Project DataBase 2

JPA and trigger Application

Giuseppe Calcagno
Giovanni Chiurco

Specifications

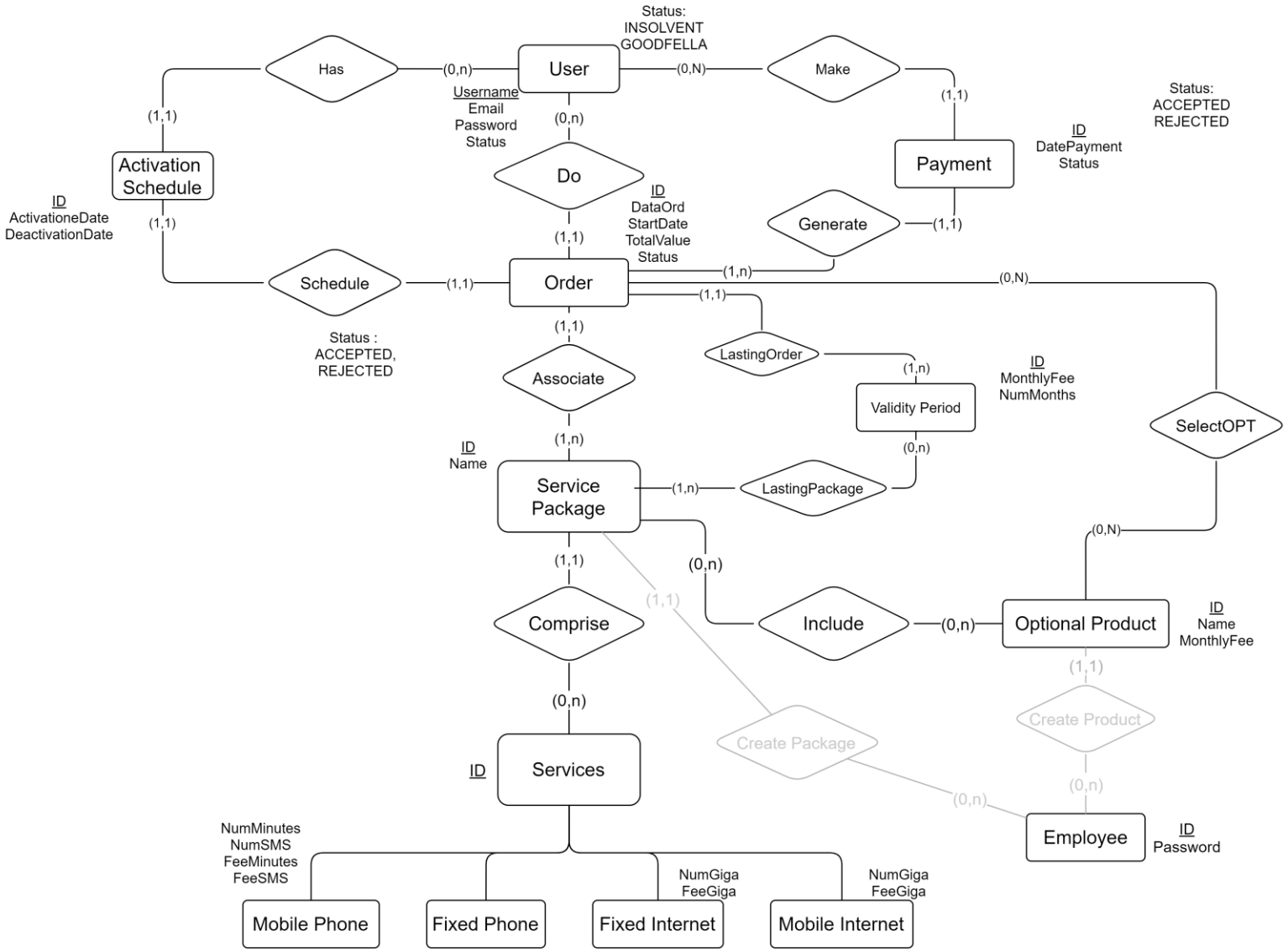
The goal of the project is to build 2 client applications using the same database for a company.

The first is a **consumer application** in which the company's customers can buy the services offered upon registration. It is composed by a landing page where customers can login, register or access to the Home Page where all available services are displayed along with all unpaid orders . A buy page allows the customer to select the ServicePackage together with the optional services, the validity period and the activation date. A confirmation Page is used to resume the order.

The second is an **employee application** in which the company's workers can login and create custom Service Packs and new Optional product. They have also access to the sales report page, where all the major selling data are displayed.

A relational database supports both applications. Triggers are used to organize some types of information, collected in specific table for better viewing.

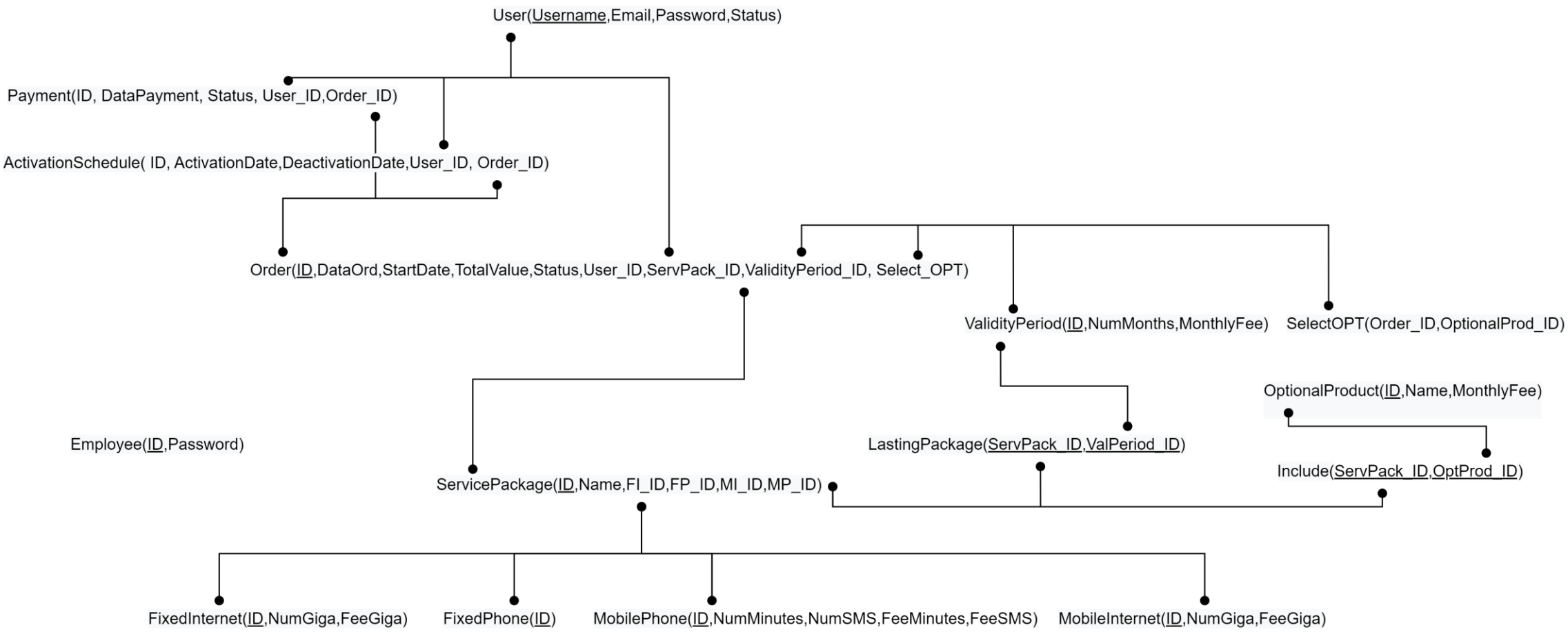
Entity Relationship



Motivations of the ER design

- In our ER design we decided to separate the different kind of user because they have totally different characteristic.
- In our scheme the employee is linked to the services created by him, however this feature has not been implemented as it is not necessary.
- The ActivationSchedule is connected to the user but also to the order, as it contains all the services (optional and not) with a common activation and deactivation date. This could lead to repeated services on the ActivationSchedule table, however it seems to us the best choice in terms of database efficiency and understandability.

