows by Date . . . 6
  - A.6 Purchase Tickets . . . 6
  - A.7 View Tickets . . . 6
  - A.8 Diagram . . . 7
- B The Venue Manager . . . 8
  - B.1 Add Event . . . 8
  - B.2 Reschedule Event . . . 8
  - B.3 Cancel Event . . . 8
  - B.4 Add Show . . . 8
  - B.5 Reschedule Show . . . 9
  - B.6 Cancel Show . . . 9
  - B.7 Change Maximum-Seats-Per-Customer Value . . . 9
  - B.8 Add Promotion . . . 10
  - B.9 Assign Promotions . . . 10
  - B.10 Change Promotion . . . 10
  - B.11 Delete Promotion . . . 10
  - B.12 Create Agent's Contract . . . 11
  - B.13 Modify Agent's Contract . . . 11
  - B.14 Cancel Agent's Contract . . . 11
  - B.15 Renew Agent's Contract . . . 12
  - B.16 Diagrams . . . 12
- C The Agent . . . 15
  - C.1 Create Customer Profile . . . 15
  - C.2 Buy tickets . . . 15
  - C.3 Update Customer Profiles . . . 15
  - C.4 View Sold Tickets . . . 15
  - C.5 Diagram . . . 16
- D Consolidated Model . . . 17

2 Identifying Classes . . . **18**

3 Data Dictionary . . . **20**
- A Event . . . 20
- B Show . . . 20
- C Seat . . . 20
- D Ticket . . . 21
- E Promotion . . . 21
- F Discount . . . 21
- G User . . . 21
- H Customer . . . 22

I Agent . . . 22

4 Report . . . 23
A Process used . . . 23
B Constraints and Assumptions . . . 23
C Difficulties Encountered . . . 23
D Contribution and discussions . . . 23

B Implementation 24

5 Requirements . . . 25
A Expected Functional . . . 25
B Non Functional . . . 26

6 Design Class Diagram . . . 27

7 Identifying Classes . . . 28

8 Sequence Model . . . 30

9 Detailed Design with Pseudo-code . . . 31
A Common Methods . . . 31
A.1 Getters . . . 31
A.2 Setters . . . 31
B Event . . . 31
B.1 Get Shows . . . 31
C Show . . . 32
C.1 Hold a Seat . . . 32
C.2 Unhold a Seat . . . 32
C.3 Reserve a Seat . . . 32
D User . . . 32
D.1 Change Password . . . 32
D.2 Login . . . 33
D.3 Logout . . . 33
E Customer . . . 33
E.1 View Tickets . . . 33
E.2 View Details . . . 33
E.3 Register . . . 33
F Venue Manager . . . 34
F.1 Reschedule an Event . . . 34
F.2 Cancel an Event . . . 34
F.3 Change a Contract . . . 34
F.4 Cancel a Contract . . . 34
G Agent . . . 35
G.1 Get Tickets . . . 35
G.2 Modify a Customer . . . 35

10 Implementation in Java . . . 36
A User . . . 37
B Agent . . . 37
C Customer . . . 37
D Venue Manager . . . 37
E Event . . . 37
F Show . . . 37
G Seat . . . 37
H Promotion . . . 37
I Discount . . . 37
J Ticket . . . 37

K Ticket System 37
L Enumerations 37
L.1 Statuses 37
L.2 WeekDay 37
L.3 Discount Types 37
L.4 Seat Statuses 37
L.5 User Statuses 37
L.6 User Types 37

C Appendix 38

Section A

Design

Use Case Models

The Consumer

Register

Pre-conditions: The consumer does not have an existing customer profile.

Post-conditions: A new customer profile exists in the system, with a name, email and password and the consumer is logged in.

Purpose: The consumer wants to book tickets for an upcoming show.

Description: The consumer fills in a registration form with their desired name, email and password. Once complete, they press the register button and they will be automatically logged in after their customer profile has been created on the system.

Login

Pre-conditions: The consumer has a customer profile.

Post-conditions: The consumer is logged in.

Purpose: The consumer has previously bought tickets for a show and wants to buy tickets for an upcoming show.

Description: The consumer enters their email and password. The details are verified on the system and if they are valid, the user is logged in. If the credentials are invalid, the consumer is prompted to re-enter their email and password.

Update User Profile

Pre-conditions: The consumer has a customer profile and is logged in.

Post-conditions: The stored customer profile information is modified.

Purpose: The consumer wants to change their email, shipping address or password.

Description: The consumer updates the fields they wish to change (e.g, their shipping address). Once they are happy with the changes, they click the save button and their customer profile will be modified on the system.

View Upcoming Events

Pre-conditions: Zero or more events exists.

Purpose: The consumer wants to see all the upcoming events that they can book tickets for.

Description: The consumer is presented with a list of events. Selecting an event will show the list of shows assigned to the event. If there are no upcoming events, then the list will be empty with a message indicating there are no events on.

View Shows by Date

Pre-conditions: Zero or more shows exist.

Purpose: The consumer wants to see all the shows on over a range of days.

Description: The consumer is presented with a list of shows that are taking place within their specified date range. Selecting a show will allow the consumer to book tickets for the show. If there are no shows taking place in this date range, the list will be empty with a message indicating there are no shows.

Purchase Tickets

Pre-conditions: The consumer is logged in and one or more shows exist.

Post-conditions: A new ticket exists in the system.

Purpose: The consumer wants to buy one or more tickets for a selected show.

Description: The consumer is asked how many tickets they wish to purchase. After selecting the amount, the system displays the best available seat(s) with their price. The consumer also has the option of manually picking their seats via a button. The manual option displays a seating chart with the available seats highlighted. The consumer can then select the needed amount of seats. With both methods, the currently selected seats are reserved and not available to other consumers that are booking tickets for the same show. The reservation is cancelled once the transaction is cancelled or has not been completed after 5 minutes.

Once they are happy with the selected seats, the consumer is shown the total cost of the tickets. Here they are able to add a valid promotion for the show. The consumer can then enter their credit card information and confirm the purchase.

Upon confirmation of the purchase, the ticket(s) are added to the system and displayed again to the user as a receipt.

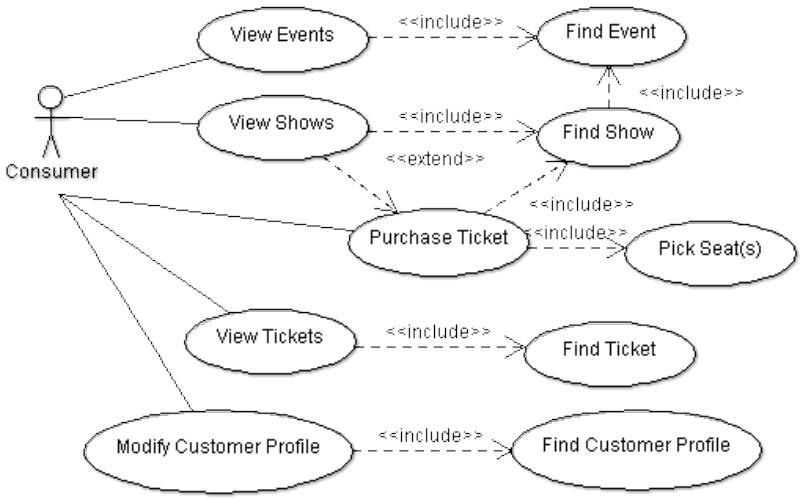
View Tickets

Pre-conditions: The consumer is logged in.

Purpose: The consumer wants to see their past purchases.

Description: The consumer is presented with a list of their tickets, showing the price, date and seat number.

Diagram


View Events
<<include>>
Find Event
<<include>>
View Shows
<<include>>
Find Show
<<extend>>
Consumer
<<include>>
Purchase Ticket
<<include>>
Pick Seat(s)
View Tickets
<<include>>
Find Ticket
Modify Customer Profile
<<include>>
Find Customer Profile

The Venue Manager

Add Event

Pre-conditions: The venue manager is logged in into the system.

Post-conditions: A new event exists in the system, with a start and end date and a name.

Purpose: The venue manager wants to add an event to the ticket selling system in order to allow customers to buy tickets for it and to promote it.

Description: The venue manager fills in the form for the creation of a new event with the name of the event and a description of it, then selects a start and end date from the "calendar menus". When they are satisfied with all the details, they press the save button and the changes are saved in the system.

Reschedule Event

Pre-conditions: The venue manager is logged in into the system and at least one event exists in the system that hasn't been cancelled already

Post-conditions: An existing event has either a new start date, a new end date or both

Purpose: The venue manager wants to change the dates an event runs for in order to allow customers to have the accurate dates the event will be occurring and for them to be able to buy tickets for it

Description: The manager selects the event from a list of events, then a screen with the event's information displayed in a form style appears, allowing the manager to make the changes he needs to make to the dates, then they press the save button and the changes are saved into the system.

Cancel Event

Pre-conditions: The venue manager is logged in into the system and at least one event exists in the system.

Post-conditions: A previously existing event will be displayed as cancelled in the system and no more tickets will be available for sale for that event (the event won't be deleted from the system though, in case customers have already seen the event in the system and/or bought tickets for it)

Purpose: The venue manager wants to cancel an event that was scheduled to occur for any reason in order to inform the customers that the event is no longer occurring.

Description: The venue manager selects the event they wish to cancel from a list and then the system prompts them with a message to confirm they wish to cancel that event. If the user selects yes, the changes are saved into the system and the event will then appear as cancelled to the users.

Add Show

Pre-conditions: The venue manager is logged in into the system and at least one event exists in the system.

Post-conditions: A new showing for an event exists in the system, with a date and start time and a maximum-seats-per-customer value.

Purpose: The venue manager wants to add a showing to an event that is planned to inform the customers and agents of the show occurrence and allow for tickets to be sold.

Description: The manager selects the event for which they want to add a show from a list, then enters the details of the show in a form (date the show is happening and start and end time as well as the maximum-seats-per-customer value) and then presses save which will save the changes in the system and inform the user of the changes.

Reschedule Show

Pre-conditions: The venue manager is logged in into the system and at least one event with a least one show exists in the system.

Post-conditions: Either the date or the time or both of a show will be modified in the system.

Purpose: The venue manager wants to reschedule a showing of an event for whatever reason, in order to inform customers and sell tickets for the show with accurate time and date information.

Description: The manager selects the event for which they want to reschedule the show from a list and then a list of the shows for that event appear, from which the user selects the show that they want to reschedule. After selecting the show, a form displaying the current date and times of the show is shown in the screen, and the user can then select the new date and/or times. When they are satisfied with the new details for the show, they press the save button. The system will then save the changes and notify the users that the changes have been saved.

Cancel Show

Pre-conditions:The venue manager is logged in the system and at least one event and one show that hasn't been cancelled for that event exist in the system.

Post-conditions: The show will be marked as cancelled in the system and no more tickets will be available for sale for that show (the show won't be deleted from the system though, in order to allow customers to see that the event has been cancelled).

Purpose: The venue manager wishes to cancel a show for whatever reason is order to inform the customers that the show is no longer occurring and block the sale of tickets for that showing of the event.

Description: The user selects the event that the showing relates to from a list, then a list appears with the possible shows. The user selects the show they wish to cancel. After that, the system displays a confirmation message to make sure that it is that show that the user wants to cancel. After the user confirms that it is that show that they want to cancel, the systems saves the changes and notifies the user that the changes have been saved.

Change Maximum-Seats-Per-Customer Value

Pre-conditions: The venue manager is logged into the system and at least one event and one show exist in the system.

Post-conditions: The show will have a different maximum-seats-per-customer value.

Purpose: The venue manager wants to change the amount of seats a customer can buy of a single show in order to adapt the number of tickets that can be sold to a single customer according to the current demand for that show.

Description: The user selects the event from a list and the system then displays a list with the shows for that event. After selecting the show the user wants to change, the system displays a form that user uses to change the maximum-seats-per-customer value. After changing the value, the user presses the save button. The system saves the changes in the system and then notifies the user that the changes have been saved.

Add Promotion

Pre-conditions: The venue manager is logged into the system

Post-conditions: A promotion (pricing structure) is created in the system, with a name, price structure for different types of tickets and applicable discounts (volume discounts and others)

Purpose: The venue manager wants to set a new promotion for an upcoming event, or for a particular time of the day, with specific prices and applicable discounts.

Description: The system displays a form with the information that needs to be filled in by the user to create the new promotion(name, prices for children, students, adults and seniors, as well as a section to add different types of discounts that is optional). The user fills in the form and presses the save button. The system saves the changes and notifies the user that the campaign has been added when all the changes have been saved.

Assign Promotions

Pre-conditions: The venue manager is logged into the system, and at least one event with one show and one promotion exist in the system.

Post-conditions: A promotion is assigned to some (if not all) seats of the selected show.

Purpose: The venue manager wishes to assign a promotion to the seats of a show so that the correct amount of money is charged to customers when they want to buy tickets for that show and in order to be able to sell tickets for the show.

Description: The user selects the event to which the show belongs from a list, then the system displays a list with the shows for that event. After the user selects the show, the system displays the seats of the room where the show is occurring. The user selects the seats to which the user wants to add a promotion, then the system displays the available promotions. The user selects the promotion to add to those seats. In that screen, the user can also select other ranges of seats and attribute a promotion to each of them. When satisfied with the changes, the user presses the save button. The system saves the changes and notifies that the promotions have been assigned when all the changes are stored.

Change Promotion

Pre-conditions: The venue manager is logged into the system, and at least one promotion exists in the system.

Post-conditions: The details of a promotion have been changed (either the name, or the pricing structure or available discounts for that promotion).

