Design Assignment (2021-22)

David García Ramallal (david.ramallal@udc.es)
Alfredo Javier Freire Bouzas (javier.freire.bouzas@udc.es)

REPORT OF EXERCISE 1 (E1):

Design Principles:

SOLID design principles are the following ones:

- *Single Responsibility Principle*
- *Open-Closed Principle*
- *Liskov Substitution Principle*
- *Dependency Inversion Principle*
- *Interface Segregation Principle*

SOLID principles used in our exercise:

- **Single Responsibility Principle (SRP)**
  "Every object should have a single responsibility entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility."

  Since each of our classes has its own responsibility and there are not "God classes", we can say that SRP is fulfilled.
  Ticket, as the name suggests, contains the data about a specific travelling ticket.
  TicketManager is devoted to work with a list of tickets, so it has methods to add and remove elements to the list or to print the list itself.
  Searching Classes are only dedicated to filter the list with different criteria.
  And and Or realised the filter of the list in different ways.

- **Open-Closed Principle (OCP)**
  "A module (class) should be open for extension but closed for modification."

  Since we can create as many searching criteria as we want without touching the rest of the code, we can say that the Open-Closed principle is applied.

- **Liskov Substitution Principle (LSP)**
  "Subclasses should be substitutable for their base classes."

  This principle can not be fulfilled in our exercise, because our Superclass is abstract so we can not create an instance of it.

- **Dependency Inversion Principle (DIP)**
  "Depend upon abstractions. Do not depend upon concretions."

  We can say that this principle is applied since the implementation of the different searching criteria depend on an interface (<u>SearchTicket</u>).
  When working with lists, we declare them as:
  List<> … = new ArrayList<>() and not as ArrayList<> … = new ArrayList<>(),
  so we do not use concrete classes.

- **Interface Segregation Principle (ISP)**
  "Having many client-specific interfaces is better than having one general-purpose interface."

  We decided to make <u>Operation</u> as an abstract class instead than as an interface, so this principle is not applied (there is only one interface).

Design Pattern:

For this exercise we have focused on working with the following pattern:

- *Composite Pattern*

"Structural pattern that composes objects into tree structures to represent part-whole hierarchies."

Composite Pattern allows us to work with objects and compositions in the same way and it is based in the use of hierarchy and composition. Also, makes easier the task of add new kinds of component to the project and, which is more important, without modifying the existing code that already works.

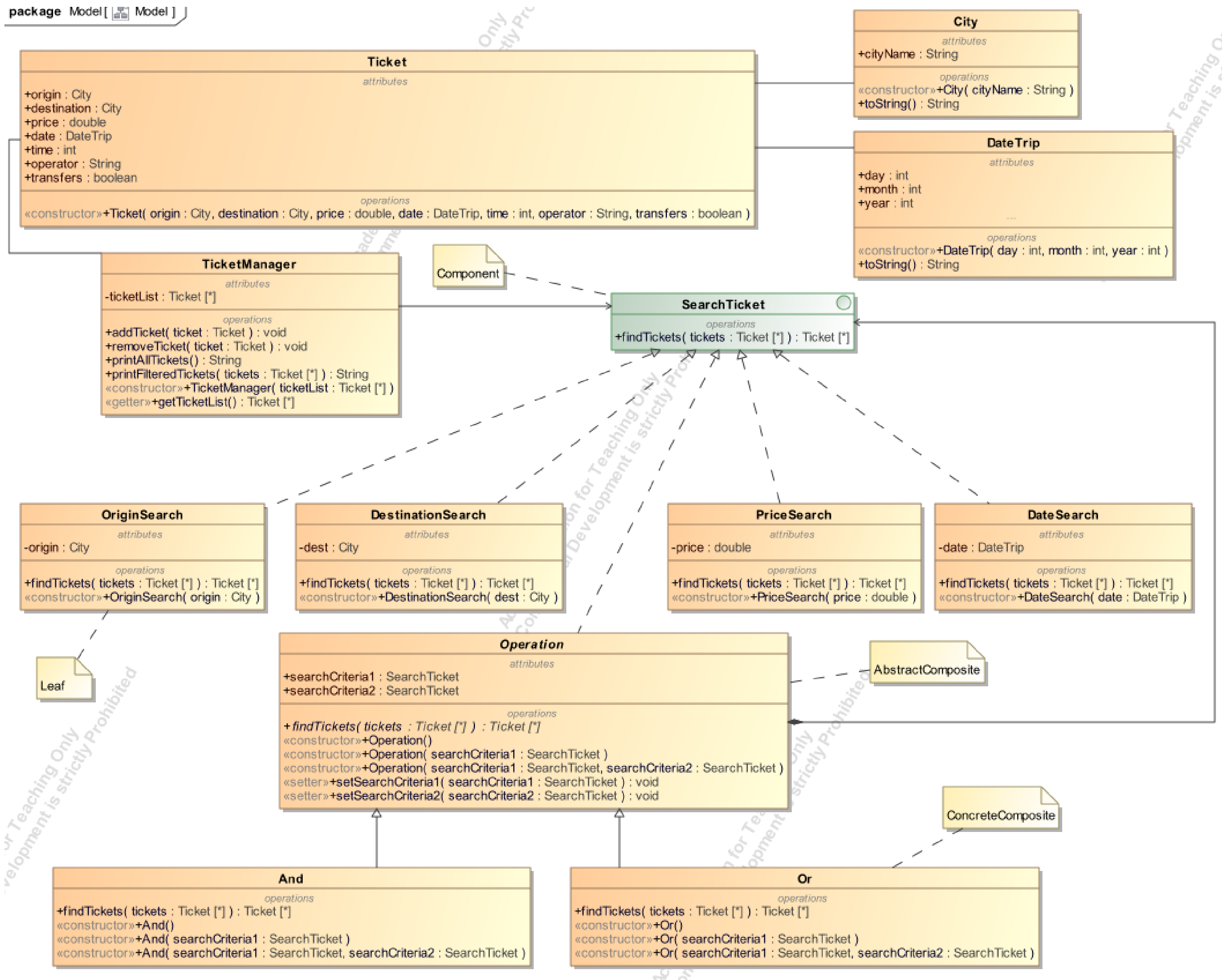
There are three main elements in this pattern:

- <u>Component</u>: declares the interface for objects in the composition. In our code is *SearchTickets*.
- <u>Leaf / Concrete Component:</u> represent an object of the composition that has no children. They are the classes devoted to the searching process
- <u>Composite:</u> defines the behaviour of the components that are parents (*And, Or*).
- <u>Abstract Composite</u>: defines the common behaviour of concrete composites. In our code is the abstract class *Operation*

Our decision of using this specific pattern is related to the need of making the ADD/OR operations. With Composite Pattern we are able to combine as many operations as we want, because ADD/OR can be introduced in the places where we can put a searching algorithm too.

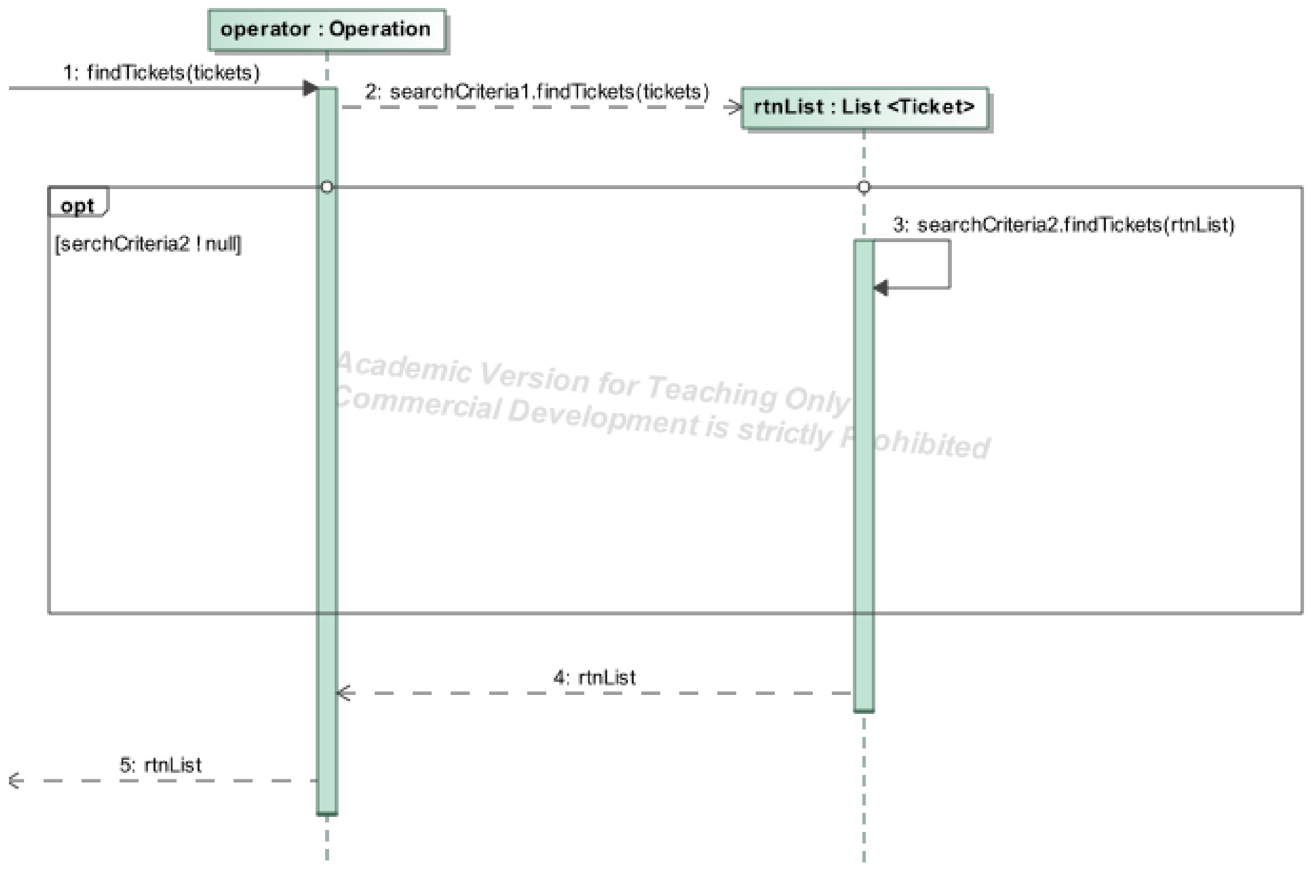
Another reason for working with this pattern is the fact that, as the statement of the exercise says, in the future may be more searching criteria will be added (time, operator, transfers…), and with Composite, we can do that without touching the rest of the code.