Relational model - Link View



Relational model – Table View

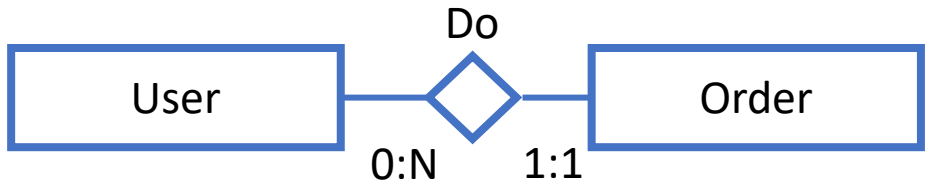
- User(Username,Email,Password,Status)
- Order(ID,DataOrd,StartDate,TotalValue,Status,User_ID,ServPack_ID,ValidityPeriod_ID, SelectedOPT)
- ActivationSchedule(ID, ActivationDate,DeactivationDate,User_ID, Order_ID)
- Payment(ID, DataPayment, Status, User_ID,Order_ID)
- ServicePackage(ID,Name,FI_ID,FP_ID,MI_ID,MP_ID)
- FixedPhone(ID)
- MobilePhone(ID,NumMinutes,NumSMS,FeeMinutes,FeeSMS)
- FixedInternet(ID,NumGiga,FeeGiga)
- MobileInternet(ID,NumGiga,FeeGiga)
- ValidityPeriod(ID,NumMonths,MonthlyFee)
- LastingPackage(ServPack_ID,ValPeriod_ID)
- OptionalProduct(ID,Name,MonthlyFee)
- Include(ServPack_ID,OptProd_ID)
- SelectOPT(Order_ID,OptionalProd_ID)
- Employee(ID,Password)

Motivations of the logical design

- Services are implemented as 4 different tables. The relationship that connects the services to the ServicePackage is implemented with 4 distinct columns. This is possible because each ServicePackage can have at most one service of each type. This avoids, at the expense of some null element in the ServicePackage table, to carry out checks on the uniqueness of the Id between the 4 Service tables, saving many accesses to the memory.
- An ActivationSchedule column in Order table has not been implemented because it is superfluous for functionality purposes.
- As anticipated, the relations regarding the Employee table have not been implemented because they are not necessary for the purposes of the application.

ORM design

Relationship "DO"

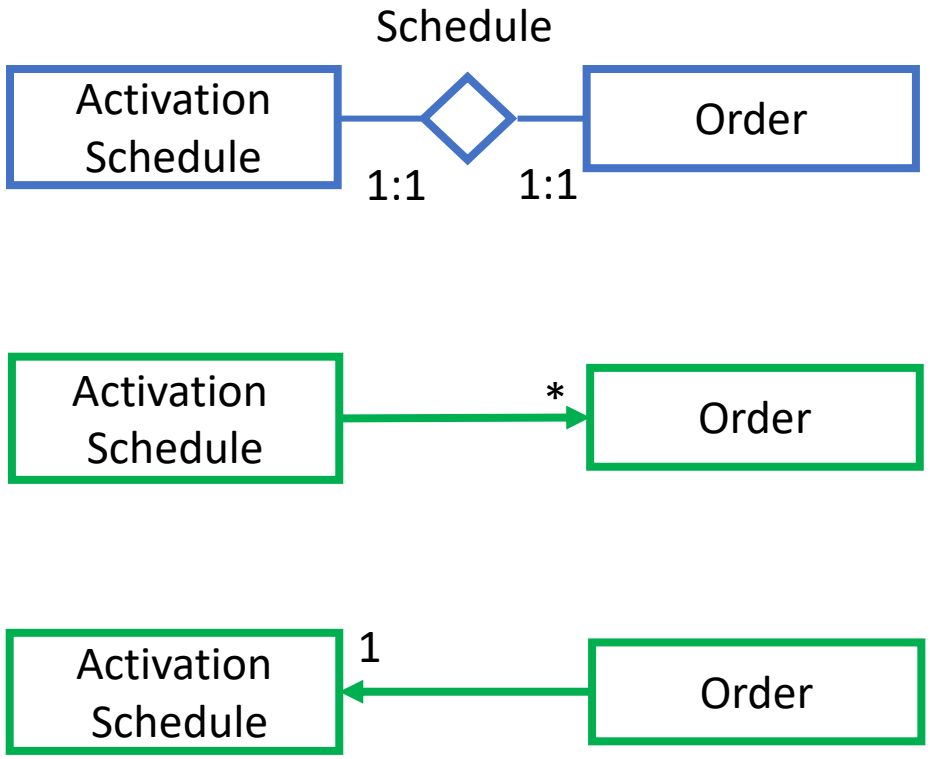


- User → Order @OneToMany is necessary to get the orders of the logged user
  - Fetch type: EAGER
- Order → User @ManyToOne is not requested by the specification

Motivation

- Bidirectional ManyToOne User to Order
  - Owned is the Order
  - FetchType EAGER is used because when a user login, we have to show him all his incumbent orders.
  - Cascade=CascadeType.PERSIST is not necessary because we don't have linked modifications between them.

Relationship "Schedule"

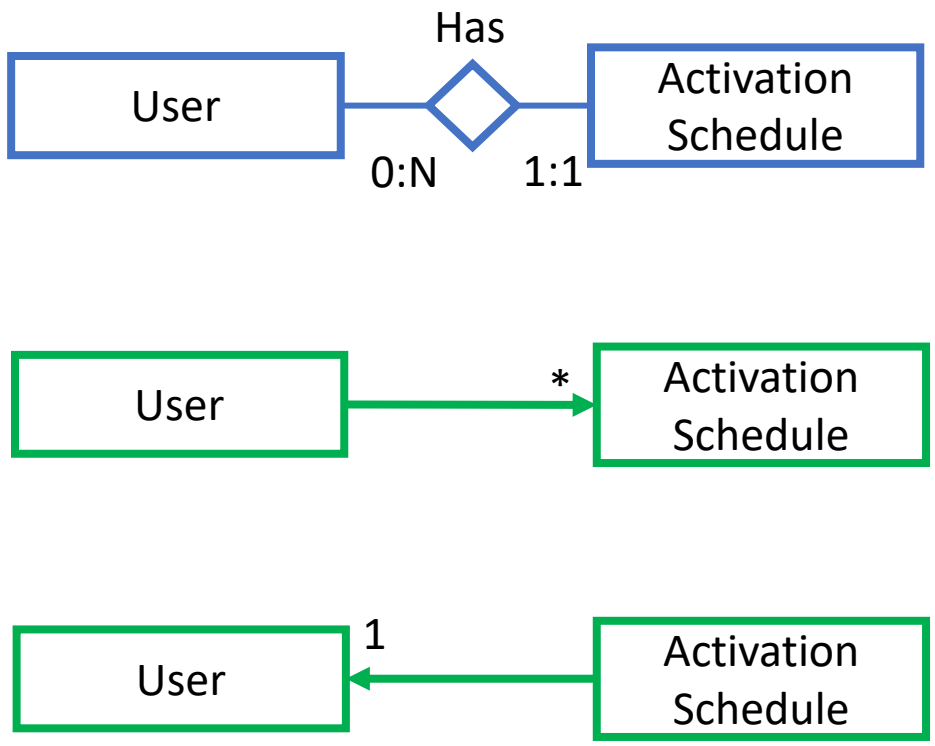


- Activation Schedule → Order @OneToOne is necessary to get the orders from the activation schedule
  - Fetch type: Lazy
- Order → Activation Schedule @ ManyToOne is not requested by the specification.

Motivation

- Bidirectional ManyToOne Order to Activation Schedule
  - Owned is the Activation Schedule
  - FetchType LAZY because we don’t access to Activation Schedule whenever we access order

Relationship "Has"

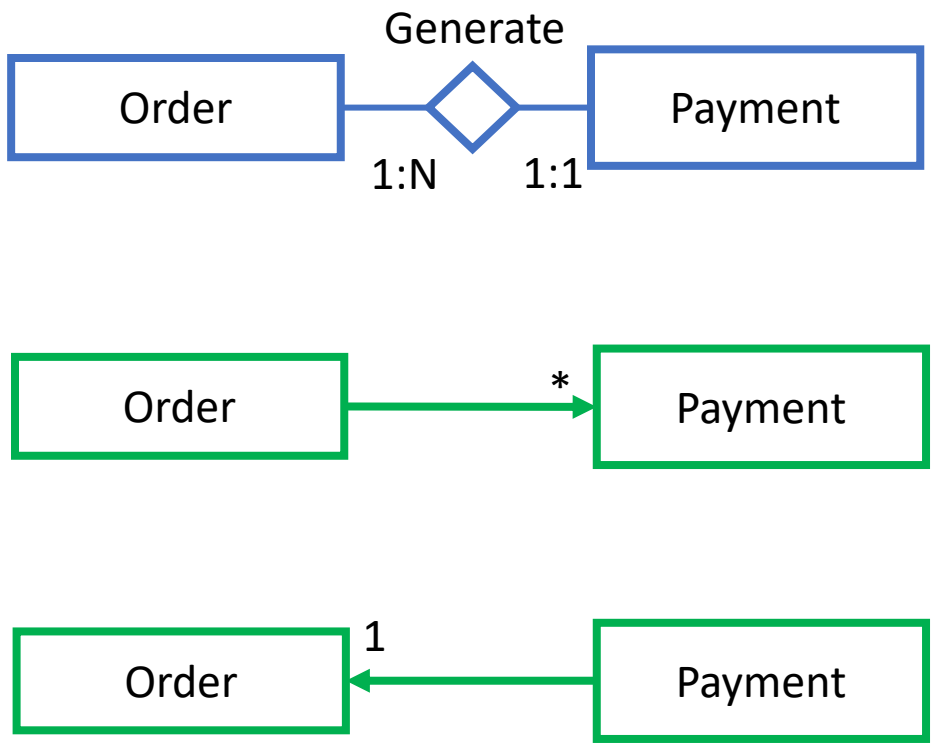


- User → Activation Schedule
  @OneToMany is necessary to get the activation schedules of user
  - Fetch type: Lazy
- Activation Schedule → User
  @ManyToOne is not requested by the specification