Purpose: The venue manager wants to change a promotion to update its prices, or modify discounts available for it.

Description: The user selects the promotion they want to change from a list. The system displays then a form containing the current information for the promotion and that also allows the user to change the information they want to change (except the name of the promotion). After they made the changes, they press the save button. The system saves the changes and notifies the user of that when all changes are stored.

Delete Promotion

Pre-conditions: The venue manager is logged into the system and at least one promotion exists on the system.

Post-conditions: The selected promotion is deleted from the system.

Purpose: The venue manager wishes to delete an irrelevant promotion from the system.

Description: The user selects the promotion they wish to delete from a list. The system displays a confirmation message to make sure that is the promotion that the user wishes to delete, and if the user confirms then the system saves the changes and notifies the user that the changes have been saved.

Create Agent's Contract

Pre-conditions: The venue manager is logged into the system.

Post-conditions: A new agent's contract is created.

Purpose: The venue manager wishes to create a new agent's contract to allow an agent to use the OTS and sell tickets with their own seats assigned and the right commission.

Description: The user fills in the form with all the details about the agent's contract (name, email, seats to be assigned, commission that they earn, start date of the contract and duration of the contract). The user then clicks on the save button and the system saves the changes. When all the changes are saved, the system will notify the users that the contract has been added successfully.

Modify Agent's Contract

Pre-conditions: The venue manager is logged into the system and there is at least one agent's contract in the system.

Post-conditions: The agent's contract has been changed and has now new values (assigned seats, commission, duration of the contract and/or start date if the contract hasn't started yet).

Purpose: The venue manager wishes to change an agent's contract for whatever reason so that the agent has access to the platform throughout the duration of their actual contract and no longer and to have a range of seats available for them to sell that is accurate to their sales, as well as to take the right commission.

Description: The user selects the name of the agent whose contract's details need to be changed. The system sends back a form with the details of the agent's contract. The user changes the details that need to be changed and, when finished, presses the save button. The system saves the changes and notifies the user when the changes have been applied.

Cancel Agent's Contract

Pre-conditions: The venue manager is logged in into the system and at least one agent contract is in the system.

Post-conditions: The agent's contract is terminated, the seats they had reserved are available to all non-agent customers, and the details of the contract are deleted from the system.

Purpose: The venue manager wishes to terminate the Contract the BCPA has with an agent for whatever reason in order to release the seats back to sale for general customers and to end their access to the system.

Description: The user selects a agent from a list, and a confirmation message appears to make sure that the user has chosen the right agent. When confirmed, the system removes the agent contract from the system and notifies the user when all the changes have been processed.

Renew Agent's Contract

Pre-conditions: The venue manager is logged into the system and there is at least one agent's contract in the system.

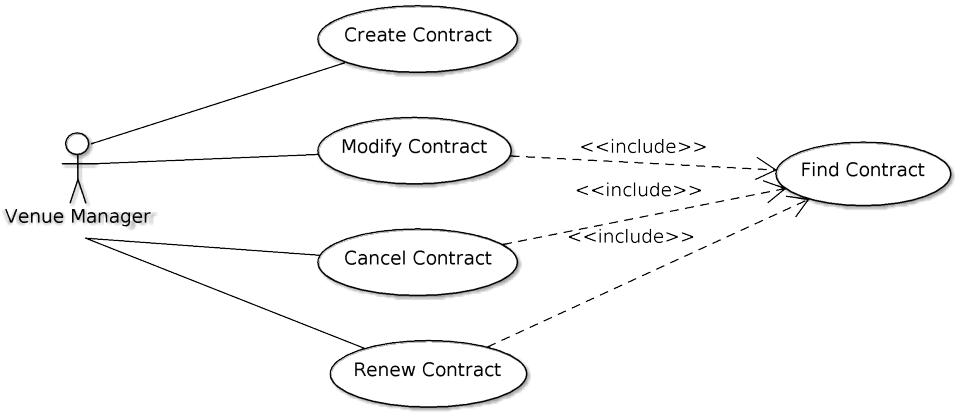
Post-conditions: The contract's end date is extended to the end of the current contract plus another duration of the contract.

Purpose: The venue manager wishes to renew a contract with an agent in order to allow them to sell tickets on the OTS for longer.

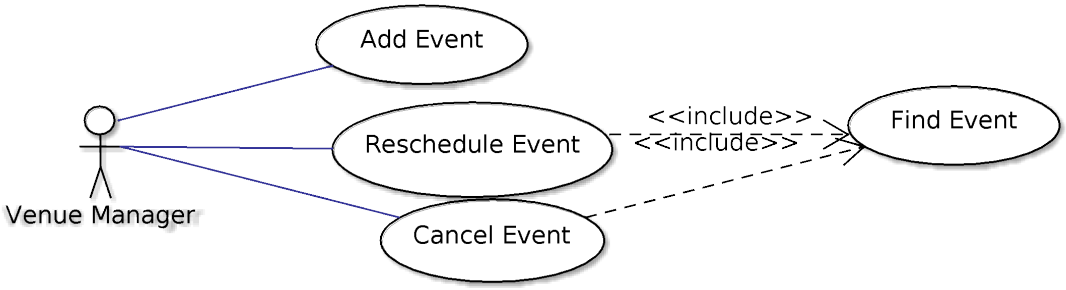
Description: The user selects an agent from a list, and a confirmation message appears to make sure that the user has chosen the right agent. When confirmed, the system will then change the end date of the contract in the system to that of the end of the current contract plus the duration of the contract and notify the user when the changes have been processed.

Diagrams

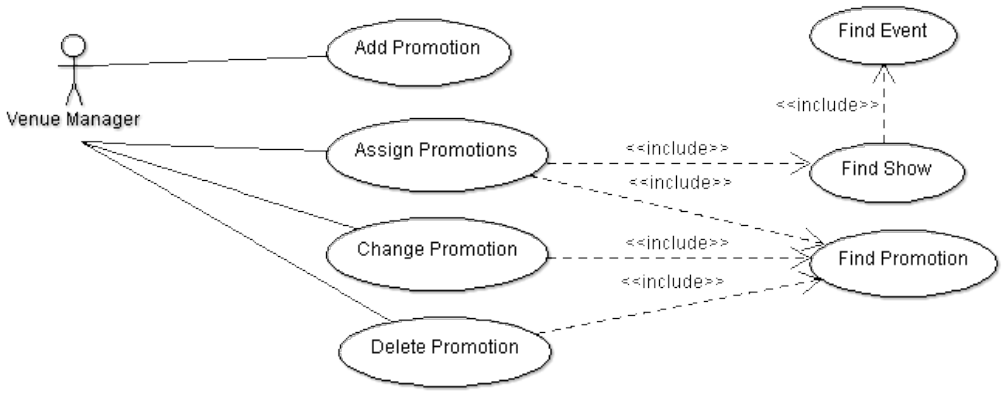
Contract-related Use Cases



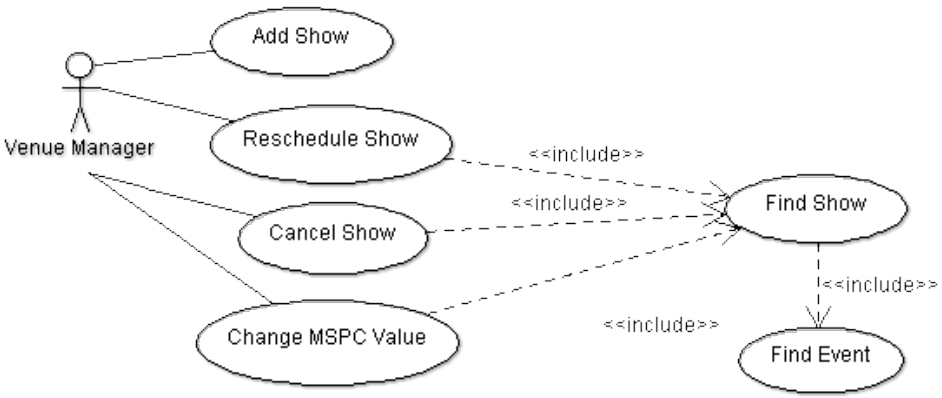
Event-related Use Cases



Promotion-related Use Cases



Show-related Use Cases



The Agent

Create Customer Profile

Pre-conditions: The Agent must be logged into the system.

Post-conditions: Customer successfully created a profile.

Purpose: The agent wants to be able to add customers to the system in order to buy tickets for them.

Description: The user fills in the form with the customer details and presses the save button. Once all the details are saved, the system notifies the user that all the changes have been saved.

Buy tickets

Pre-conditions: An agent must be logged in into the system and at least one event and one show exist on the system, as well as one customer that is managed by the agent.

Post-conditions: Tickets have been bought by an agent on behalf of a customer at the right price with the agent's commission.

Purpose: The agent wants to be able to sell tickets for his customers in order to gain money and commission, and to have the tickets reserved for the right person

Description: The user selects the customer he wants to buy tickets for and presses OK, the system then prompts the user to select the event they want to buy tickets for. The user selects the event and then presses OK, and then a list of the available shows appears with the number of seats available for the agent to sell for this show. The agent then selects the show and presses OK. The user is then prompted to choose the number of tickets that he wants to buy, and the user selects the number and presses OK. The system then displays the seats that are available for the agent to reserve. The agent selects the seats to be reserved according to the number of tickets that they are buying and then select OK. The total price for the tickets then appears in the system, explaining which values are for the tickets and which are for the agent's commission, then the user presses the Pay button to complete the purchase.

Update Customer Profiles

Pre-conditions: An agent is logged into the system and at least one customer profile managed by this manager exists in the system.

Post-conditions: A profile managed by an agent is updated

Purpose: An agent wants to make changes to the information held in the profile of a customer that they manage (name, address, payment information, etc.)

Description: The system provides to the user a list of the users that they manage and the user selects the user to which they want to change the information. A form then appears holding the current information of the customer. The user changes the information necessary and when they are happy with the changes they press the save button. When all the changes have been saved, the system notifies the user that the changes have been successfully been applied.

View Sold Tickets

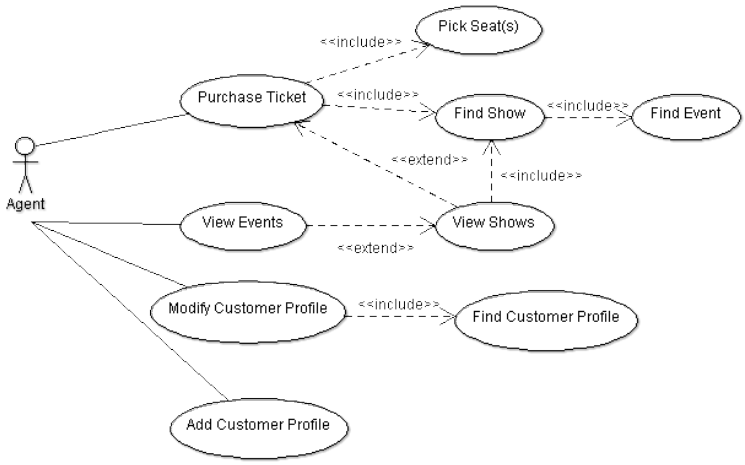
Pre-conditions: An agent is logged into the system.

Post-conditions: A list with the tickets sold and the total number of tickets that the agent has sold either for a show or for a range of dates are displayed to the user.

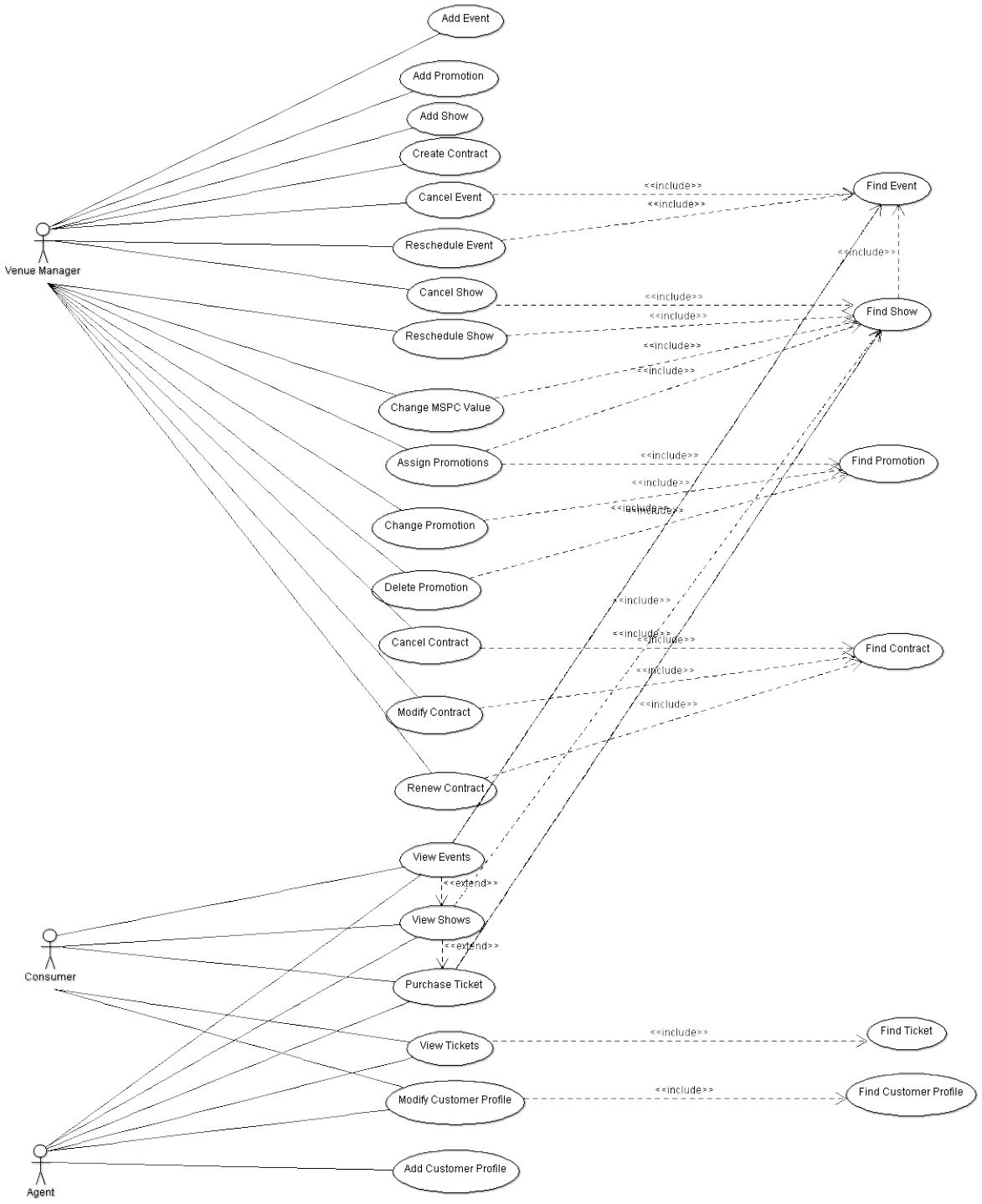
Purpose: An agent wants to see how many tickets they have sold in the platform, either for a particular show or for all shows in a date range.