Class Diagram:

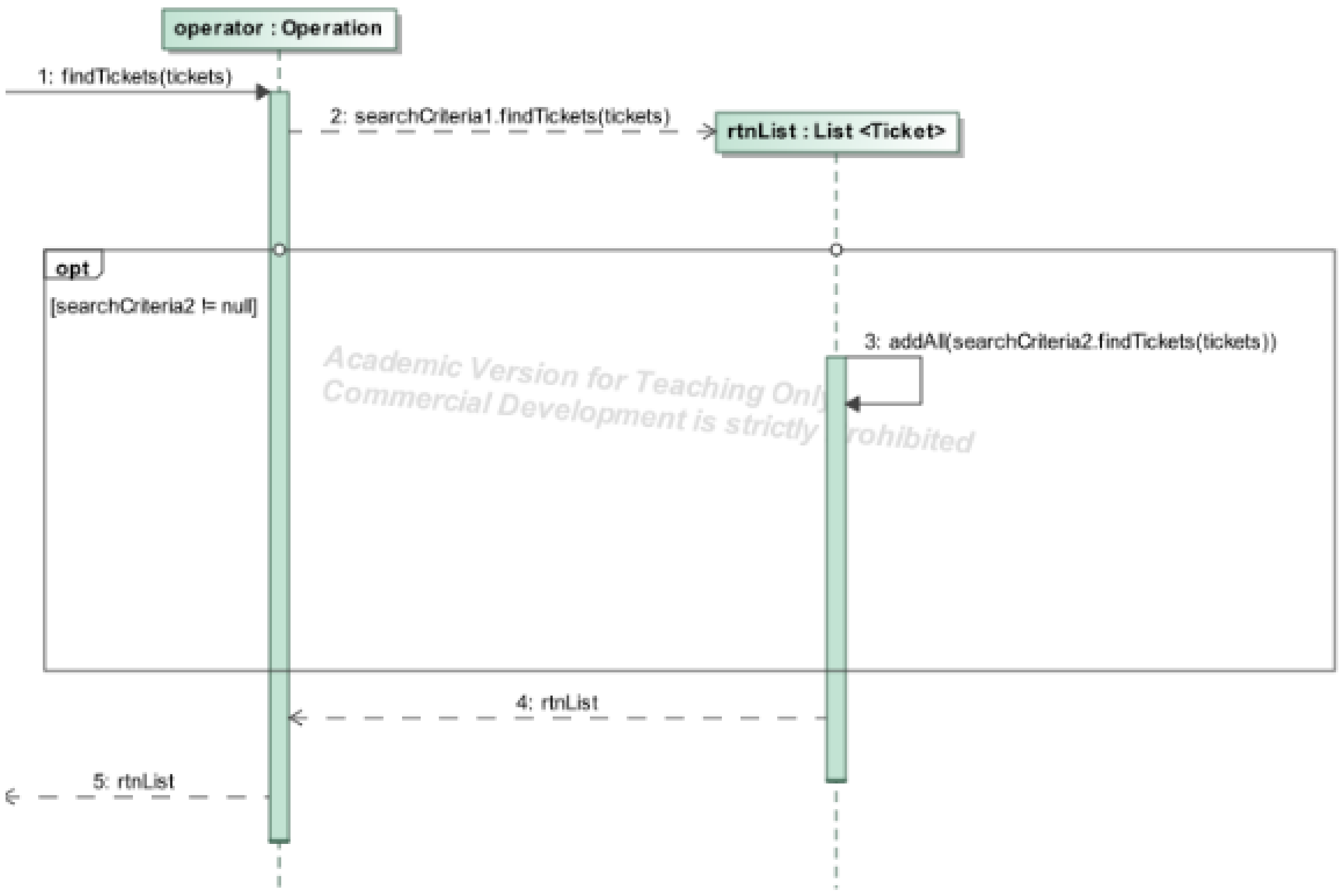

package Model [Model]
City
attributes
+cityName : String
operations
«constructor»+City(cityName : String)
+toString() : String
Ticket
attributes
+origin : City
+destination : City
+price : double
+date : DateTrip
+time : int
+operator : String
+transfers : boolean
operations
«constructor»+Ticket(origin : City, destination : City, price : double, date : DateTrip, time : int, operator : String, transfers : boolean)
DateTrip
attributes
+day : int
+month : int
+year : int
...
operations
«constructor»+DateTrip(day : int, month : int, year : int)
+toString() : String
TicketManager
attributes
-ticketList : Ticket [*]
operations
+addTicket(ticket : Ticket) : void
+removeTicket(ticket : Ticket) : void
+printAllTickets() : String
+printFilteredTickets(tickets : Ticket [*]) : String
«constructor»+TicketManager(ticketList : Ticket [*])
«getter»+getTicketList() : Ticket [*]
Component
SearchTicket
operations
+findTickets(tickets : Ticket [*]) : Ticket [*]
OriginSearch
attributes
-origin : City
operations
+findTickets(tickets : Ticket [*]) : Ticket [*]
«constructor»+OriginSearch(origin : City)
DestinationSearch
attributes
-dest : City
operations
+findTickets(tickets : Ticket [*]) : Ticket [*]
«constructor»+DestinationSearch(dest : City)
PriceSearch
attributes
-price : double
operations
+findTickets(tickets : Ticket [*]) : Ticket [*]
«constructor»+PriceSearch(price : double)