Motivation

- Bidirectional ManyToOne User to Activation Schedule
  - Owned is the Activation Schedule
  - PERSIST is not necessary because we don't need to propagate any modifications. Activation Schedule is managed by triggers.
  - FetchType LAZY because we don’t access to Activation Schedule whenever we access User

Relationship "Generate"

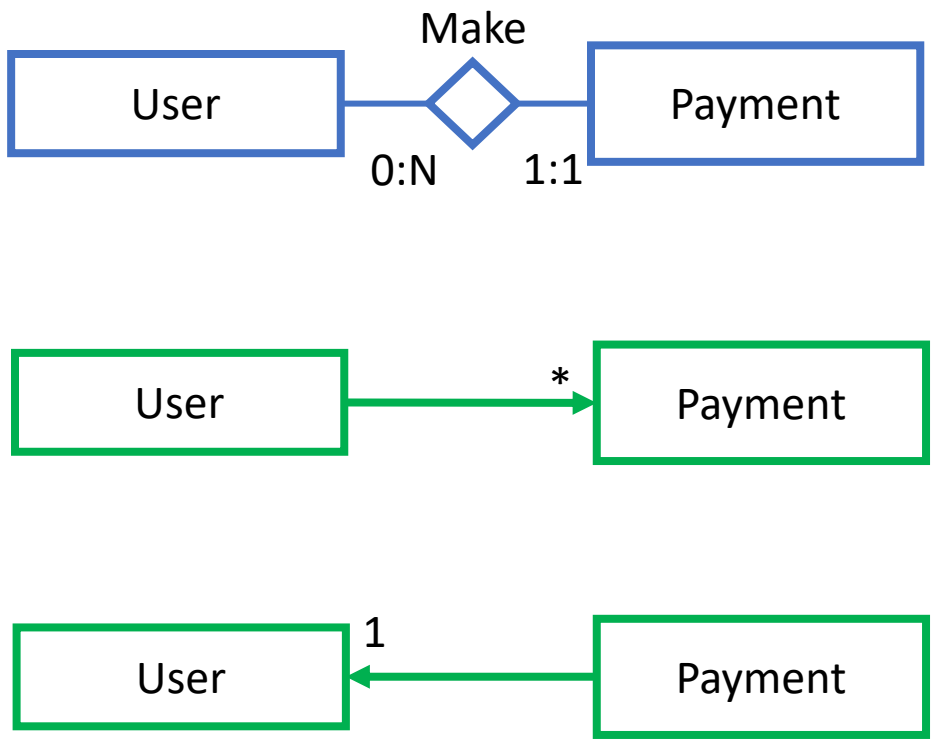


- Order → Payment @OneToMany is necessary to get the payment for a order
  - Fetch type: Lazy
  - Cascade: Persist.
- Payment → Order @ManyToOne is not requested by the specification

Motivation

- Bidirectional OneToMany Order to Payment
  - Owned is the Payment
  - PERSIST are cascaded. Payment are linked to Order, so modifications of Order impact also them.
  - FetchType LAZY because we don’t access to Payment whenever we access Order

Relationship " Make "

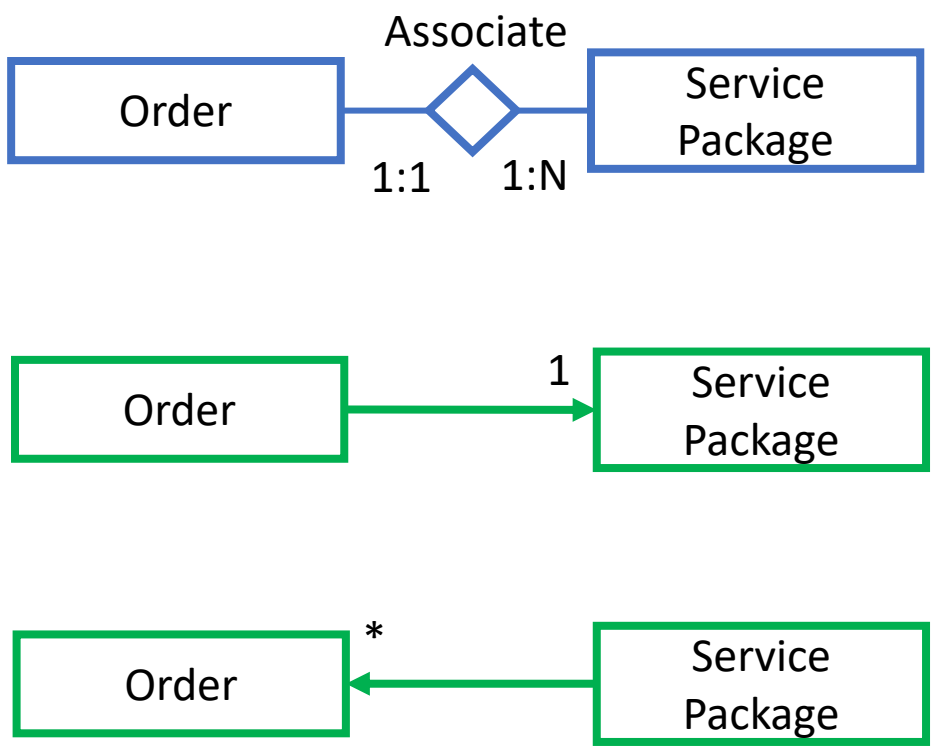


- User→ Payment @OneToManyis not requested by the specification
  - Fetch:LAZY
- Payment → User @ManyToOne is required to access the user that makes the payment for a order

Motivation

- Bidirectional OneToMany Used to Payment
  - Owner is the Payment
  - We implemented this relation between User and Payment because we need a reference of the user in payment to set it as "INSOLVENT" when a payment are rejected.

Relationship "Associate"

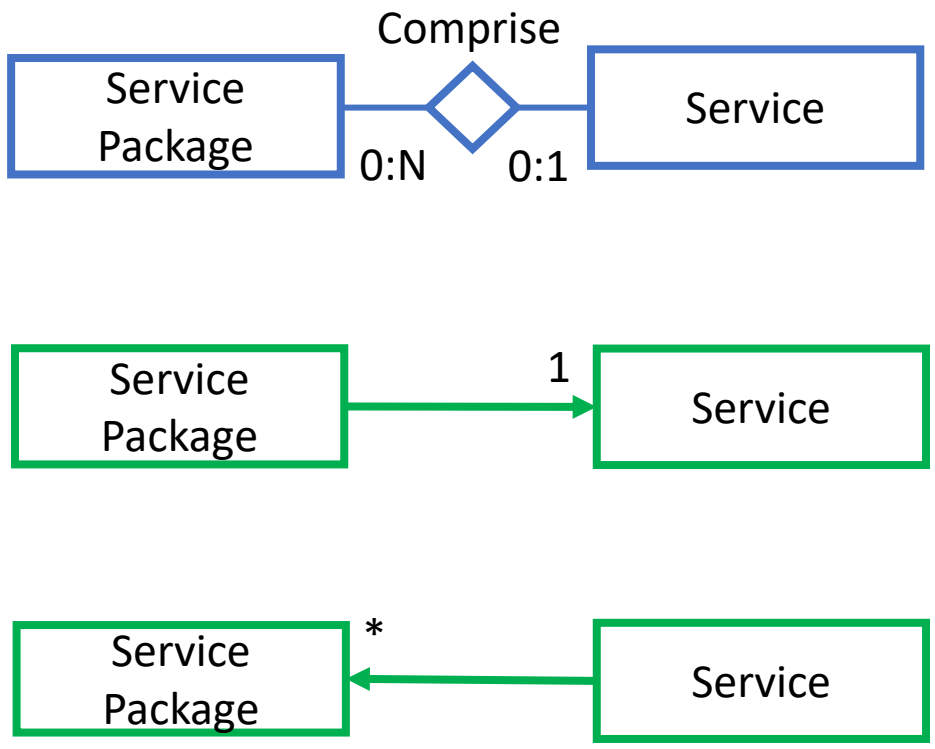


- Order → Service Package @ManyToOne is necessary to get the purchased Service Package
- Service Package → Order @OneToMany is used to synchronize the modification of Service Package to the linked Order
  - Fetch type: EAGER

Motivation

- Bidirectional ManyToOne Order to Service Package
  - **Owner** is Order
- Bidirection OneToMany Service Package to Order
  - Cascade PERSIST is not necessary because creation of a order doesn't impact ServicePackages and viceversa.
  - EAGER fetch type is used because is necessary to load all the orders of a ServicePackage

Relationship "Comprise"