Description: The system ask the user if they want to see the tickets that they have sold for a show or for a specific date range. The user selects one of the two options. If they have selected a show, the system will display a list of events, from which the user must choose an event, then a list of shows for that event. The user then selects the show and the list of sold tickets and the total number of tickets is shown to the user. In case the user wants to see the tickets that they have sold for a date range, the user selects the begin date of the search and then the end date of the search and the system will then display all the tickets sold and the total number of tickets.

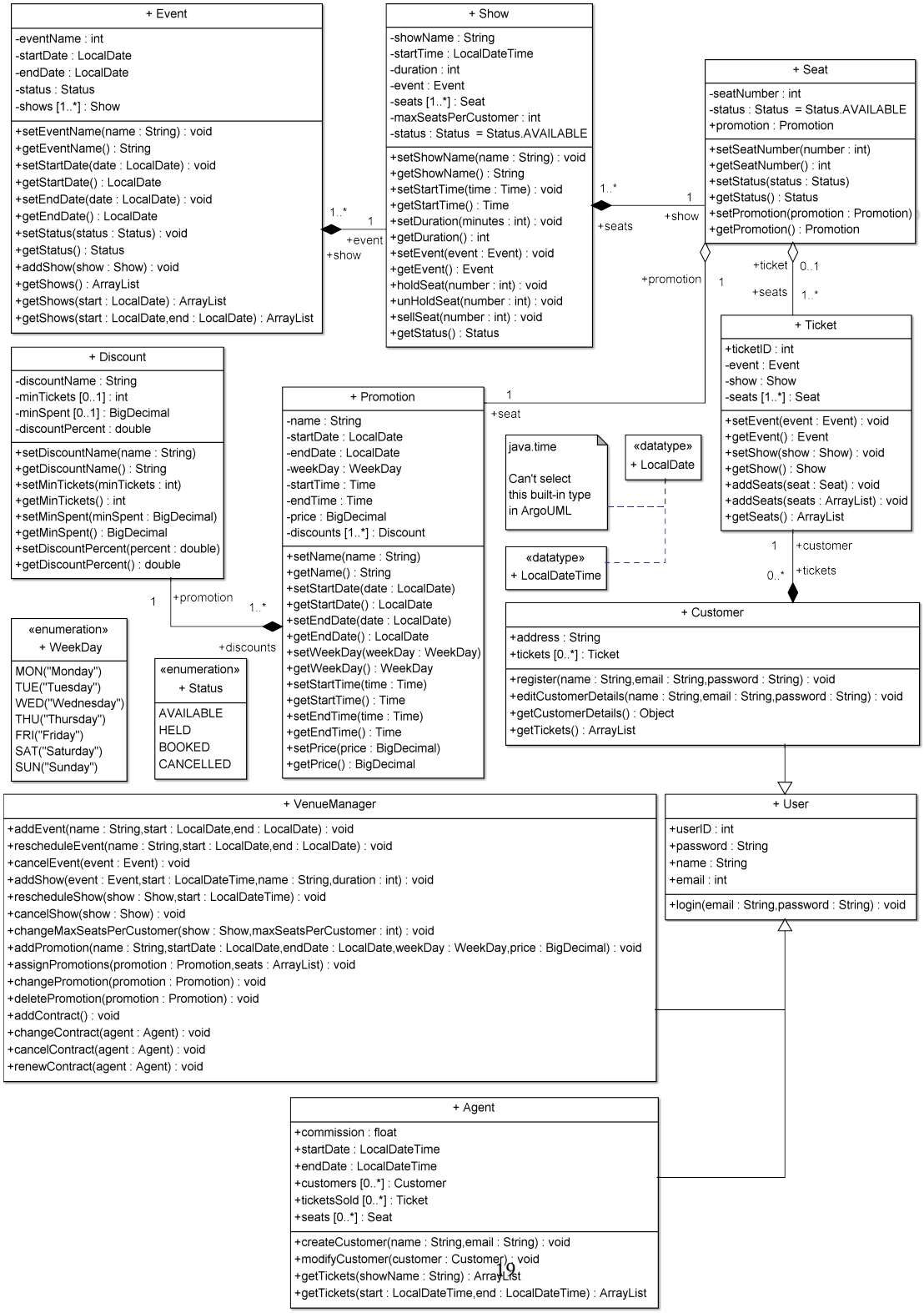
Diagram



Consolidated Model



Identifying Classes



Data Dictionary

Event

Attribute	Type	Description	Example
eventName	String	Name of the Event	The Wizard of Oz
startDate	LocalDate	Date the Event starts	2017-04-09
endDate	LocalDate	Date the Event ends	2017-04-11
status	Status	Status of the event (e.g available/sold out)	AVAILABLE
shows	ArrayList<Show>	Shows part of the event	

Show

Attribute	Type	Description	Example
showName	String	Name of the Show	The Wizard of Oz (Thu)
startTime	LocalDateTime	Date the Show starts	2017-04-09T19:30:00
duration	Integer	Duration of the show in minutes	125
event	Event	Event the Show is part of	
seats	ArrayList<Seat>	Seats assigned to the show	
maxSeatsPerCustomer	Integer	Number of seats a customer can buy at once	6
status	Status	Status of show, e.g available/sold out	CANCELLED

Seat

Attribute	Type	Description	Example
seatNumber	Integer	Unique seat number for the show	29
status	Status	Status of the seat, e.g available/booked/held	HELD
promotion	Promotion	Promotion assigned to the seat by the manager	

Ticket

Attribute	Type	Description	Example
ticketID	Integer	Unique ID of the Ticket	1
event	Event	Event the ticket was purchased for	
show	Show	Show the ticket was purchased for	
seats	ArrayList<Seat>	Seats purchased	

Promotion

Attribute	Type	Description	Example
name	String	Name of the promotion	Weekend Special
startDate	LocalDate	Date the promotion is valid from	2017-12-23
endDate	LocalDate	Date the promotion is valid until	2017-12-24
weekDay	WeekDay	Day the week the promotion is valid	TUE
startTime	Time	Time the promotion is valid from	15:00:00
endTime	Time	Time the promotion is valid until	18:00:00
price	BigDecimal	Price of the ticket	4.30
discounts	ArrayList<Discount>	Discounts associated to the promotion	

Discount

Attribute	Type	Description	Example
discountName	String	Name of the discount	25% off 2 or more tickets
minTickets	Integer	Minimum number of tickets required	2
minSpent	BigDecimal	Minimum total cost required	22.50
discountPercent	Double	Percent off of the cost	25

User

Attribute	Type	Description	Example
userID	Integer	Unique ID of each user in the system	1
password	String	Password for the user to login	hunter2
name	String	First and last name of the user	John Doe
email	String	Email for the user to login	john.doe@gmail.com

Customer

Attribute	Type	Description	Example
address	String	Address to post tickets to	60 Brook St, High Wycombe, HP11 2EQ
tickets	ArrayList<Ticket>	Tickets the customer has purchased	

Agent

Attribute	Type	Description	Example
commission	Float		
startDate	LocalDateTime	Start of the agent's contract	2017-01-28T09:00:00
endDate	LocalDateTime	End of the agent's contract	2017-09-28T17:00:00
customers	ArrayList<Customer>	Customers managed by the agent	
ticketsSold	ArrayList<Ticket>	Tickets sold by the agent	
seats	ArrayList<Seat>	Seats managed by the agent	

Report

Process used

We analysed the study case first and tried to obtain the use cases first so that we could know what exactly the application would have to do. After that, we started to think about how we could implement that functionality and that's how we started to do the class diagrams, since the implementation thinking process helped us identify necessary classes, operations and attributes.

Constraints and Assumptions

In order to produce the model, we did some assumptions:

- The payments would always be accepted
- An event represents a performance that runs in the theatre for a certain period of time (for example, a musical that is being played at the theatre for two months)
- A show is an individual performance of an event (continuing in the musical example, a show would be a single performance of the musical occurring in an evening)

Difficulties Encountered

The major encountered difficulties we had was with identifying how the system would work and how all the classes would be linked to each other, as well as what each one would do (responsibility assignment). In order to overcome it, we decided to modulate the operations as much as possible and get every interested party to do a little bit of work towards its completion.

Contribution and discussions

The log of contributions and discussions from the team can be found in the appendix at the end of the report.

Section B

Implementation

Requirements

Expected Functional

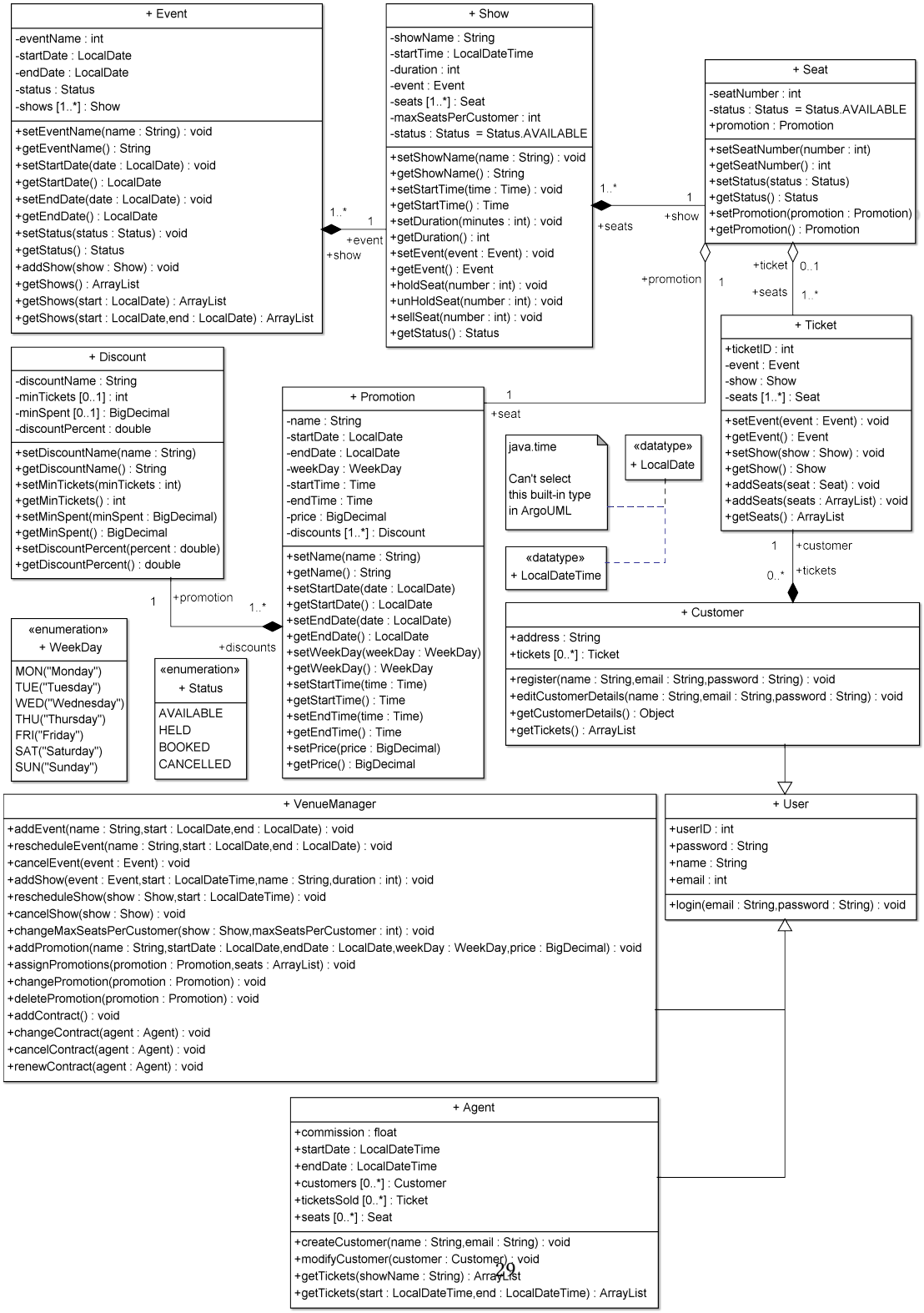
- The system should allow consumers to register and login
- The system should allow consumers to see the upcoming events and shows
- The system should allow consumers to purchase tickets for shows
- The system should allow for specific seats to be reserved when purchasing tickets
- The system should allow the users (agents and consumers) to see the tickets they have purchased
- The system should allow the venue manager to add events and shows
- The system should allow the venue manager to change the events and shows (reschedule and cancel)
- The system should allow the venue manager to change the maximum-seats-per-customer value
- The system should allow the venue manager to add promotions
- The system should allow the venue manager to assign promotions to seats of specific shows
- The system should allow the venue manager to delete promotions
- The system should allow the venue manager to add agent's contracts
- The system should allow the venue manager to cancel an agent's contract
- The system should allow the venue manager to renew an agent's contract
- The system should allow an agent to see the number of tickets they have sold