DateSearch
attributes
-date : DateTrip
operations
+findTickets(tickets : Ticket [*]) : Ticket [*]
«constructor»+DateSearch(date : DateTrip)
Leaf
Operation
attributes
+searchCriteria1 : SearchTicket
+searchCriteria2 : SearchTicket
operations
+findTickets(tickets : Ticket [*]) : Ticket [*]
«constructor»+Operation()
«constructor»+Operation(searchCriteria1 : SearchTicket)
«constructor»+Operation(searchCriteria1 : SearchTicket, searchCriteria2 : SearchTicket)
«setter»+setSearchCriteria1(searchCriteria1 : SearchTicket) : void
«setter»+setSearchCriteria2(searchCriteria2 : SearchTicket) : void
AbstractComposite
ConcreteComposite
And
operations
+findTickets(tickets : Ticket [*]) : Ticket [*]
«constructor»+And()
«constructor»+And(searchCriteria1 : SearchTicket)
«constructor»+And(searchCriteria1 : SearchTicket, searchCriteria2 : SearchTicket)
Or
operations
+findTickets(tickets : Ticket [*]) : Ticket [*]
«constructor»+Or()
«constructor»+Or(searchCriteria1 : SearchTicket)
«constructor»+Or(searchCriteria1 : SearchTicket, searchCriteria2 : SearchTicket)

Dynamic Diagrams:

- **AND operation**



- **OR operation**



operator : Operation
1: findTickets(tickets)
2: searchCriteria1.findTickets(tickets)
rtnList : List <Ticket>
opt
[searchCriteria2 != null]
3: addAll(searchCriteria2.findTickets(tickets))
Academic Version for Teaching Only
Commercial Development is strictly prohibited
4: rtnList
5: rtnList