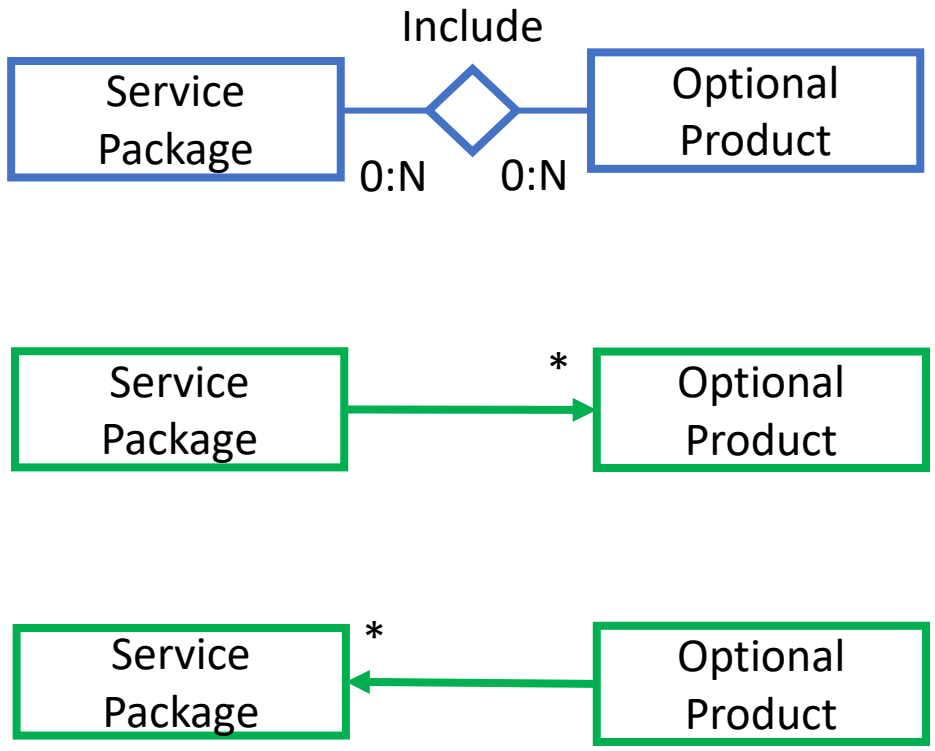
- Service Package → Service
  @ManyToOne is necessary to get the service of a Service Package
  - Fetch type: EAGER
- Service → Service Package
  @OneToMany is not necessary

We implemented this relationship for each Services (FixedPhone, MobilePhone, FixedInternet, MobileInternet)

Motivation

- Bidirectional ManyToOne Service Package to Service
  - **Owner** is ServicePackage
  - FetchType EAGER because we have to load all the informations about Services

Relationship "Include"

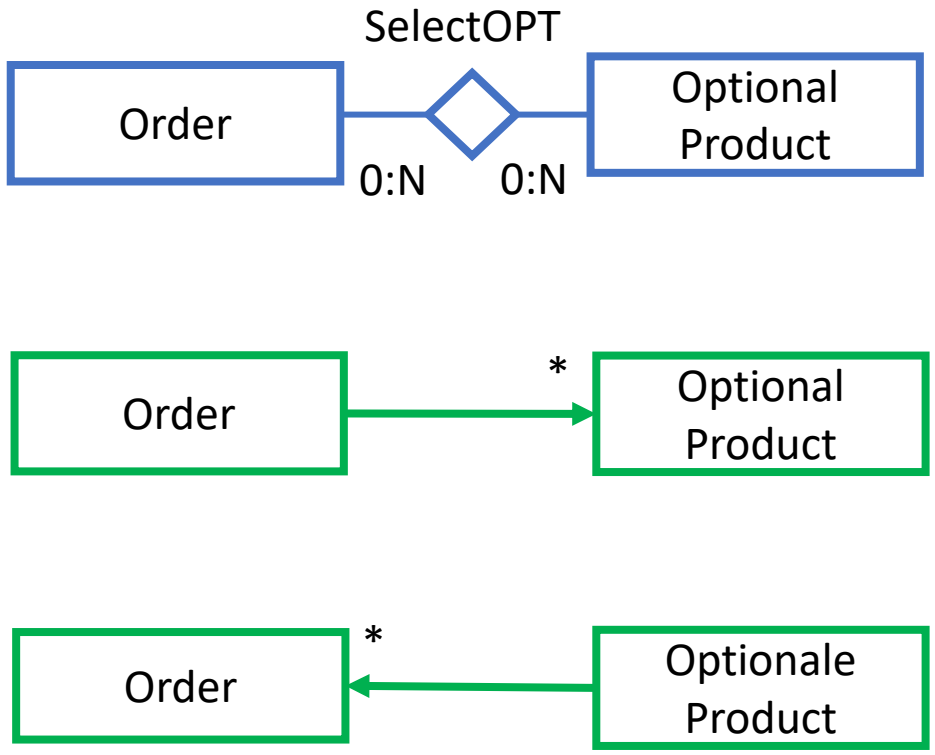


- Service Package → Optional Product
  @ManyToMany is necessary to get the optional product of a Service Package
  - Fetch type: EAGER
- Optional Product → Service Package
  @ManyToMany is not necessary for the specification

Motivation

- Bidirectional ManyToMany Optional product to Service
  - FetchType EAGER because we have to show included services in Service Package.

Relationship SelectOPT

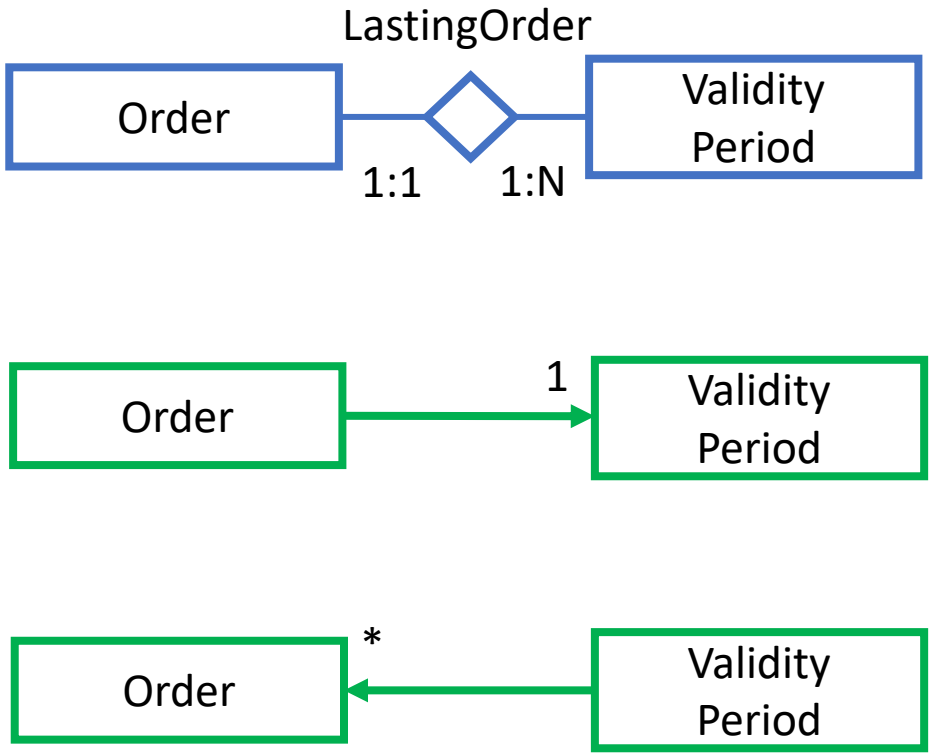


- Order → Optional Product
  @ManyToMany is necessary to get the optional product of a Order
  - Fetch type: EAGER
- Optional Product → Order
  @ManyToMany is not necessary for the specification

Motivation

- Bidirectional ManyToMany Order to Optional Product
  - FetchType EAGER because we have to show included services in Service Package.

Relationship "LastingOrder"

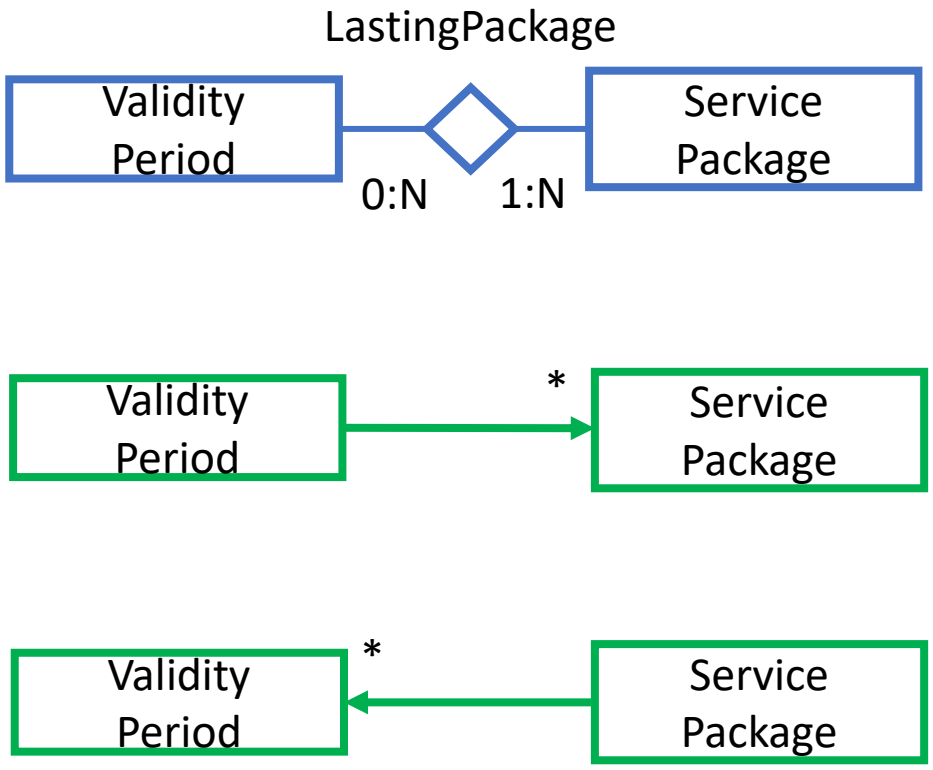


- Order → Validity Period @ManyToOne is necessary to get the relative Validity Period
  - Fetch type: EAGER
- Validity Period → Order @OneToMany is not necessary