Non Functional

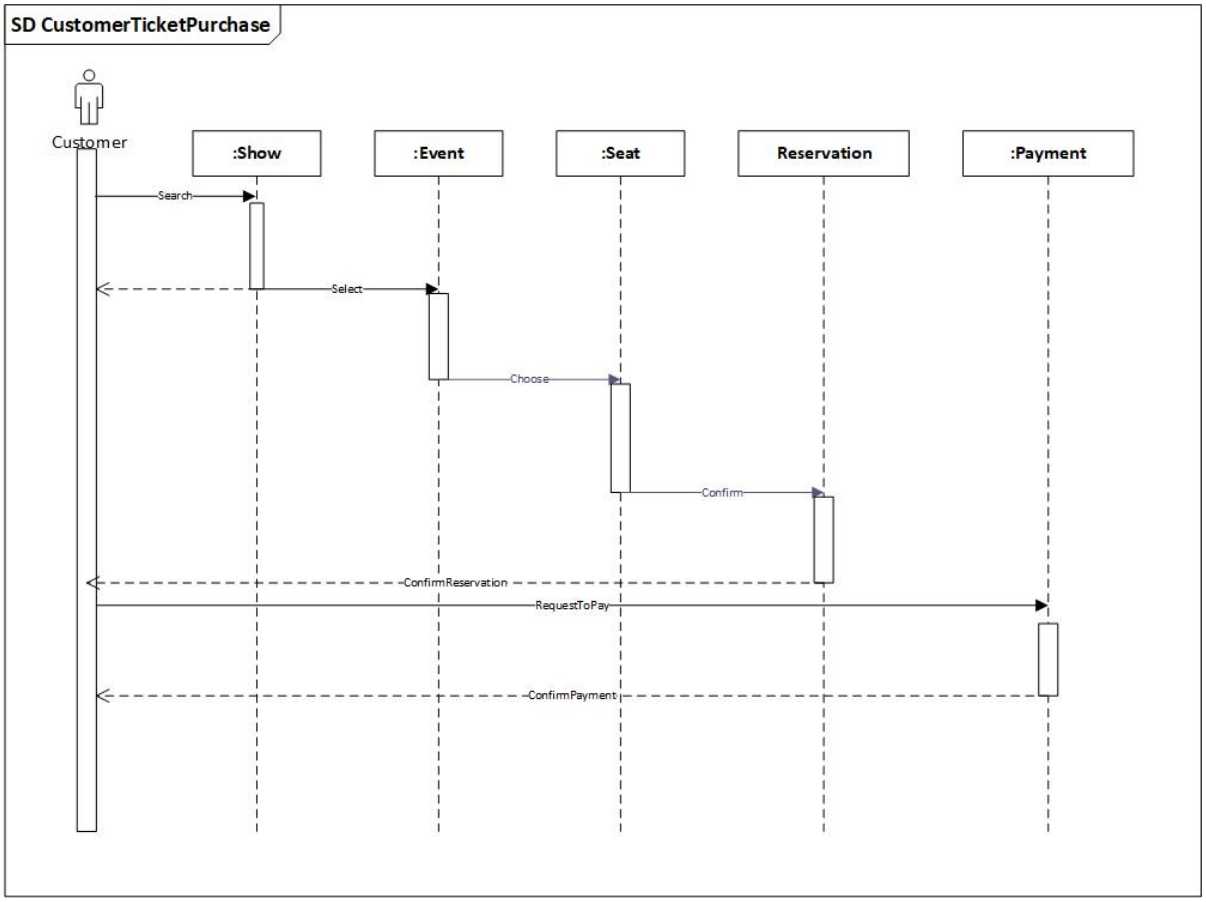
- The login operation shouldn't take more than 2 seconds to be completed.
- The search for information shouldn't take more than 5 seconds
- New events should be advertised and appear automatically on the platform only two weeks before the start date
- The login for an agent should start working automatically on the contract start date.
- The login for an agent should stop working automatically after the contract end date.
- After any changes to the data of the system, it should automatically be updated to the system and changes should be noticed in real-time.

Design Class Diagram

Identifying Classes



Sequence Model



Detailed Design with Pseudo-code

Common Methods

Getters

Methods to get an attributes value will all follow the following structure. For example, to get an event's name from the Event class:

```
RETURN event name
```

Setters

Methods to modify an attribute's value(s) will all have the following structure. For example, to set a new name for an event from the Event class:

```
OBTAIN the new event name
SET event name to the new event name
```

Event

Get Shows

Method called with the $start$ and end parameters:

```
OBTAIN the start date
OBTAIN the end date

FOR each show in the event
  IF the show's startDate is after the start date AND the show's ndDate is before the end date   THEN
    DISPLAY the show's details
  END IF
END FOR
```

Method called with the $showDate$ parameter:

```
OBTAIN the showDate

FOR each show in the event
```

```
  IF the show's startDate equals the showDate THEN
      DISPLAY the show's details
  END IF
END FOR
```

Show

Hold a Seat

```
OBTAIN seat number
GET seat from seats where seat number matches

IF seat status is available THEN
  SET seat status to held
  DISPLAY held seat number
ELSE
  DISPLAY seat is not available
END IF
```

Unhold a Seat

```
OBTAIN seat number
GET seat from seats where seat number matches

SET seat status to available
```

Reserve a Seat

```
OBTAIN seat number
GET seat from seats where seat number matches

IF seat status is not reserved THEN
  SET seat status to reserved
ELSE
  DISPLAY seat is already reserved
END IF
```

User

Change Password

```
OBTAIN new password from the user
OBTAIN current password from the user

IF the current password matches the existing password THEN
    IF the new password is not null THEN
        SET password to the new password
    END IF
END IF
```

Login

```
OBTAIN email from the user
OBTAIN password from the user

IF status is loggedOut THEN
    IF password matches stored password THEN
        SET status to loggedIn
        DISPLAY greeting message
    ELSE
        DISPLAY wrong password message
    END IF
ELSE
    DISPLAY user is already logged in message
```

Logout

```
SET status to loggedOut
DISPLAY logged out message
```

Customer

View Tickets

```
FOR each ticket in tickets
    DISPLAY ticket details
END FOR
```

View Details

```
DISPLAY customer name
DISPLAY customer email
DISPLAY customer address
```

Register

```
OBTAIN name
OBTAIN password
OBTAIN email
GET last user ID

ADD a new customer with last user ID, name, password and email
INCREMENT last user ID by 1
```

Venue Manager

Reschedule an Event

```
OBTAIN event
OBTAIN new start date
OBTAIN new end date

IF new start date exists THEN
  SET event start date to new start date
END IF

IF new end date exists THEN
  SET event end date to new end date
END IF
```

Cancel an Event

```
OBTAIN the event

SET event status to cancelled
```

Change a Contract

```
OBTAIN agent
OBTAIN new contract start date
OBTAIN new contract end date
OBTAIN new commission

IF new contract start date exists THEN
  SET agent start date to new contract start date
END IF

IF new contract end date exists THEN
  SET agent end date to new contract end date
END IF

IF new commision exists THEN
  SET agent commission to new comission
END IF
```

Cancel a Contract

```
OBTAIN agent

SET agent end date to current date and time
```

Agent

Get Tickets

Get tickets by event name

```
OBTAIN showName

FOR each ticket in tickets
  SET name to ticket show's name
  IF name matches showName THEN
    DISPLAY ticket details
  END IF
END FOR
```

Get tickets within a date range

```
OBTAIN startDate
OBTAIN endDate

FOR each ticket in tickets
  SET showTime to the ticket show's start time
  IF showTime is after the startDate AND showTime is before the endDate THEN
    DISPLAY ticket details
  END IF
END FOR
```

Modify a Customer

```
OBTAIN customer's user ID
OBTAIN new name
OBTAIN new email
OBTAIN new address

SET customer to customer instance with matching user ID

IF new name exists THEN
  SET customer name to new name
END IF

IF new email exists THEN
  SET customer email to new name
END IF

IF new address exists THEN
  SET customer address to new address
END IF
```

Implementation in Java

User

```
1  /**
2   * Defines a user of the Ticket System
3   *
4   * @author Alex Costa
5   * @version 1.00
6   */
7  public class User
8  {
9     // instance variables - store the basic details of the user, including login info
10    public int userID;
11    private String password;
12    protected String name;
13    protected String email;
14    private UserStatuses status;
15    protected UserTypes userType;

16    /**
17     * Constructor for objects of class User
18     * @param lastUserID The id of the last user to create the new ID
19     * @param newName The name of the new user
20     * @param newPwd The password for the new user
21     * @param newEmail The email of the new user
22     */
23    public User(int lastUserID, String newName, String newPwd, String newEmail)
24    {
25       userID = (lastUserID + 1);
26       password = newPwd;
27       name = newName;
28       email = newEmail;
29       status = status.SignedOff;
30    }

31    /**
32     * A method to allow the user to change their password
33     * @param newPwd The new password for the user
34     */
35    public void changePassword(String newPwd)
36    {
37       if(newPwd != null){
38          password = newPwd;
39       }
40    }
```

```
41  /**
42   * A method to allow the user to do their login
43   * @param logPwd The password entered for the login
44   */
45  public void login(String logPwd){
46     if (status == status.SignedOff){
47        if(logPwd== password){
48           status = status.LoggedIn;
49           System.out.println("Welcome back, " + name + "!");
50        }
51        else{
52           System.out.println("You didn't enter the right password");
53        }
54     }
55     else{
56        System.out.println("You are already logged in.");
57     }
58  }

59  /**
60   * A method that allows the user to log off
61   */
62  public void logOff(){
63     status = status.SignedOff;
64     System.out.println("You are now logged off.");
65  }

66  /**
67   * A method that allows to get the name of the user
68   */
69  public String getName(){
70     return name;
71  }

72  /**
73   * Method that allows a user to change their email
74   * @param newEmail The new email of the user
75   */
76  public void setEmail(String newEmail)
77  {
78     email = newEmail;
79  }

80  /**
81   * Method that allows other classes to get the ID of the user
82   * @return The ID of the user in the int type
83   */
84  public int getID()
85  {
86     return userID;
87  }

88  /**
89   * Method that allows other classes to get the type of user
90   * @return The user type for the user
91   */
92  public UserTypes getType()
93  {
```

```
94        return userType;
95    }

96    public String getEmail()
97    {
98     return email;
99    }
100 }
```