Motivation

- Bidirectional OneToMany Order to Validity Period
  - FetchType EAGER because we have to show the Validity Period of the Order

Relationship "LastingPackage"



- Validity Period → Service Package @ManyToMany is not necessary to get the relative Validity Period
- Service Package → Validity Period @ManyToMany is necessary to chose the relative Validity Period
  - Fetch type: EAGER

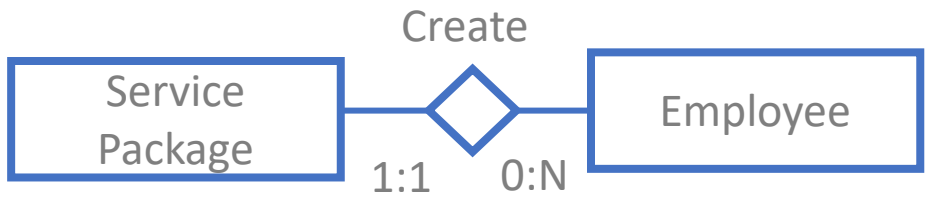
Motivation

- Bidirectional ManyToMany Service Package to Validity Period
  - FetchType EAGER because we have to show the Validity Period of the Service Package

Notes

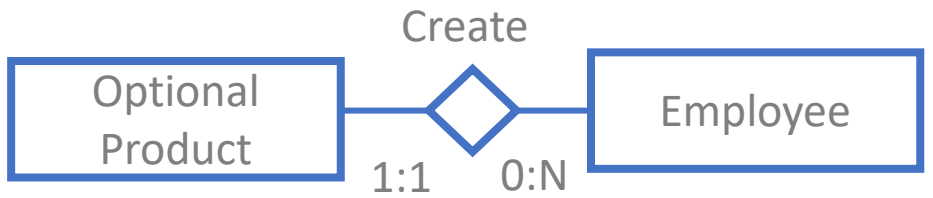
- We have implemented all the unnecessary relationship for completeness

Relationship "Create Package"



This relationship is not implemented beacuse is not necessary for the specification

Relationship "Create Product"



This relationship is not implemented beacuse is not necessary for the specification

Motivation

These relationships are not relevant for the project. We used these relation to link the Employee table with the rest of the database.

ENTITY Design

User

```
@Entity
@Table(name = "User", schema = "db_telco")
@NamedQuery(name = "User.checkCredentials", query = "SELECT r FROM User r  WHERE r.username = ?1 and r.password = ?2")
public class User {
   private static final long serialVersionUID = 1L;

   @Id
   private String username;

   private String email;

   private String password;

   private String status;

   @OneToMany(mappedBy="user_ID", fetch = FetchType.LAZY)
   private List<ActivationSchedule> scheduleList;

   @OneToMany(mappedBy="user_ID", fetch = FetchType.LAZY)
   private List<Payment> paymentList;

   @OneToMany(mappedBy="user_ID", fetch = FetchType.EAGER)
   private List<Order> orderList;

}
```

Validity Period

```
@Entity
@Table(name="ValidityPeriod", schema = "db_telco")
@NamedQuery(name="ValidityPeriod.findAll", query="SELECT v FROM ValidityPeriod v")
public class ValidityPeriod {
private static final long serialVersionUID = 1L;

@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
private int ID;

private int NumMonths;

private BigDecimal MonthlyFee;

@ManyToMany(mappedBy="valPeriodList")
private List<ServicePackage> servPackList;

@OneToMany(mappedBy="ValPeriod_ID")
private List<Order> OrderList;
}
```

ServicePackage

```
@Entity
@Table(name = "ServicePackage", schema = "db_telco")
@NamedQuery(name="ServicePackage.findAll", query="SELECT s FROM ServicePackage s")
public class ServicePackage {
   private static final long serialVersionUID = 1L;

   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private int ID;

   private String Name;

   @OneToMany(mappedBy="ServPack_ID", fetch= FetchType.EAGER)
   private List<Order> orderList;

   @ManyToOne()
   @JoinColumn(name="FP_ID")
   private FixedPhone FP_ID;

   @ManyToOne()
   @JoinColumn(name="MP_ID")
   private MobilePhone MP_ID;

   @ManyToOne()
   @JoinColumn(name="FI_ID")
   private FixedInternet FI_ID;
```

```
    @ManyToOne()
    @JoinColumn(name="MI_ID")
    private MobileInternet MI_ID;

    @ManyToMany(mappedBy="servPackList", fetch
= FetchType.EAGER, cascade = CascadeType.PERSIST) //qui giusto
    private List<OptionalProduct> optProdList;

    @ManyToMany(fetch=FetchType.EAGER) //rimossi i cascade
    @JoinTable(
            name="LastingPackage",
            joinColumns = @JoinColumn(name = "ServPack_ID"),
            inverseJoinColumns = @JoinColumn(name="ValPeriod_ID"))
    private List<ValidityPeriod> valPeriodList;
    optProd.removeServPack(this);
    }
}
```

Payment

```
@Entity
@Table(name = "Payment", schema = "db_telco")
public class Payment {
   private static final long serialVersionUID = 1L;

   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private int ID;

   @Temporal(TemporalType.TIMESTAMP)
   private Date DatePayment;

   private String Status;

   @ManyToOne
   @JoinColumn(name="user_ID")
   private User user_ID;

   @ManyToOne
   @JoinColumn(name="Order_ID")
   private Order Order_ID;
```

```
    public Payment() {};

    public Payment(Order order) {
        this.Order_ID=order;
        this.user_ID=order.getUser_ID();

        this.DatePayment= new Date();//initialized with the current date
        Random rand = new Random();
        int result= rand.nextInt(2);
        if(result==1) {
            Status="ACCEPTED";
            order.setStatus(true);
        }
        else
        {
            Status="REJECTED";
            order.setStatus(false);
        }
        order.addPayment(this);
    }
}
```

Activation Schedule

```
@Entity
@Table(name="ValidityPeriod", schema = "db_telco")
@NamedQuery(name="ValidityPeriod.findAll", query="SELECT v FROM ValidityPeriod v")
public class ValidityPeriod {
   private static final long serialVersionUID = 1L;

   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private int ID;

   private int NumMonths;

   private BigDecimal MonthlyFee;

   @ManyToMany(mappedBy="valPeriodList", fetch = FetchType.EAGER)
   private List<ServicePackage> servPackList;

   @OneToMany(mappedBy="ValPeriod_ID", fetch = FetchType.EAGER)
   private List<Order> OrderList;
}
```

Employee

```
@Entity
@Table(name = "Employee", schema = "db_telco")
@NamedQuery(name = "Employee.checkCredentials", query = "SELECT e FROM Employee
e  WHERE e.ID = ?1 and e.Password = ?2")
public class Employee {
   private static final long serialVersionUID = 1L;

   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private int ID;

   private String Password;
}
```

Fixed Internet

```
@Entity
@Table(name = "FixedInternet", schema = "db_telco")
@NamedQuery(name="FixedInternet.findAll", query="SELECT fi FROM FixedInternet fi")
public class FixedInternet {
   private static final long serialVersionUID = 1L;

   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private int ID;

   private int NumGiga;

   private int FeeGiga;

   @OneToMany(mappedBy="FI_ID", fetch=FetchType.EAGER)
   private List<ServicePackage> servPackList;

   public void addServPack(ServicePackage servPack) {
      getServPackList().add(servPack);
   }

   public void removeServPack(ServicePackage servPack) {
      getServPackList().remove(servPack);
   }
}
```

Fixed Phone

```
@Entity
@Table(name = "FixedPhone", schema = "db_telco")
@NamedQuery(name="FixedPhone.findAll", query="SELECT fp FROM FixedPhone fp")
public class FixedPhone{
   private static final long serialVersionUID = 1L;

   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private int ID;

   @OneToMany(mappedBy="FP_ID", fetch=FetchType.EAGER)
   private List<ServicePackage> servPackList;
}
```

Mobile Internet

```
@Entity
@Table(name = "MobileInternet", schema = "db_telco")
@NamedQuery(name="MobileInternet.findAll", query="SELECT mi FROM MobileInternet mi")
public class MobileInternet {
   private static final long serialVersionUID = 1L;

   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private int ID;

   private int NumGiga;

   private int FeeGiga;

   @OneToMany(mappedBy="MI_ID", fetch=FetchType.EAGER)
   private List<ServicePackage> servPackList;
}
```

Mobile Phone

```
@Entity
@Table(name = "MobilePhone", schema = "db_telco")
@NamedQuery(name="MobilePhone.findAll", query="SELECT mp FROM MobilePhone mp")
public class MobilePhone {
   private static final long serialVersionUID = 1L;

   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private int ID;

   private int NumMinutes;

   private int NumSMS;

   private int FeeMinutes;

   private int FeeSMS;

   @OneToMany(mappedBy="MP_ID", fetch=FetchType.EAGER)
   private List<ServicePackage> servPackList;
}
```

OptionalProduct

```
@Entity
@Table(name = "OptionalProduct", schema = "db_telco")
@NamedQuery(name="OptionalProduct.findAll", query="SELECT o FROM OptionalProduct o")
public class OptionalProduct {
   private static final long serialVersionUID = 1L;

   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private int ID;

   private String Name;

   private BigDecimal MonthlyFee;

   @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.PERSIST)
   @JoinTable(name="Include",
            joinColumns= @JoinColumn(name="OptProd_ID"),
            inverseJoinColumns= @JoinColumn(name="ServPack_ID"))
   private List<ServicePackage> servPackList;

   @ManyToMany(fetch = FetchType.EAGER)
   @JoinTable(name="selectopt",
            joinColumns= @JoinColumn(name="OptProd_ID"),
            inverseJoinColumns= @JoinColumn(name="Order_ID"))
   private List<Order> orderList;
}
```

Order

```
@Entity
@Table(name = "Order", schema = "db_telco")
@NamedQuery(name = "Order.rejectedOrdersFromUserID", query = "SELECT o FROM Order o JOIN o.user_ID u WHERE locate(:user_ID,
u.username)>0 AND o.Status = \"REJECTED\"")
public class Order {
   private static final long serialVersionUID = 1L;

   @Id
   @GeneratedValue(strategy=GenerationType.IDENTITY)
   private int ID;

   @Temporal(TemporalType.TIMESTAMP)
   private Date DataOrd;

   @Temporal(TemporalType.TIMESTAMP)
   private Date StartOrd;

   private BigDecimal TotalValue;

   private String Status;

   ...
```

```
 @ManyToOne
   @JoinColumn(name="user_ID")
   private User user_ID;

   @ManyToOne
   @JoinColumn(name="ServPack_ID")
   private ServicePackage ServPack_ID;

   @ManyToOne
   @JoinColumn(name="ValPeriod_ID")
   private ValidityPeriod ValPeriod_ID;



   @OneToOne(mappedBy="Order_ID", fetch = FetchType.LAZY)
   private ActivationSchedule schedule;



   @OneToMany(mappedBy="Order_ID", fetch = FetchType.LAZY, cascade = CascadeType.PERSIST)
   private List<Payment> payment;

   @ManyToMany(mappedBy="orderList", fetch = FetchType.EAGER) //CASCADE pericolosa
   private List<OptionalProduct> optProdList;
}
```

TRIGGER Design

General View

We decided to use triggers to implements the Activation scheduler and the Sales Report. The latter is divided in 3 materialized view (implemented in MySQL as table) based on the possibility of data aggregation:

*SalesReport1 (**ServPack_ID,** NumSales_SP, NumSales_W, NumSales_WO, AvgSales_OPT, NumSales_OPT)*

*SalesReport2 (**ServPack_ID**, **ValPeriod_ID**, NumSales)*

*SalesReport3 (**OPTprod_ID,** NumSales)*

The first report table is managed using multiple triggers, we took advantage of trigger execution ordering to make developing of them less complex. The other tables are managed by 2 triggers: one activated by the insertion of a completed order, the other by the completion of an unpaid order (Multiple conditions trigger are not supported in MySQL).

Other tables are present for: list of Insolvent users, suspended orders and alert

Sales Report 1 - 1A

SalesReport1_After_Insert

Event: Insertion of a tuple in Order

Condition: Order.Status = " ACCEPTED"

Action: if the ID of the servicePack selected in the order is not present in salesReport1, add a tuple with ServicePack_ID as key and initialize it. Else update the total number of sales of the selected ServicePack

This pair of triggers is the first to be executed, this guarantees that in the table there is always a tuple with the Id of the chosen ServicePack

```
CREATE TRIGGER `SalesReport1_AFTER_INSERT`
AFTER INSERT  ON `Order`
FOR EACH ROW
BEGIN
   if exists (select *
         from `SalesReport1`
         where `SalesReport1`.ServPack_ID = new.ServPack_ID and new.Status = "ACCEPTED")
         then
   begin
    update `SalesReport1` SET
    NumSales_SP = NumSales_SP + 1
    where `SalesReport1`.ServPack_ID = new.ServPack_ID;
   end;
 else
 if exists (select *
         from `db_telco`.`Order`
         where `db_telco`.`Order`.ID = new.ID and  new.Status = "ACCEPTED") then
   begin
    insert into `SalesReport1`(ServPack_ID, NumSales_SP, NumSales_W,NumSales_WO,AvgSales_OPT,NumSales_OPT)
         values (new.ServPack_ID, 1, 0, 0, 0, 0);
      end;
      END IF;
      end IF;
```

Sales Report 1 - 1B

SalesReport1_After_Update

Event: Update Status of a Order Tuple

Condition: Order.Status = " ACCEPTED"

Action: if the ID of the servicePack selected in the order is not present in salesReport1, add a tuple with ServicePack_ID as key and initialize it. Else update the total number of sales of the selected ServicePack

This pair of triggers is the first to be executed, this guarantees that in the table there is always a tuple with the Id of the chosen ServicePack

```
CREATE TRIGGER `SalesReport1_AFTER_UPDATE`
AFTER UPDATE ON `Order`
FOR EACH ROW
BEGIN
if exists (select *
            from `SalesReport1`
            where `SalesReport1`.ServPack_ID = new.ServPack_ID and new.Status = "ACCEPTED") then
   begin
    update `SalesReport1` SET
          NumSales_SP = NumSales_SP + 1
           where `SalesReport1`.ServPack_ID = new.ServPack_ID;
   end;
  else
  if exists (select *
     from `db_telco`.`Order`
     where `db_telco`.`Order`.ID = new.ID and  new.Status = "ACCEPTED") then
   begin
     insert into `SalesReport1`(ServPack_ID, NumSales_SP, NumSales_W,NumSales_WO,AvgSales_OPT,NumSales_OPT)
     values (new.ServPack_ID, 1, 0, 0, 0, 0);
    end;
  END IF;
  END IF
END
```

Sales Report 1- 2A

SalesReport1_AverangeOPT_After_Insert

Event: Insertion of a tuple in SelectOPT

Condition: Status of Linked Order equal to "ACCEPTED"

Action: For each selected Optional Product update the column of the linked ServicePack

- *NumSales_OPT = NumSales_OPT + 1*
- *AvgSales_OPT = NumSales_OPT / NumSales_SP*

The trigger is not activated on a new Order entry because the optional services have not yet been loaded into the database. Using a FOR EACH STATEMENT trigger would have been more appropriate but it is not supported in MySQL

```
CREATE TRIGGER `SalesReport1_AverangeOPT_AFTER_INSERT`
AFTER INSERT ON `SelectOpt`
FOR EACH ROW
BEGIN
#Order must exist -> previus trigger load it in the table
#select opt is load after  Order so i can't trigger on on OrderINSERT,
#also for each statement is not supported in MySQL
IF exists( select *
from `Order` join `SelectOpt` on `Order`.ID = `SelectOpt`.Order_ID
          where `Order`.Status="ACCEPTED" and `Order`.ID = new.Order_ID)
then
      BEGIN
       update `SalesReport1` SET
        NumSales_OPT = numSales_OPT + 1,
        AvgSales_OPT = numSales_OPT / NumSales_SP
        where `SalesReport1`.ServPack_ID = (select ServPack_ID
                                            from `Order`
                                            where `Order`.ID=new.Order_ID);
      END;
      END IF;
END
```

Sales Report 1- 2B

SalesReport1_AverangeOPT_After_Update

Event: Update Status of a Order Tuple

Condition: Order.Status = "ACCEPTED"

Action: Count the total number of selected Optional product and update the columns:

- *NumSales_OPT = NumSales_OPT + count(SelectedOPT)*
- *AvgSales_OPT = NumSales_OPT / NumSales_SP*

```
CREATE TRIGGER `SalesReport1_AverangeOPT_AFTER_UPDATE`
AFTER UPDATE  ON `Order`
FOR EACH ROW
BEGIN
#Order must exist -> previus trigger load it in the table
    update `SalesReport1` SET
    NumSales_OPT = numSales_OPT + (select count(*)
                                   from `Order` join `SelectOpt` on `Order`.ID= `SelectOpt`.Order_ID
                                   where new.ID=`SelectOpt`.Order_ID),
AvgSales_OPT = numSales_OPT / NumSales_SP
where `SalesReport1`.ServPack_ID = new.ServPack_ID;
END
```