Agent

```
1  import java.time.LocalDate;
2  import java.util.HashMap;
3  import java.util.ArrayList;
4  import java.time.format.DateTimeFormatter;
5  /**
6   * Class storing all the information related to an agent of the Ticket System.
7   * This class also specifies all the actions that an agent can do and how they
8   * are performed.
9   *
10  * @author Alex Costa
11  * @version 1.00
12  */
13 public class Agent extends User
14 {
15    // instance variables - hold information about the seats that are allocated
16    //to the agent as well as other contract information
17    private float commission;
18    private LocalDate startDate;
19    private LocalDate endDate;
20    private HashMap<String, Integer> customersManaged;
21    private ArrayList<Ticket> ticketsSold;


22    /**
23     * Constructor for objects of class Agent
24     * @param lastUserID The ID of the last user to create the ID for this user
25     * @param newName The name of the new user
26     * @param newPwd The password for the new user
27     * @param newEmail The e-mail of the new user
28     * @param newCom The commission for the new agent
29     * @param newDate The start date of the agent's contract
30     * @param dayDuration The duration of the contract counted as days
31     */
32    public Agent(int lastUserID, String newName, String newPwd, String newEmail, float newCom, LocalDate newDate, long dayD
33    {
34       super(lastUserID, newName, newPwd, newEmail);
35       super.userType = UserTypes.Agent;
36       commission = newCom;
37       startDate = newDate;
38       endDate = startDate.plusDays(dayDuration);
39       customersManaged = new HashMap<String,Integer>();
40       ticketsSold = new ArrayList<Ticket>();
41    }

42    /**
```

```
43    * A method to create new customers managed by the currently logged in Agent
44    * @param users The ArrayList containing all the users of the system, in order to add the new customer
45    * @param lastUserID The ID of the last created user to create the ID for the new user
46    * @param newName The name of the new user
47    * @param newPwd The password for the new user
48    * @param newEmail The email of the new user
49    */
50   public void createCustomer(int lastUserID, String newName, String newPwd, String newEmail, String newAdd){
51      TicketSystem.users.add(new Customer(lastUserID, newName, newPwd, newEmail, newAdd, super.userID));
52      customersManaged.put(newName, (lastUserID + 1));
53   }

54   /**
55    * A method that allows the agent to see the tickets that they have sold for a specified event
56    * @param eventName The name of the event that the agent is looking for
57    */
58   public void viewTickets(String eventName){
59      ticketsSold.stream().filter(ticket -> ticket.getEventName().equals(eventName))
60      .forEach(ticket -> ticket.printTicket());
61      System.out.println("Total: " +
62      ticketsSold.stream().filter(ticket -> ticket.getEventName().equals(eventName))
63      .count() + " tickets");
64   }

65   /**
66    * A method that allows the agent to see the tickets that they have sold in a date range
67    * @param startRange The date from which to look for tickets (exclusive)
68    * @param endRange The date from which to look for tickets (exclusive)
69    */
70   public void viewTickets(LocalDate startRange, LocalDate endRange){
71      ticketsSold.stream().filter(ticket -> ticket.getDate().isAfter(startRange) && ticket.getDate().isBefore(endRange))
72      .forEach(ticket -> ticket.printTicket());
73      System.out.println("Total " +
74      ticketsSold.stream().filter(ticket -> ticket.getDate().isAfter(startRange) && ticket.getDate().isBefore(endRange))
75      .count() + " tickets");
76   }

77   public void viewTickets()
78   {
79    ticketsSold.forEach(ticket -> ticket.printTicket());
80    System.out.println("Total: " + ticketsSold.stream().count() + " tickets");
81   }

82   /**
83    * Method allowing the agent to see the details of a customer
84    * @param custName The name of the customer
85    */
86   public void viewCustomer(String custName){
87      int index = customersManaged.get(custName);
88      Customer customer = (Customer) TicketSystem.users.get(index);
89      customer.viewDetails();
90   }

91   /**
92    * Method allowing the agent to modify certain aspects of the customer's data
93    * @param custName The name of the customer to look up
94    * @param newName The new name of the customer, can be null
95    * @param newAdd The new address of the customer, can be null
```

```
96      * @param newEmail The new email of the customer, can be null
97      */
98     public void modifyCustomer(String custName, String newName, String newAdd, String newEmail)
99     {
100      int index = customersManaged.getOrDefault(custName, -1);
101      if (index != -1)
102      {
103        Customer customer = (Customer) TicketSystem.users.get(index);

104        if(newName != null){
105          customer.setName(newName);
106          customersManaged.remove(custName);
107          customersManaged.put(newName, index);
108        }
109        if(newAdd != null)
110        {
111          customer.setAddress(newAdd);
112        }
113        if(newEmail != null)
114        {
115          customer.setEmail(newEmail);
116        }
117      }
118    }

119    /**
120    * Method that allows to renew a contract for a certain amount of days
121    * after the normal end of the contract
122    * @param days The number of days that the contract should be extended for
123    */
124    public void renewFor(long days){
125      endDate = endDate.plusDays(days);
126    }

127    /**
128    * Method that allows to assign a new commission to the agent
129    * @param newCom The new commission to be assigned to the agent
130    */
131    public void setCommission(float newCom)
132    {
133     commission = newCom;
134    }

135    /**
136    * Method that allows to change the start date of the contract
137    * @param newDate The new start date for the contract
138    */
139    public void setNewStart(LocalDate newDate)
140    {
141     startDate = newDate;
142    }
143  }
```

Customer

```
1  import java.util.ArrayList;
2  /**
3   * Class storing all  the customer's information and what they can do in the
4   * system.
5   *
6   * To implement: Purchase Ticket method
7   *
8   * @author Alex Costa
9   * @version 1.00
10  */
11 public class Customer extends User
12 {
13    // instance variables - replace the example below with your own
14    private String address;
15    private int agentID;
16    private ArrayList<Ticket> tickets;
17    private int lastTicketID = -1;

18    public Customer(int lastUserID, String newName, String newPwd, String newEmail)
19    {
20     super(lastUserID, newName, newPwd, newEmail);
21     super.userType = UserTypes.Customer;
22     tickets = new ArrayList<Ticket>();
23    }

24    /**
25     * Constructor for objects of class Customer
26     * @param lastUserID The id of the last user to create the new ID
27     * @param newName The name of the new user
28     * @param newPwd The password for the new user
29     * @param newEmail The email address of the new user
30     * @param newAdd The address of the new user
31     * @param agent The ID of the agent able to buy tickets for this customer. Null if no agent
32     */
33    public Customer(int lastUserID, String newName, String newPwd, String newEmail, String newAdd, Integer agent)
34    {
35       super(lastUserID, newName, newPwd, newEmail);
36       super.userType = UserTypes.Customer;
37       address = newAdd;
38       if(agent != null){
39          agentID = agent;
40       }
41       else{
42          agentID = 0;
43       }
44       tickets = new ArrayList<Ticket>();
45    }

46    public void viewTickets()
47    {
48       tickets.forEach(ticket -> ticket.printTicket());
49    }

50    public void viewDetails()
51    {
```

```
52      System.out.println(super.name);
53      System.out.println(super.email);
54      System.out.println(address);
55      System.out.println(agentID);
56    }

57    /**
58     * Method that allows to change the name of the customer
59     * @param newName The new name for the customer
60     */
61    public void setName(String newName)
62    {
63      super.name = newName;
64    }

65    /**
66     * Method that allows to change the address of the customer
67     * @param newAdd The new address for the customer
68     */
69    public void setAddress(String newAdd)
70    {
71      address = newAdd;
72    }

73    /**
74     * Method that allows to change the email address of the customer
75     * @param newEmail The new email address of the customer
76     */
77    public void setEmail(String newEmail)
78    {
79      super.setEmail(newEmail);
80    }

81    public void addTicket(int showID, int eventID, int seatNumber)
82    {
83     tickets.add(new Ticket(lastTicketID, showID, eventID, seatNumber, name));
84    }
85  }
```

Venue Manager

```
1  import java.time.LocalDate;
2  import java.time.LocalDateTime;
3  /**
4   * Class than defines the manager and what they can do in the system
5   *
6   * @author Alex Costa
7   * @version 1.00
8   */
9  public class Manager extends User
10 {
11   /**
12    * Constructor for objects of class Manager
13    * @param lastUserID The id of the last user to create the new ID
14    * @param newName The name of the new user
15    * @param newPwd The password for the new user
```

```
16     * @param newEmail The email of the new user
17     */
18    public Manager(int lastUserID, String newName, String newPwd, String newEmail)
19    {
20       super(lastUserID, newName, newPwd, newEmail);
21       super.userType = UserTypes.Manager;
22    }

23    //TODO: implement the enum for event statuses and finish the implementation
24    /**
25     * Allows the manager to reschedule an event in the system
26     * @param event The event that the manager is working on
27     * @param startDate The new start date for the event
28     * @param endDate The new end date for the event
29     */
30    public void rescheduleEvent(Event event, LocalDate startDate, LocalDate endDate)
31    {
32       if(startDate != null)
33       {
34          event.setStartDate(startDate);
35       }
36       if(endDate != null)
37       {
38          event.setEndDate(endDate);
39       }
40    }

41    /**
42     * A method that allows the manager to reschedule a show from the system
43     * @param show The show that the manager is working on
44     * @param newStart The new start date and time of the show
45     * @param newEnd The new end date and time of the show
46     */
47    public void rescheduleShow(Show show, LocalDateTime newStart, LocalDateTime newEnd)
48    {
49       if(newStart != null)
50       {
51          show.setStart(newStart);
52       }
53       if(newEnd != null)
54       {
55          show.setEnd(newEnd);
56       }
57    }

58    /**
59     * A method that allows the manager to assign a certain promotion to a range of seats
60     * @param seats An array of integers containing the IDs of the seats to which the promotion
61     * is being assigned
62     * @param promotionID The ID of the promotion we wish to assign to the seats
63     * @param show The show to whose seats we are assigning a promotion
64     */
65    public void assignPromotion(int promoID, Show show, int[] seats)
66    {
67     for(int seat : seats)
68     {
69       show.assignPromotion(seat, promoID);
70     }
```

```
71    }

72    /*
73     * A method that allows the manager to change the details of an agent
74     * @param agent The agent which details need to be changed
75     * @param newCom The new commisison for the agent
76     * @param newDate The new start date for the agent's contract
77     */
78    public void changeContract(Agent agent, float newCom, LocalDate newDate)
79    {
80      if(newCom != -1)
81      {
82        agent.setCommission(newCom);
83      }
84      if(newDate != null)
85      {
86        agent.setNewStart(newDate);
87      }
88    }
89  }
```

Event

```
1   import java.util.ArrayList;
2   import java.time.LocalDate;
3   import java.time.LocalDateTime;
4   import java.time.format.DateTimeFormatter;
5   /**
6    * Class that describes what an event is and what it does
7    *
8    * @author Alex Costa
9    * @version 1.00
10   */
11  public class Event
12  {
13    // instance variables - replace the example below with your own
14    private ArrayList<Show> shows;
15    private LocalDate startDate;
16    private LocalDate endDate;
17    private String eventName;
18    private int eventID;
19    private Statuses status;

20    /**
21     * Constructor for objects of class Event
22     * @param newName The name for the new event
23     * @param start The start date for the new event
24     * @param end The end date for the new event
25     * @param lastEventID The ID of the last event in order to create to add the new event
26     */
27    public Event(String newName, LocalDate start, LocalDate end, int lastEventID)
28    {
29      eventName = newName;
30      startDate = start;
31      endDate = end;
32      eventID = (lastEventID + 1);
```

```
33      status = Statuses.Confirmed;
34      shows = new ArrayList<>();
35  }

36  /**
37   * Method that creates a new show for this event
38   * @param start The start time and date of the show
39   * @param end The end time and date of the show
40   * @param MSPC The max seats per customer value for the show
41   */
42  // TODO: figure out how to add the seats to the show
43  public void addShow(LocalDateTime start, LocalDateTime end, int MSPC)
44  {
45      shows.add(new Show(start, end, MSPC, shows.size()));
46  }

47  /**
48   * Method that allows to retrieve all the details for the shows of an event
49   */
50  public void getShows()
51  {
52    shows.stream().forEach(show -> show.getDetails());
53  }

54  /**
55   * Method that allows to change the name of an event
56   * @param newName The new name for the event
57   */
58  public void setName(String newName)
59  {
60    eventName = newName;
61  }

62  /**
63    * Method that allows other classes to get the name of the event
64    * @return The name of the event
65    */
66  public String getName()
67  {
68      return eventName;
69  }

70  /**
71    * Method that allows other classes to get the ID of the event
72    * @return The ID of the event
73    */
74  public int getID()
75  {
76      return eventID;
77  }

78  /**
79   * Method that allows other classes to get the status of the show
80   * @return The status of the show
81   */
82  public Statuses getStatus()
83  {
84    return status;
```

```
85    }
86    /**
87     * Method that allows to change the start date of the event
88     * @param newDate The new start date for the event
89     */
90    public void setStartDate(LocalDate newDate)
91    {
92      startDate = newDate;
93      setStatus(Statuses.Rescheduled);
94    }
95    /**
96     * Method that allows to change the end date of the event
97     * @
98     */
99    public void setEndDate(LocalDate newDate)
100   {
101     endDate = newDate;
102     setStatus(Statuses.Rescheduled);
103   }
104   /**
105    * Method that allows to set a new status to the event
106    * @param newStatus The new status of the event
107    */
108   public void setStatus(Statuses newStatus)
109   {
110     status = newStatus;
111   }
112   /**
113    * Method that allows to get the shows for a specific date
114    * @param showDate The date for which we are looking if the event has shows
115    */
116   public void getShows(LocalDateTime showDate)
117   {
118     shows.stream().filter(show -> show.getStart().getYear() == showDate.getYear() && show.getStart().getDayOfYear() == showDa
119       .forEach(show -> show.getDetails());
120   }
121   /**
122    * Method that displays the details of all shows that are in a range
123    * @param startDate The date from which to start to look for shows (exclusive)
124    * @param endDate The date from which to end looking for shows (exclusive)
125    */
126   public void getShows(LocalDateTime startDate, LocalDateTime endDate)
127   {
128     shows.stream().filter(show -> show.getStart().isAfter(startDate) && show.getEnd().isBefore(endDate))
129       .forEach(show -> show.getDetails());
130   }
131   /**
132    * Method that allows other classes to get a Show object from this class
133    * @param showID The ID of the show to retrieve from the ArrayList
134    * @return A reference to an existing show object
135    */
136   public Show getShow(int showID)
```

```
137   {
138     return shows.get(showID);
139   }

140   /**Method that allows other classes to get a Show object form this event
141   * @param showStart The date and time of the start of the show
142   * @return The reference to the show object
143   */
144   public Show getShow(LocalDateTime showStart)
145   {
146    Show returnShow = new Show();
147    for(Show show : shows)
148    {
149     if(show.getStart().equals(showStart))
150     {
151      returnShow = show;
152     }
153     else
154     {
155      System.out.println("That show does not exist in the system.");
156      returnShow = null;
157     }
158    }
159    return returnShow;
160   }

161   /**
162   * Method that prints the details of the show in the system
163   */
164   public void viewEvent()
165   {
166    DateTimeFormatter format = DateTimeFormatter.ofPattern("dd.MM.yyyy");
167    System.out.println("Name: " + eventName);
168    System.out.println("Start date: " + startDate.format(format));
169    System.out.println("End date: " + endDate.format(format));
170    System.out.println("Status: " + status.toString());
171    System.out.println("Number of shows: " + shows.stream().count());
172    System.out.println();
173   }

174   public void viewShows()
175   {
176    for(Show show : shows)
177    {
178     if(show.getStatus() != Statuses.Cancelled)
179     {
180      show.getDetails();
181     }
182    }
183   }
184 }
```

Show

```
1  import java.util.ArrayList;
2  import java.time.LocalDateTime;
```

```
3  import java.time.format.DateTimeFormatter;

4  /**
5   * Class that contains all the data and methods for the a show
6   * TODO: Find a way to display the seats that aren't booked yet efficiently
7   * Display the prices for the different seats efficiently (aka show the range of seat with
8   * its price in a single Console line)
9   *
10  * @author Alex Costa
11  * @version 1.00
12  */

13 class Show
14 {
15   //instance variables
16   private LocalDateTime start;
17   private LocalDateTime end;
18   private int showID;
19   private Statuses status;
20   private ArrayList<Seat> seats;
21   private int MaxSeatsPerCustomer;


22   /**
23    * Contructor for Show Objects
24    * @param newStart The start date and time for the new show
25    * @param newEnd The end date and time for the new show
26    * @param lastShowID The ID of the last show created
27    * @param MSPC The set max seats per customer for this show
28    */
29   public Show(LocalDateTime newStart, LocalDateTime newEnd, int lastShowID, int MSPC)
30   {
31     start = newStart;
32     end = newEnd;
33     showID = (lastShowID + 1);
34     status = Statuses.Confirmed;
35     MaxSeatsPerCustomer = MSPC;
36     seats = new ArrayList<>();
37   }
38   //So that the compiler leaves me alone
39   public Show(){}

40   public int getID()
41   {
42     return showID;
43   }

44   /**
45    * Method that allows to set a new start date and time to a show
46    * @param newStart The new start date and time for the show
47    */
48   public void setStart(LocalDateTime newStart)
49   {
50     start = newStart;
51   }

52   /**
53    * Method that allows to set a new end date and time to a show
```

```
54   * @param newEnd The new end date and time for the show
55   */
56  public void setEnd(LocalDateTime newEnd)
57  {
58     end = newEnd;
59  }

60  /**
61   * Method that allows to set a new status to the show
62   * @param newStatus The new status for the show
63   */
64  public void setStatus(Statuses newStatus)
65  {
66     status = newStatus;
67  }

68  /**
69   * Method that allows to assign a new MSPC for the show
70   * @param newMSPC The new max seats per customer value for the show
71   */
72  public void setMSPC(int newMSPC)
73  {
74     MaxSeatsPerCustomer = newMSPC;
75  }

76  /**
77   * Method that allows other classes to retrieve the value of the start date
78   * @return The start date and time for the show
79   */
80  public LocalDateTime getStart()
81  {
82     return start;
83  }

84  public Seat getSeat(int seatNumber)
85  {
86    return seats.get(seatNumber);
87  }
88  /**
89   * Method that allows other classes to retrieve the end date and time of the show
90   * @return The end date and time for the show
91   */
92  public LocalDateTime getEnd()
93  {
94     return end;
95  }

96  public Statuses getStatus()
97  {
98    return status;
99  }

100 /**
101  * Method that allows to print the details of the show.
102  * Prints all the details for the show in the console
103  */
104 public void getDetails()
105 {
```

```
106     DateTimeFormatter format = DateTimeFormatter.ofPattern("dd.MM.yyyy hh:mm a");
107     System.out.println("Start Date and Time: " + start.format(format));
108     System.out.println("End Date and Time: " + end.format(format));
109     System.out.println("Maximum Seats that a customer can buy: " + MaxSeatsPerCustomer);
110     System.out.println("Status of the show: " + status.toString());
111     System.out.println();
112   }

113   /**
114    * Method that allows to hold a seat for a show
115    * @param seatNumber The number of the seat
116    */
117   public void holdSeat(int seatNumber)
118   {
119     Seat seat = seats.get(seatNumber);
120     seat.setStatus(SeatStatuses.Held);
121   }

122   /**
123    * Method that allows to unhold a seat for a show
124    * @param seatNumber The number of the seat
125    */
126   public void unholdSeat(int seatNumber)
127   {
128     Seat seat = seats.get(seatNumber);
129     seat.setStatus(SeatStatuses.Unheld);
130   }

131   /**
132    * Method that allows to reserve a seat for a show
133    * @param seatNumber The number of the seat
134    */
135   public void reserveSeat(int seatNumber)
136   {
137     Seat seat = seats.get(seatNumber);
138     seat.setStatus(SeatStatuses.Reserved);
139   }

140   /**
141    * Method that assigns a promotion to a seat
142    * @param seatNum The number of the seat we are assigning the promotion to
143    * @param promotionID The ID of the promotion we are assigning to the seat
144    */
145   public void assignPromotion(int seatNum, int promotionID)
146   {
147     seats.get(seatNum).setPromotion(promotionID);
148   }
149 }
```

Seat

```
1  /**
2   * Class storing the data and behaviour of a seat in the
3   * Online Ticket Sale system
4   *
5   * @author Alex Costa
```

```
 * @version 1.00
 */
public class Seat
{
    // instance variables
    private int seatNumber;
    private SeatStatuses status;
    private int promotionID;

    /**
     * Constructor for objects of class Seat
     * @param number The number of the seat (should correspond with its index in the
     * show's seats ArrayList)
     */
    public Seat(int number)
    {
        seatNumber = number;
        status = SeatStatuses.Unheld;
    }

    /**
     * A method that allows other methods to retrieve the number of the seat
     * @return The number of the seat
     */
    public int getNumber()
    {
        return seatNumber;
    }

    /**
     * Method that allows other classes to retrieve the status of the seat
     * @return The status of the seat (held, unheld, reserved)
     */
    public SeatStatuses getStatus()
    {
        return status;
    }

    /**
     * Method that allows to change the status of a seat
     * @param newStatus Contains the new status for the seat
     */
    public void setStatus(SeatStatuses newStatus)
    {
        status = newStatus;
    }

    /**
     * Method that allows to assign a promotion to a seat
     * @param promoID The ID of the promotion being assigned to the seat
     */
    public void setPromotion(int promoID)
    {
        promotionID = promoID;
    }

    public int getPromoID()
    {
```

```
58      return promotionID;
59    }
60  }
```

Promotion

```
1   import java.time.LocalDate;
2   import java.time.LocalTime;
3   import java.math.BigDecimal;
4   /**
5    * Class that stores all the data and behaviour of a promotion
6    *
7    * TODO: Work on a way to assign a new discount tarif to the promotion
8    *
9    * @author Alex Costa
10   * @version 1.00
11   */
12  public class Promotion
13  {
14    // instance variables
15    private int promotionID;
16    private String name;
17    private LocalDate startDate;
18    private LocalDate endDate;
19    private WeekDay day;
20    private LocalTime startTime;
21    private LocalTime endTime;
22    private BigDecimal priceChild;
23    private BigDecimal priceStudent;
24    private BigDecimal priceAdult;
25    private BigDecimal priceSenior;
26    private int discountID;
27    private DiscountTypes typeDiscount;
28    private int minAmount;
29    private float discount;

30    /**
31     * Constructor for objects of class Promotion
32     * @param lastPromoID The ID of the last promotion to create the one for this promo
33     * @param promoName The name of the new promotion
34     * @param start The start date of the promotion
35     * @param end The end date of the promotion
36     * @param promoDay The day that this promotion runs
37     * @param startT The time that the promotion starts
38     * @param endT The time at which the promotion stops running
39     * @param child The price for a child's ticket
40     * @param student The price for a student's ticket
41     * @param adult The price for an adult's ticket
42     * @param senior The price for a senior's ticket
43     * @param discount The ID of the type of discount to be applied to this promotion
44     */
45    public Promotion(int lastPromoID, String promoName, LocalDate start, LocalDate end, WeekDay promoDay, LocalTime startT
46    , BigDecimal child, BigDecimal student, BigDecimal adult, BigDecimal senior, int discount)
47    {
48      promotionID = (lastPromoID + 1);
49      name = promoName;
```

```
50      startDate = start;
51      endDate = end;
52      day = promoDay;
53      startTime = startT;
54      endTime = endT;
55      priceChild = child;
56      priceStudent = student;
57      priceAdult = adult;
58      priceSenior = senior;
59      discountID = discount;
60      getDiscountDetails();
61    }

62    /**
63     * Method that allows other classes to retrieve the day of the week that this promotion runs on
64     * @return The day of the week that this promotion runs in
65     */
66    public WeekDay getday()
67    {
68      return day;
69    }

70    /**
71     * Method that sets a new week day for the promotion to run on
72     * @param newDay The new day that the promotion will run on
73     */
74    public void setDay(WeekDay newDay)
75    {
76      day = newDay;
77    }

78    /**
79     * Method that allows other classes to retrieve the time at which the promotion starts
80     * @return The time at which the promotion starts
81     */
82    public LocalTime getStartTime()
83    {
84      return startTime;
85    }

86    /**
87     * Method that allows other classes to retrieve the time at which the promotion ends
88     * @return The time at which the promotion stops running
89     */
90    public LocalTime getEndTime()
91    {
92      return endTime;
93    }

94    /**
95     * Method that allows to set a new start time for the promotion
96     * @param newStart The new start time for the promotion
97     */
98    public void setStartTime(LocalTime newStart)
99    {
100     startTime = newStart;
101   }
102   /**
```

```
103     * Method that allows to set a new end time for the promotion
104     * @param newEnd The new end time for the promotion
105     */
106    public void setEndTime(LocalTime newEnd)
107    {
108        endTime = newEnd;
109    }

110    /**
111     * Method that allows other methods to get the start date of the promotion
112     * @return The start date of the promotion
113     */
114    public LocalDate getStartDate()
115    {
116        return startDate;
117    }

118    /**
119     * Method that allows other methods to get the end date of the promotion
120     * @return The end date of the promotion
121     */
122    public LocalDate getEndDate()
123    {
124        return endDate;
125    }

126    /**
127     * Method that allows other methods to get the price of a child ticket
128     * @return The price of a child ticket
129     */
130    public BigDecimal getChildPrice()
131    {
132        return priceChild;
133    }

134    /**
135     * Method that allows other methods to get the price of a student ticket
136     * @return The price of a student ticket
137     */
138    public BigDecimal getStudentPrice()
139    {
140        return priceStudent;
141    }

142    /**
143     * Method that allows other methods to get the price of an adult ticket
144     * @return The price of an adult ticket
145     */
146    public BigDecimal getAdultPrice()
147    {
148        return priceAdult;
149    }

150    /**
151     * Method that allows other methods to get the price of a senior ticket
152     * @return The price of a senior ticket
153     */
154    public BigDecimal getSeniorPrice()
```

```
155   {
156     return priceSenior;
157   }

158   /**
159    * Method that allows to change the price of a child's ticket
160    * @param child The new price for a child's ticket
161    */
162   public void setChildPrice(BigDecimal child)
163   {
164     priceChild = child;
165   }

166   /**
167    * Method that allows to change the price of a student's ticket
168    * @param child The new price for a student's ticket
169    */
170   public void setStudentPrice(BigDecimal student)
171   {
172     priceStudent = student;
173   }

174   /**
175    * Method that allows to change the price of an adult's ticket
176    * @param child The new price for an adult's ticket
177    */
178   public void setAdultPrice(BigDecimal adult)
179   {
180     priceAdult = adult;
181   }

182   /**
183    * Method that allows to change the price of a senior's ticket
184    * @param child The new price for a senior's ticket
185    */
186   public void setSeniorPrice(BigDecimal senior)
187   {
188     priceSenior = senior;
189   }

190   /**
191    * Method that allows to set a new discount to a promotion
192    * @param newDiscount The ID of the new discount applied to the promotion
193    */
194   public void setDiscount(int newDiscount)
195   {
196     discountID = newDiscount;
197     getDiscountDetails();
198   }

199   /**
200    * Method that allows to get the details of the discount tarif that is applied in the promotion
201    */
202   public void getDiscountDetails()
203   {
204     Discount thisDiscount = TicketSystem.discounts.get(discountID);
205     if(thisDiscount.getMinTickets() == -1)
206     {
```

```
207         typeDiscount = typeDiscount.Spend;
208         minAmount = thisDiscount.getMinSpent();
209       }
210       else{
211         typeDiscount = typeDiscount.Tickets;
212         minAmount = thisDiscount.getMinTickets();
213       }

214       discount = thisDiscount.getDiscount();
215     }

216     /**
217      * Method that allows other classes to get the name of the promotion
218      * @return The name of the promotion in the String format
219      */
220     public String getName()
221     {
222       return name;
223     }

224     /**
225      * Method that allows other classes to retrieve the ID of the promotion
226      * @return The ID of the promotion in the integer type
227      */
228     public int getID()
229     {
230       return promotionID;
231     }
232 }
```

Discount

```
1  /**
2   * Class that stores all the data and behaviour of a discount tarif
3   *
4   * @author Alex Costa
5   * @version 1.00
6   */
7  public class Discount
8  {
9    // instance variables
10   private int discountID;
11   private String discountName;
12   private int minTickets;
13   private int minSpent;
14   private float discountPercent;

15   /**
16    * Constructor for objects of class Discount
17    * @param lastDiscountID The ID of the last discount to create the ID for this one
18    * @param name The name of the new discount tarif
19    * @param tickets The minimum number of tickets to be bought (can be null)
20    * @param spent The minimum amount of money that need to be spent to active the discount (can be null)
21    * @param percent The percentage of discount that is given when the min is achieved
22    */
23   public Discount(int lastDiscountID, String name, Integer tickets, Integer spent, float percent)
```

```
24  {
25      discountID = (lastDiscountID + 1);
26      discountName = name;
27      if(tickets != null){minTickets = tickets;}else{ minTickets = -1;}
28      if(spent != null){minSpent = spent;}else{spent = -1;}
29      discountPercent = percent;
30  }

31  /**
32   * Method that allows other classes to get the name of the discount
33   * @return The name of the discount
34   */
35  public String getName()
36  {
37      return discountName;
38  }

39  /**
40   * Method that allows other classes to get the minimum number of tickets to activate the discount
41   * @return The minimum number of tickets to buy to activate the discount
42   */
43  public int getMinTickets()
44  {
45      return minTickets;
46  }

47  /**
48   * Method that allows other classes to get the minimum to be spent to activate the discount
49   * @return The minimum to be spent
50   */
51  public int getMinSpent()
52  {
53      return minSpent;
54  }

55  /**
56   * Method that allows other classes to get the percentage of discount given
57   * @return The percentage of discount
58   */
59  public float getDiscount()
60  {
61      return discountPercent;
62  }

63  public int getID()
64  {
65    return discountID;
66  }

67  /**
68   * Method that allows to set a new name to this discount tarif
69   * @param newName The new name for this discount tarif
70   */
71  public void setName(String newName)
72  {
73      discountName = newName;
74  }
```

```
75    /**
76     * Method that allows to set a new minimum of tickets to be bought to active the discount
77     * @param newMin The new minimum of tickets
78     */
79    public void setMinTickets(Integer newMin)
80    {
81       minTickets = newMin;
82    }

83    /**
84     * Method that allows to set a new minimum amount of money to be spent to activate the discount
85     * @param newMin The new minimum amount of money to be spent‘
86     */
87    public void setMinSpent(Integer newMin)
88    {
89       minSpent = newMin;
90    }
91 }
```