Sales Report 1 – 3A

SalesReport1_SalesOPTW_AFTER_INSERT

Event: insertion of a Order tuple
Condition: Order.Status = " ACCEPTED"
Action: for the Service Package purchased with the order the trigger increments the values of sales with the Optional products, in the column *NumSales_W*, and without them, in the column *NumSales_WO*.

```
CREATE TRIGGER `SalesReport1_SalesOPTW_AFTER_INSERT`
AFTER INSERT ON `Order`
FOR EACH ROW
BEGIN
   IF exists(    select *
            from `Order`
            where `Order`.Status="ACCEPTED" and `Order`.ID = new.ID) then
      BEGIN
         update `SalesReport1`
            SET NumSales_W = NumSales_W + new.TotalValue,
               NumSales_WO = NumSales_WO + (select `ValidityPeriod`.NumMonths*`ValidityPeriod`.MonthlyFee
                                    from `Order` join `ValidityPeriod` on `Order`.ValPeriod_ID=`ValidityPeriod`.ID
                                    where `Order`.ID=new.ID)
         where `SalesReport1`.ServPack_ID = new.ServPack_ID;
      END;
   END IF;
END
```

Sales Report 1 – 3B

SalesReport1_SalesOPTW_AFTER_UPDATE

Event: update of a Order tuple
Condition: the updated tuple has *Status* = "ACCEPTED"
Action: for the Service Package purchased with the order the trigger increments the values of sales with the Optional products, in the column *NumSales_W*, and without them, in the column *NumSales_WO*.

```
CREATE TRIGGER `SalesReport1_SalesOPTW_AFTER_UPDATE`
AFTER UPDATE ON `Order`
FOR EACH ROW
BEGIN
   IF exists (    select *
                  from `Order`
                  where `Order`.ID=new.ID and new.Status="ACCEPTED") then
      BEGIN
         update `SalesReport1`
            SET NumSales_W = NumSales_W + new.TotalValue,
                NumSales_WO = NumSales_WO + (select `ValidityPeriod`.NumMonths * `ValidityPeriod`.MonthlyFee
                                                    from `Order` join `ValidityPeriod`
                                                          on `Order`.ValPeriod_ID=`ValidityPeriod`.ID
                                                    where `Order`.ID=new.ID)
         where `SalesReport1`.ServPack_ID = new.ServPack_ID;
      END;
   END IF;
END
```

ActivationSchedule - A

ActivationSchedule_AFTER_INSERT

Event: Insertion of a tuple in Order

Condition: Order.Status = "ACCEPTED"

Action: Calculate the Deactivation Date of The Services linked in the order and Add a new tuple in ActivationSchedule table

```
CREATE TRIGGER `ActivationSchedule_AFTER_INSERT`
AFTER INSERT  ON `Order`
FOR EACH ROW
BEGIN

IF(new.Status="ACCEPTED") THEN
BEGIN
   INSERT INTO `db_telco`.`ActivationSchedule`
   (`User_ID`, `Order_ID`, `ActivationDate`, `DeactivationDate`)
   VALUES
   (new.User_ID,new.ID,new.StartOrd,  (SELECT DATE_ADD(new.StartOrd,INTERVAL (select NumMonths
                                                from `ValidityPeriod` join `Order` on `ValidityPeriod`.ID = `Order`.ValPeriod_ID
                                                where `Order`.ID=new.ID
                                            ) MONTH)));
END;
END IF;
```

ActivationSchedule - B

ActivationSchedule_AFTER_UPDATE

Event: Update Status of a Order Tuple

Condition: Order.Status = "ACCEPTED"

Action: Calculate the Deactivation Date of The Services linked in the order and Add a new tuple in ActivationSchedule table

```
CREATE TRIGGER `ActivationSchedule_AFTER_UPDATE`
AFTER UPDATE  ON `Order`
FOR EACH ROW
BEGIN

IF(new.Status="ACCEPTED") THEN
BEGIN
   INSERT INTO `db_telco`.`ActivationSchedule`
   (`User_ID`, `Order_ID`, `ActivationDate`, `DeactivationDate`)
   VALUES
   (new.User_ID,new.ID,new.StartOrd,  (SELECT DATE_ADD(new.StartOrd,INTERVAL (select NumMonths
                                                from `ValidityPeriod` join `Order` on `ValidityPeriod`.ID = `Order`.ValPeriod_ID
                                                where `Order`.ID=new.ID
                                            ) MONTH)));
END;
END IF;
END
```

Sales Report 2 – 1A

salesreport2_AFTER_INSERT

Event: insertion of a Order tuple
Condition: the inserted tuple has *Status* = "ACCEPTED"
Action: if the SalesReport2 table has already a tuple with the Service Package purchased in the current Order, the trigger increments of 1 the values of the column *NumSales*, which indicates the number of total purchases per package and validity period; otherwise, it inserts a new tuple in the table initializing it with the Service Package ID and the Validity Period ID listed in the order and the value of 1 in the *NumSales* column.

```
CREATE TRIGGER `salesreport2_AFTER_INSERT`
AFTER INSERT ON `Order`
FOR EACH ROW
BEGIN
if exists (select *
        from `SalesReport2`
        where `SalesReport2`.ServPack_ID = new.ServPack_ID and `SalesReport2`.ValPeriod_ID = new.ValPeriod_ID and new.Status = "ACCEPTED")
        then
   begin
      update `SalesReport2` SET
      NumSales = NumSales + 1
      where `SalesReport2`.ServPack_ID = new.ServPack_ID and `SalesReport2`.ValPeriod_ID = new.ValPeriod_ID;
   end;
 else
 if exists (select *
            from `db_telco`.`Order`
            where `db_telco`.`Order`.ID = new.ID and  new.Status = "ACCEPTED") then
   begin
     insert into `SalesReport2`(ServPack_ID, ValPeriod_ID, NumSales)
        values (new.ServPack_ID, new.ValPeriod_ID, 1);
      end;
  END IF;
 end IF;
END
```

Sales Report 2 – 1B

salesreport2_AFTER_UPDATE

Event: update of a Order tuple
Condition: *Status* = "ACCEPTED"
Action: if the SalesReport2 table has already a tuple with the Service Package purchased in the current Order, the trigger increments of 1 the values of the column *NumSales*, which indicates the number of total purchases per package and validity period; otherwise, it inserts a new tuple in the table initializing it with the Service Package ID and the Validity Period ID listed in the order and the value of 1 in the *NumSales* column.

```
CREATE TRIGGER `salesreport2_AFTER_UPDATE`
AFTER UPDATE ON `Order`
FOR EACH ROW
BEGIN
if exists (select *
            from `SalesReport2`
            where `SalesReport2`.ServPack_ID = new.ServPack_ID  and `SalesReport2`.ValPeriod_ID = new.ValPeriod_ID and new.Status = "ACCEPTED")
          then
   begin
      update `SalesReport2` SET
      NumSales = NumSales + 1
      where `SalesReport2`.ServPack_ID = new.ServPack_ID and `SalesReport2`.ValPeriod_ID = new.ValPeriod_ID;
     END;
 else
 if exists (select *
             from `db_telco`.`Order`
             where `db_telco`.`Order`.ID = new.ID and  new.Status = "ACCEPTED") then
   begin
     insert into `SalesReport2`(ServPack_ID, ValPeriod_ID, NumSales)
          values (new.ServPack_ID, new.ValPeriod_ID, 1);
      end;
  END IF;
 END IF;
END
```

Sales Report 3- 1A

SalesReport3_After_Insert

Event: Insertion of a tuple in SelectOPT

Condition: Status of Linked Order equal to "ACCEPTED"

Action: For each new tuple present in the SelectedOPT table search for a correspondent tuple in SalesReport3: if present update the value of the sales using the MonthyFree of the optional product and the duration of the correspondent order plus the old value of the sales, otherwise create a tuple with the correspondent Optional Product ID and initialize the sales in the same way.

The trigger is not activated on a new Order entry because the optional services have not yet been loaded into the database

```
CREATE TRIGGER `salesreport3_AFTER_INSERT`
AFTER INSERT ON `SelectOpt`
FOR EACH ROW
BEGIN

if((select `Status` from `Order` where `Order`.ID= new.Order_ID) = "ACCEPTED") then #order Accepted
BEGIN
    if exists (select *
    from `SalesReport3`
    where `SalesReport3`.OptProd_ID = new.OptProd_ID) then
        BEGIN #element is already present
        update `SalesReport3` as T  SET
        T.NumSales= T.NumSales + (select NumMonths
                                  from  `Order` join `ValidityPeriod` on `Order`.ValPeriod_ID = `ValidityPeriod`.ID
                                  where `Order`.ID = new.Order_ID)
                                  *
                                  (select MonthlyFee
                                  from `OptionalProduct`
                                  where new.OptProd_ID = `OptionalProduct`.ID)
                                  where T.OptProd_ID = new.OptProd_ID;
END;

        ELSE #element is new
        BEGIN
            insert into `SalesReport3`(OptProd_ID,NumSales)
            value (new.OptProd_ID,(select NumMonths
                                    from `Order` join `ValidityPeriod` on `Order`.ValPeriod_ID = `ValidityPeriod`.ID
                                    where `Order`.ID = new.Order_ID)
                                     *
                                    (select MonthlyFee
                                    from `OptionalProduct`
                                    where new.OptProd_ID = `OptionalProduct`.ID));
END;

END IF;
END;
END IF;
END
```

Sales Report 3- 1B

SalesReport3_After_Update

Event: Update Status of a Order Tuple

Condition: Order.Status = "ACCEPTED"