Ticket

```
1  import java.math.BigDecimal;
2  import java.time.LocalDateTime;
3  import java.time.LocalDate;
4  import java.time.format.DateTimeFormatter;
5  /**
6   * Write a description of class Ticket here.
7   *
8   * @author (your name)
9   * @version (a version number or a date)
10  */
11 public class Ticket
12 {
13    // instance variables - replace the example below with your own
14    private int ticketID;
15    private int showID;
16    private int eventID;
17    private int seatNumber;
18    private BigDecimal price;
19    private String customerName;
20    private DateTimeFormatter date = DateTimeFormatter.ofPattern(”dd.MM.yyyy HH:mm”);

21    /**
22     * Constructor for objects of class Ticket
23     * @param lastTicketID The index of the last ticket sold to create the one for this ticket
24     * @param show The ID of the show this ticket is for
25     * @param event The ID of the event this ticket is for
26     * @param seat The number of the seat this ticket is for
27     * @param money The amount of money paid for the ticket
28     * @param customer The name of the customer that bought the ticket
29     */
30    public Ticket(int lastTicketID, int show, int event, int seat, String customer)
31    {
32       ticketID = (lastTicketID + 1);
33       showID = show;
34       eventID = event;
```

```
35      seatNumber = seat;
36      int promoID = TicketSystem.events.get(eventID).getShow(eventID).getSeat(seat).getPromoID();
37      price = TicketSystem.promotions.get(promoID).getAdultPrice();
38      customerName = customer;
39    }

40    /**
41     * Method that allows to print the details of the ticket
42     */
43    public void printTicket()
44    {
45      System.out.println("Event: " + TicketSystem.events.get(eventID).getName());
46      System.out.println("Start: " + TicketSystem.events.get(eventID).getShow(showID).getStart().format(date));
47      System.out.println("End: " + TicketSystem.events.get(eventID).getShow(showID).getEnd().format(date));
48      System.out.println("Seat: " + seatNumber);
49      System.out.println("Price: " + price.toString());
50    }

51    /**
52     * Method that allows other classes to get the name of the event this ticket belongs to
53     * @return The name of the event that this ticket is for
54     */
55    public String getEventName()
56    {
57      return TicketSystem.events.get(eventID).getName();
58    }

59    public LocalDate getDate()
60    {
61      return TicketSystem.events.get(eventID).getShow(showID).getStart().toLocalDate();
62    }
63  }
```

Ticket System

```
1  import java.util.ArrayList;
2  import java.time.LocalDate;
3  import java.time.LocalDateTime;
4  import java.time.LocalTime;
5  import java.math.BigDecimal;
6  import java.time.format.DateTimeFormatter;

7  /**
8   * Class containing all the information and behaviour for the TicketSystem
9   *
10  * @author Alex Costa
11  * @version 1.00
12  */
13 public class TicketSystem
14 {
15    // instance variables - replace the example below with your own
16    public static ArrayList<Event> events = new ArrayList<>();
17    public static ArrayList<Promotion> promotions = new ArrayList<>();
18    public static ArrayList<Discount> discounts = new ArrayList<>();
19    public static ArrayList<User> users = new ArrayList<>();
20    private int lastPromoID;
```

```
21    private int lastUserID;
22    private int lastEventID;
23    private int lastDiscountID;
24    private UserTypes currentUserType;
25    private int userID;

26    /**
27     * Constructor for objects of class TicketSystem
28     */
29    public TicketSystem()
30    {
31     lastPromoID = promotions.size() - 1;
32     lastUserID = users.size() - 1;
33     lastEventID = events.size() - 1;
34     lastDiscountID = discounts.size() - 1;
35    }

36    /**
37    * Method that allows other methods to retrieve an Event from
38    * the events ArrayList based on its name
39    * @param eventName The name of the event that we are looking for
40    * @return The event object. If no event corresponds to eventName,
41    * null will be sent out as an answer
42    */
43    public Event findEvent(String eventName)
44    {
45     int index = -1;
46     for(Event event : events)
47     {
48      if(event.getName().equals(eventName))
49      {
50       index = event.getID();
51      }
52     }

53     if(index == -1)
54     {
55      System.out.println("There is no event named " + eventName + " in the system.");
56      System.out.println();
57      return null;
58     }
59     else
60     {
61      return events.get(index);
62     }
63    }

64    /**Method that allows other methods to find a particular show
65    * @param eventName The name of the event that is being looked for
66    * @param showStart The date and time of the start of the show
67    * in the dd.MM.yyyy HH:mm format
68    * @return A reference to the show object that is being worked on
69    */
70    public Show findShow(String eventName, String showStart)
71    {
72     if(findEvent(eventName) != null)
73     {
74      DateTimeFormatter format = DateTimeFormatter.ofPattern("dd.MM.yyyy HH:mm");
```

```
75      LocalDateTime startDate = LocalDateTime.parse(showStart, format);
76      Show show = findEvent(eventName).getShow(startDate);
77      return show;
78    }
79    else
80    {
81      return null;
82    }
83  }

84  /**
85   * Method that allows to find a promotion based on its name
86   * @param promoName The name of the promotion that the user is looking for
87   * @return The instance of the Promotion class that corresponds to that promotion
88   */
89  public Promotion findPromotion(String promoName)
90  {
91    int index = -1;
92    for(Promotion promo : promotions)
93    {
94      if(promo.getName().equals(promoName))
95      {
96        index = promotions.indexOf(promo);
97      }
98    }

99    if(index == -1)
100   {
101     System.out.println("There is no promotion named " + promoName + " in the system.");
102     System.out.println();
103     return null;
104   }
105   else
106   {
107     return promotions.get(index);
108   }
109 }

110 /**
111  * Method that allows other methods to find a user by their name
112  * @param userName The name of the user that is being looked for
113  * @return The user object that corresponds to that user
114  */
115 public User findUser(String userName)
116 {
117   int index = -1;
118   for(User user : users)
119   {
120     if(user.getName().equals(userName))
121     {
122       index = users.indexOf(user);
123     }
124   }

125   if(index == -1)
126   {
127     return null;
128   }
```

```
129     else
130     {
131       return users.get(index);
132     }
133   }

134   public Discount findDiscount(String discountName)
135   {
136     int index = -1;
137     for(Discount discount : discounts)
138     {
139       if(discount.getName().equals(discountName))
140       {
141         index = discounts.indexOf(discount);
142       }
143     }

144     if(index == -1)
145     {
146       System.out.println("There is no discount tarif with the name " + discountName + " in the system");
147       System.out.println();
148       return null;
149     }
150     else
151     {
152       return discounts.get(index);
153     }
154   }

155   /**
156    * Method that checks if the logged in user is a manager
157    * @return a boolean that says if the manager is logged in or not
158    */
159   public boolean isManager()
160   {
161     if(currentUserType.equals(UserTypes.Manager))
162     {
163       return true;
164     }
165     else
166     {
167       System.out.println("You are not authorized to perform this operation.");
168       System.out.println("You have to be a manager to access this function.");
169       System.out.println();
170       return false;
171     }
172   }

173   public boolean isAgent()
174   {
175     if(currentUserType.equals(UserTypes.Agent))
176     {
177       return true;
178     }
179     else
180     {
181       System.out.println("You are not authorized to perform this operation.");
182       System.out.println("You have to be an agent to access this function.");
```

```
183         System.out.println();
184         return false;
185       }
186     }

187     public boolean isCustomer()
188     {
189       if(currentUserType.equals(UserTypes.Customer))
190       {
191         return true;
192       }
193       else
194       {
195         System.out.println("You are not authorized to perform this operation.");
196         System.out.println("You have to be a customer to access this function.");
197         System.out.println();
198         return false;
199       }
200     }

201     /**
202      * Method that allows the manager to delete an agent from the system,
203      * which in practicality means cancelling their contract
204      * @param agentName The name of the agent
205      */
206     public void cancelContract(String agentName)
207     {
208       if(isManager())
209       {
210         User possibleAgent = findUser(agentName);
211         if(possibleAgent != null && possibleAgent.getType().equals(UserTypes.Agent))
212         {
213           users.remove(users.indexOf(possibleAgent));
214           System.out.println("The agent " + agentName + " has been removed from the system");
215           System.out.println();
216         }
217         else
218         {
219           System.out.println("There is no agent named " + agentName + " in the system.");
220           System.out.println();
221         }
222       }
223     }


224     /**
225      * Method that allows a logged in Manager to create a new event in the system
226      * @param startDate The start date of the event in format dd.MM.yyyy
227      * @param endDate The end date of the event in format dd.MM.yyyy
228      * @param name The name of the event
229      */
230     public void addEvent(String startDate, String endDate, String name)
231     {
232       DateTimeFormatter format = DateTimeFormatter.ofPattern("dd.MM.yyyy");
233       if(isManager())
234       {
235         events.add(new Event(name, LocalDate.parse(startDate, format), LocalDate.parse(endDate, format), lastEventID));
236         lastEventID++;
```

```
237       }
238     }

239     /**
240      * Method that allows a logged in Manager to reschedule a previously created event
241      * @param eventName The event that the manager wants to change
242      * @param startDate The new start date for the event (if that date is not to be changed
243      * then send a null) in format dd.MM.yyyy
244      * @param endDate The new end date for the event (if that date is not to be changed
245      * then send a null) in format dd.MM.yyyy
246      */
247     public void rescheduleEvent(String eventName, String startDate, String endDate)
248     {
249       DateTimeFormatter format = DateTimeFormatter.ofPattern("dd.MM.yyyy");
250       if(isManager())
251       {
252         if(findEvent(eventName) != null)
253         {
254           Manager manager = (Manager) users.get(userID);
255           manager.rescheduleEvent(findEvent(eventName), LocalDate.parse(startDate, format), LocalDate.parse(endDate, forma
256         }
257       }
258     }

259     /**
260     * Method that allows a logged in manager to cancel an event
261     * @param eventName The name of the event that the manager wants to cancel
262     */
263     public void cancelEvent(String eventName)
264     {
265      if(isManager())
266      {
267       if(findEvent(eventName) != null)
268       {
269        findEvent(eventName).setStatus(Statuses.Cancelled);
270       }
271      }
272     }

273     /**
274     * Method that allows the manager to add a show to an event in the system
275     * @param event The event for which the manager is adding the show
276     * @param start The start time and date for the show in format dd.MM.yyyy HH:mm
277     * @param end The end time and date for the show in format dd.MM.yyyy HH:mm
278     * @param mspc The maximum seats per customer value for this show
279     */
280     public void addShow(String eventName, String start, String end, int mspc)
281     {
282      DateTimeFormatter format = DateTimeFormatter.ofPattern("dd.MM.yyyy HH:mm");
283      if(isManager())
284      {
285       if(findEvent(eventName) != null)
286       {
287        findEvent(eventName).addShow(LocalDateTime.parse(start, format), LocalDateTime.parse(end, format), mspc);
288       }
289      }
290     }
```

```
291   /**
292   * Method that allows the manager to reschedule a show
293   * @param eventName The name of the show's event
294   * @param start The current start time and date of the show
295   * @param newStart The new start date and time for the show (can be null if there is no change)
296   * in the pattern dd.MM.yyyy HH:mm
297   * @param newEnd The end date and time for the show (can be null if there is no change)
298   * in the format dd.MM.yyyy HH:mm
299   */
300   public void rescheduleShow(String eventName, String start, String newStart, String newEnd)
301   {
302    DateTimeFormatter format = DateTimeFormatter.ofPattern("dd.MM.yyyy HH:mm");
303    if(isManager())
304    {
305     Show show = findShow(eventName, start);
306     if(show != null)
307     {
308      Manager manager = (Manager) users.get(userID);
309      manager.rescheduleShow(show, LocalDateTime.parse(newStart, format), LocalDateTime.parse(newEnd, format));
310     }
311    }
312   }

313   /**
314   * Method that allows the manager to cancel a show on the system
315   * @param eventName The name of the show's event
316   * @param start The start date and time of the show in the format dd.MM.yyyy HH:mm
317   */
318   public void cancelShow(String eventName, String start)
319   {
320    if(isManager())
321    {
322     Show show = findShow(eventName, start);
323     if(show !=null)
324     {
325      show.setStatus(Statuses.Cancelled);
326     }
327    }
328   }

329   /**
330   * A method that allows the manager to change the max seats per customer
331   * of a show.
332   * @param eventName The name of the show's event
333   * @param start The start date and time of the show in the format dd.MM.yyyy HH:mm
334   * @param newMSPC The new max seats per customer value for the show
335   */
336   public void changeMSPC(String eventName, String start, int newMSPC)
337   {
338    if(isManager())
339    {
340     Show show = findShow(eventName, start);
341     if(show !=null)
342     {
343      show.setMSPC(newMSPC);
344     }
345    }
346   }
```

```
347  /**
348  * Method that allows a manager to add a promotion to the system
349  * @param promoName Name of the new promotion
350  * @param start The start date of the promotion in the format dd.MM.yyyy
351  * @param end The end date of the promotion in the format dd.MM.yyyy
352  * @param promoDay The day that this promotion runs
353  * @param startT The time that the promotion starts in the format HH:mm
354  * @param endT The time at which the promotion stops running in the format HH:mm
355  * @param child The price for a child's ticket
356  * @param student The price for a student's ticket
357  * @param adult The price for an adult's ticket
358  * @param senior The price for a senior's ticket
359  * @param discountName The name of the discount tarif to be applied to this promotion
360  */
361 public void addPromotion(String promoName, String start, String end, WeekDay promoDay, String startTime, String endTime
362 {
363   DateTimeFormatter date = DateTimeFormatter.ofPattern("dd.MM.yyyy");
364   DateTimeFormatter hour = DateTimeFormatter.ofPattern("HH:mm");
365   if(isManager())
366   {
367     if(findDiscount(discountName) != null)
368     {
369       int discountID = findDiscount(discountName).getID();
370       promotions.add(new Promotion(lastPromoID, promoName, LocalDate.parse(start, date), LocalDate.parse(end, date), pron
371       lastPromoID++;
372     }
373   }
374 }

375 /**
376  * Method that allows a manager to assign a promotion to a specific seat in a show
377  * @param promotionName The name of the promotion that is being assigned
378  * @param eventName The name of the show's event for which the promotion is being assigned
379  * @param start The start date and time of the show for which seats are being assigned a promotion
380  * @param seats Array containinng the numbers of these to which the selected promotion is being assigned.
381  */
382 public void assignPromotion(String promotionName, String eventName, String start, int[] seats)
383 {
384   if(isManager())
385   {
386     if(findPromotion(promotionName) != null)
387     {
388       int promoID = findPromotion(promotionName).getID();
389       Show show = findShow(eventName, start);
390       if(show != null)
391       {
392         Manager manager = (Manager) users.get(userID);
393         manager.assignPromotion(promoID, show, seats);
394       }
395     }
396   }
397 }

398 /**
399  * Method that allows the manager to delete a promotion from the system
400  * @param promotionName The name of the promotion to be deleted
401  */
```

```
402  public void deletePromotion(String promotionName)
403  {
404   if(isManager())
405   {
406    if(findPromotion(promotionName) != null)
407    {
408     promotions.remove(findPromotion(promotionName));
409    }
410   }
411  }

412  /**
413  * Method that allows a manager to add a new agent to the system with their contract all set up
414  * @param name The name of the new agent
415  * @param newPwd The password of the new agent
416  * @param newEmail The email of the new agent
417  * @param newCom The commission of the new agent
418  * @param newDate The start date of the agent's contract in format dd.MM.yyyy
419  * @param dayDuration The duration of the contract measured in days
420  */
421  public void addContract(String name, String newPwd, String newEmail, float newCom, String newDate, long dayDuration)
422  {
423   DateTimeFormatter date = DateTimeFormatter.ofPattern("dd.MM.yyyy");
424   if(isManager())
425   {
426    users.add(new Agent(lastUserID, name, newPwd, newEmail, newCom, LocalDate.parse(newDate, date), dayDuration));
427    lastUserID++;
428   }
429  }

430  /**
431  * Method that allows a manager to change the contract that it has with an agent
432  * @param agentName The name of the agent for which the manager is changing the contract
433  * @param newCom The new commission for the agent (set to -1 means no change)
434  * @param newDate The new startDate for the contract (set null for no changes) in the format
435  * dd.MM.yyyy
436  */
437  public void changeContract(String agentName, float newCom, String newDate)
438  {
439   if(isManager())
440    {
441      User possibleAgent = findUser(agentName);
442      if(possibleAgent != null && possibleAgent.getType().equals(UserTypes.Agent))
443      {
444        DateTimeFormatter date = DateTimeFormatter.ofPattern("dd.MM.yyyy");
445        Manager thisOne = (Manager) users.get(userID);
446        thisOne.changeContract((Agent) possibleAgent, newCom, LocalDate.parse(newDate, date));
447      }
448      else
449      {
450        System.out.println("There is no agent named " + agentName + " in the system.");
451        System.out.println();
452      }
453    }
454  }

455  /**
456  * Method that allows to renew a contract for a certain amount of days
```

```
457     * after the normal end of the contract
458     * @param agentName The name of the agent for which the contract is being extended
459     * @param days The number of days that the contract should be extended for
460     */
461    public void renewContract(String agentName, long days)
462    {
463     if(isManager())
464     {
465      Agent possibleAgent = (Agent) findUser(agentName);
466      if(possibleAgent != null && possibleAgent.getType().equals(UserTypes.Agent))
467      {
468       possibleAgent.renewFor(days);
469      }
470      else
471      {
472        System.out.println("There is no agent named " + agentName + " in the system.");
473        System.out.println();
474      }
475     }
476    }

477    /**
478     * Method that allows for new users to register to the system
479     * @param newName The new name for the user
480     * @param newPwd The password for the new user
481     * @param newEmail The email of the new user
482     * @param userKind The type of user (Customer, Manager)
483     */
484    public void register(String newName, String newPwd, String newEmail, UserTypes userKind)
485    {
486     if(userKind == UserTypes.Customer)
487     {
488      users.add(new Customer(lastUserID, newName, newPwd, newEmail));
489      lastUserID++;
490     }
491     else if(userKind == UserTypes.Manager)
492     {
493      users.add(new Manager(lastUserID, newName, newPwd, newEmail));
494     }
495    }

496    /**
497     * Method that allows users to login into the system
498     * @param email The email of the user that is logging in
499     * @param pwd The password provided by the user
500     */
501    public void login(String email, String pwd)
502    {
503     int index = -1;
504     for(User user : users)
505     {
506       if(user.getEmail().equals(email))
507       {
508         index = users.indexOf(user);
509         userID = index;
510       }
511     }
```

```
512     if(index == -1)
513     {
514       System.out.println("The email that you are using isn't registered with any account");
515     }
516     else
517     {
518       users.get(index).login(pwd);
519     }

520     currentUserType = users.get(index).getType();
521   }

522   /**
523   * Method that displays all the events that aren't cancelled
524   */
525   public void viewEvents()
526   {
527     for(Event event : events)
528     {
529       if(event.getStatus() != Statuses.Cancelled)
530       {
531         event.viewEvent();
532       }
533     }
534   }

535   /**
536   * Method that allows to see the shows from an event
537   */
538   public void viewShows(String eventName)
539   {
540     Event event = findEvent(eventName);
541     if(event != null)
542     {
543       event.viewShows();
544     }
545   }

546   public void buyTicket(String eventName, String start, int seatTicket)
547   {
548     if(isCustomer())
549     {
550       Show show = findShow(eventName, start);
551       if(show != null)
552       {
553         show.reserveSeat(seatTicket);
554         Customer cust = (Customer) users.get(userID);
555         cust.addTicket(show.getID(), findEvent(eventName).getID(), seatTicket);
556         System.out.println("Your ticket has been bought");
557       }
558       else
559       {
560         System.out.println("Sorry, but there was a problem buying your ticket.");
561       }
562     }
563   }

564   public void viewTickets()
```

```
565     {
566       if(isCustomer())
567       {
568         Customer customer = (Customer) users.get(userID);
569         customer.viewTickets();
570       }
571     }

572     public void viewSoldTickets()
573     {
574       if(isAgent())
575       {
576         Agent agent = (Agent) users.get(userID);
577         agent.viewTickets();
578       }
579     }
580 }
```

Enumerations

Statuses

```
1  /**
2   * Enumeration class Statuses - write a description of the enum class here
3   *
4   * @author (your name here)
5   * @version (version number or date here)
6   */
7  public enum Statuses
8  {
9      Confirmed,
10     Cancelled,
11     Rescheduled
12 }
```

WeekDay

```
1  /**
2   * Enumeration class WeekDay - write a description of the enum class here
3   *
4   * @author (your name here)
5   * @version (version number or date here)
6   */
7  public enum WeekDay
8  {
9      MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
10 }
```

Discount Types

```
1  /**
2   * Enumeration class DiscountTypes - write a description of the enum class here
```

```
 *
 * @author (your name here)
 * @version (version number or date here)
 */
public enum DiscountTypes
{
    Tickets,
    Spend
}
```

Seat Statuses

```
/**
 * Enumeration class SeatStatuses - write a description of the enum class here
 *
 * @author (your name here)
 * @version (version number or date here)
 */
public enum SeatStatuses
{
    Unheld,
    Held,
    Reserved
}
```

User Statuses

```
/**
 * Enumeration class UserStatuses - write a description of the enum class here
 *
 * @author (your name here)
 * @version (version number or date here)
 */
public enum UserStatuses
{
    SignedOff,
    LoggedIn,
}
```

User Types

```
/**
 * Enumeration class UserTypes - write a description of the enum class here
 *
 * @author (your name here)
 * @version (version number or date here)
 */
public enum UserTypes
{
    Customer,
    Agent,
    Manager
}
```

Section C

Appendix

#	Author	Date	Commit Message	Files	++	--
1	James Harris	2017-11-19	feat: add assignment brief	1	0	0
2	Alexandre Costa	2017-11-29	feat: add venue manager use case diagrams	4	0	0
3	Alexandre Costa	2017-11-29	feat: add manager use cases	2	24	0
4	wopian	2017-11-30	feat: add consumer usecases	33	6958	5059
5	Alexandre Costa	2017-12-04	feat: add the functional requirements	1	20	1
6	Alex Costa	2017-12-04	feat: change associations from background functions that should be relationships	2	351	358
7	Alex Costa	2017-12-04	feat: add non-functional requirements	1	10	1
8	James Harris	2017-12-04	feat: add start of data dictionary	6	50	35
9	Qasim12341	2017-12-07	feat:Add agent Use case	1	18	16
10	Qasim Maruf	2017-12-07	feat: add agent usecase diagram	6	8174	6414
11	Qasim Maruf	2017-12-07	feat: add agent usecase diagram	1	0	0
12	Jack Standen	2017-12-07	feat: add class diagram	1	2980	0
13	Jack Standen	2017-12-07	feat: add class diagram to report	3	2977	2977
14	Alex Costa	2017-12-07	feat: develop agent use cases text	1	60	39
15	Alex Costa	2017-12-07	feat: consolidate use case diagrams	5	1396	1385
16	Alex Costa	2017-12-07	feat: consolidate all use cases into one	4	6300	381
17	Alex Costa	2017-12-08	feat: add class diagram and basic classes	1	5169	3
18	Alex Costa	2017-12-08	feat: create main class	1	3431	2309
19	James Harris	2017-12-09	feat: update class diagram	1	1424	306
20	Alexandre Costa	2017-12-12	feat: add getShows method	1	8	0
21	Alexandre Costa	2017-12-12	feat: add set and getShow methods	1	58	1
22	Alexandre Costa	2017-12-12	feat: Create show class with construct	1	37	0
23	Alex Costa	2017-12-13	feat: implement the discount class	1	96	0
24	Alex Costa	2017-12-13	feat: implement the Promotion class	1	235	0
25	Alex Costa	2017-12-13	feat: implement the Seat class	1	62	0
26	Alex Costa	2017-12-13	feat: add new statuses	1	3	1
27	Alex Costa	2017-12-13	feat: add TIcket and some methods	2	64	1
28	Alex Costa	2017-12-14	feat: add TicketSystem and optimise some methods	7	98	23
29	James Harris	2017-12-14	feat: simplify and assign aggregations, compositions & multiplicity to the class diagram	3	4954	1910
30	James Harris	2017-12-14	feat: complete data dictionary	1	92	15
31	James Harris	2017-12-14	feat: add customer pseudo code	2	27	1
32	James Harris	2017-12-14	feat: add event pseudo code	1	36	0
33	James Harris	2017-12-14	feat: add user pseudo code	1	35	0
34	Alex	2017-12-14	feat: add find event and promo methods	1	56	10
35	Alex	2017-12-14	feat: work on add promo method	1	18	22

#	Author	Date	Commit Message	Files	++	--
36	Alex Costa	2017-12-15	feat: implement manager functions and optimise	16	875	211
37	Alex	2017-12-15	feat: add view events and shows	3	76	3
38	James Harris	2017-12-15	feat: add generic get/set pseudo code	8	39	25
39	James Harris	2017-12-15	feat: add agent pseudo code	1	38	6
40	Alex	2017-12-15	feat: add buy and basic view ticket methods	6	127	43
41	Alex	2017-12-15	feat: add view tickets	1	6	0
42	James Harris	2017-12-15	feat: add venue manager pseudo code	1	37	7
43	James Harris	2017-12-15	feat: finish customer pseudo code	1	12	0
44	James Harris	2017-12-15	feat: finish manager pseudo code	1	1	1
45	James Harris	2017-12-15	feat: finish show pseudo code	4	23	57
46	Alex	2017-12-15	feat: fix bugs and screenshots	14	19	12
47	Jack Standen	2017-12-15	feat: add sequence diagram	1	0	0
48	Alex	2017-12-15	feat: add missing src to report	2	9	0
49	James Harris	2017-12-15	feat: test strategy	10	26	19