Action: For each tuple present in the SelectedOPT table with the orderID equal to the current Order ID search for a correspondent tuple in SalesReport3: if present update the value of the sales using the MonthyFree of the optional product and the duration of the correspondent order plus the old value of the sales, otherwise create a tuple with the correspondent Optional Product ID and initialize the sales in the same way.

```
CREATE TRIGGER `salesreport3_AFTER_UPDATE`
AFTER UPDATE ON `Order`
FOR EACH ROW
BEGIN

if(new.Status = "ACCEPTED") then #order Accepted
BEGIN #ORDER ACCEPTED

   DECLARE var1 INT DEFAULT 0;
   DECLARE var2 INT DEFAULT 0;
   SELECT COUNT(*) FROM (select * from `order` join `selectOpt` on `order`.ID = `selectOPT`.Order_ID where `order`.ID = new.ID) as t INTO var1;
   SET var2 =0;
   WHILE var2 < var1 DO
      if exists(select *
            from `salesreport3`
            where `salesreport3`.OptProd_ID = (select OptProd_ID
                              from `order` join `selectOpt` on `order`.ID = `selectOPT`.Order_ID
                              where `order`.ID = new.ID LIMIT var2,1))
            then
            BEGIN #opt already in `salesreport3`
                  update `SalesReport3` SET
                     NumSales= NumSales + ((select NumMonths
                              from  `validityperiod` join  `Order` on `Order`.ValPeriod_ID= `validityperiod`.ID
                              where `Order`.ID = new.ID)
                           *
                              (select MonthlyFee
                                 from `OptionalProduct`
                                 where `OptionalProduct`.ID = (select OptProd_ID
                                                from `order` join `selectOpt` on `order`.ID = `selectOPT`.Order_ID
                                                where `order`.ID = new.ID LIMIT var2,1)))

                  where OptProd_ID = (select OptProd_ID
                              from `order` join `selectOpt` on `order`.ID = `selectOPT`.Order_ID
                              where `order`.ID = new.ID LIMIT var2,1);
            END;
```

```
        ELSE

            BEGIN #ELEMENT IS NEW
                insert into `SalesReport3`(OptProd_ID,NumSales)
                value (
                    (select OptProd_ID
                    from `order` join `selectOpt` on `order`.ID = `selectOPT`.Order_ID
                    where `order`.ID = new.ID LIMIT var2,1)
                    ,
                    ((select NumMonths
                    from  `validityperiod` join  `Order` on `Order`.ValPeriod_ID= `validityperiod`.ID
                    where `Order`.ID = new.ID)
                    *
                    (select MonthlyFee
                    from `OptionalProduct`
                    where `OptionalProduct`.ID = (select OptProd_ID
                                    from `order` join `selectOpt` on `order`.ID = `selectOPT`.Order_ID
                                    where `order`.ID = new.ID LIMIT var2,1))));
            END;

        END IF;
      SET var2 = var2 + 1;

   END WHILE;
END;
END IF;
END
```

Insolvent Users

InsolventUsers_AFTER_UPDATE

Event: update of a User tuple
Condition: Status="INSOLVENT" or "REGULAR"
Action: when a User's order is rejected by the payment service, he becomes *insolvent* and his username is inserted in the *InsolventUsers* table, if it is not present already. Instead, when a *insolvent* User pays successfully all of his rejected orders, his username is deleted from the *InsolventUsers*.

```
CREATE TRIGGER `InsolventUsers_AFTER_UPDATE`
AFTER UPDATE ON `User`
FOR EACH ROW
BEGIN
   if exists (    select *
                  from `User`
                  where `User`.username = new.username
                     and `User`.status = "INSOLVENT") then
         begin
            insert into `InsolventUsers`(username)
               values(new.username);
         end;
   else
   if exists (    select *
                  from `InsolventUsers`
                  where `InsolventUsers`.username=new.username
                     and new.status = "GOODFELLA") then
         begin
            delete from `InsolventUsers`
               where `InsolventUsers`.username = new.username;
         end;
   END IF;
   END IF;
END
```

Suspended Orders - 1A

SuspendedOrders_AFTER_INSERT

Event: insertion of an Order tuple
Condition: Status = "REJECTED"
Action: when an order is rejected by the payment service it is inserted in the *SuspendedOrders* table.

```
CREATE TRIGGER `SuspendedOrders_AFTER_INSERT`
AFTER INSERT ON `Order`
FOR EACH ROW
BEGIN
    if exists (    select *
                   from `Order`
                   where `Order`.ID = new.ID
                       and new.Status="REJECTED") then
        begin
            insert into `SuspendedOrders`(ID)
                values (new.ID);
        end;
    END IF;
END
```

Suspended Orders - 1B

SuspendedOrders_AFTER_UPDATE

Event: update of an Order tuple
Condition: Status = "ACCEPTED"
Action: when a user pay successfully a rejected order, the latter is deleted from the *SuspendedOrders* table.

```
CREATE TRIGGER `SuspendedOrders_AFTER_UPDATE`
AFTER UPDATE ON `Order`
FOR EACH ROW
BEGIN
    if exists (    select *
                   from `SuspendedOrders`
                   where `SuspendedOrders`.ID = new.ID
                      and new.Status="ACCEPTED") then
        begin
            delete from `SuspendedOrders`
                where `SuspendedOrders`.ID=new.ID;
        end;
    END IF;
END
```

Alert

Alert_AFTER_INSERT

Event: update of a Payment tuple
Condition: when the same user causes three failed payments
Action: a tuple is created in the *Alert* table, with the username, email, and the amount, date and time of the last rejection.

Every three failed payments, a tuple is inserted in the table by means of *modulus* operation

```
CREATE TRIGGER `Alert_AFTER_INSERT`
AFTER INSERT ON `Payment`
FOR EACH ROW
BEGIN
   if ((    select MOD(count(*),3)
         from `Payment`
         where `Payment`.User_ID=new.User_ID
            and `Payment`.Status="REJECTED") = 0) and
      ((    select count(*)
            from `Payment`
            where `Payment`.User_ID=new.User_ID
             and `Payment`.Status="REJECTED") >= 1) and
      exists(    select *
            from `Payment`
            where `Payment`.User_ID=new.User_ID
               and new.Status="REJECTED") then
      begin
         insert into `Alert`(User_Username,
                        User_Email,
                        Order_TotalValue,
                        Payment_Date)
               values(    new.User_ID,
                        (select `User`.email
                        from `User`
                        where `User`.username=new.User_ID),
                        (select `Order`.TotalValue
                        from `Order`
                        where `Order`.ID=new.Order_ID),
                        new.DatePayment);
      end;
   END IF;
END
```

Functional Analysis of the Consumer Interaction

The Web Application allows users to purchase Service Packages. In the Landing Page the user can log in or register. The Home Page can be visited after login and shows the list of available Service Packages and the list of rejected orders. A "log out" button can bring the user to the Landing Page. Near each Service Package there is a "Details" button, that leads to a Service Package details page, in which the user can see all the Service Package's informations. With the "buy" button the user access the Buy Page. On this page there is a form which the user can create a new order by entering all the data which are mandatory. Then the Confirmation Page is opened and the user can confirm the order clicking the "Confirm" button. A new order is created with its relative payment data. The Check Out page shows the order and payment's Status; also a "Home Page" button lets the user to return back to the Home Page. In the Home Page the user can click a "Buy again" button, linked to a rejected order, that brings again to the Confirmation Page of the order, where the user can repeat the payment process.

From the Landing Page an unlogged user can access the Home Page with a "Home Page" button. In the Home Page only the list of available Service Packages and the relative "Details" buttons are showed. The unlogged user can click the "Buy Page" button to go to Buy Page and place an order like a logged user. But, in the Payment page instead of the "Confirm" button the "Login" button is showed, that redirects the user to the Landing page. After logged in or registered the user can confirm in the Payment page clicking on the "Confirm" button, just like the logged user.

- Pages (views), view components, events, actions

Functional Analysis of the Employee application

The Web appication allows employees to create a Service Package or a Optional product. In the Landing Page a employee can log in and access to the Employee Home page, where two forms are present: one to create a new Service Package and another to create a new Optional Product. In the top right of the page there are the "Logout" button, which redirects to the Landing page, and the "SalesReport" button, that leads to the Sales Report page. In this page there are four tables that show all the relevant data for the employee; there is also a "Go to Home page" button to go back to the Employee Home page.

- Pages (views), view components, events, actions

Components

Client components

WebServlet:

- CheckLoginConsumer
- CheckLoginEmployee
- CreateConsumer
- CreateOptionalProduct
- CreateOrder
- CreateServicePackage
- GetRejectedOrder
- GetServicePackageDetails
- OrderCheckOUT
- Logout
- GoToSalesReport
- GoToPay
- GoToLogin
- GoToHomePage
- GoToEmployeeHomePage
- GoToBuyService

HTML Page:

- ServicePackageDetails.htlm
- SalesReport.htlm
- PaymentPage.htlm
- OrderCheckOut.htlm
- HomeEmployee.htlm
- Home.htlm
- BuyPage.htlm
- Index.htlm

Business Components (EJBs)

ValidityPeriodServices

- public List<ValidityPeriod> findAllValPeriod()
- public ValidityPeriod getFromID(int ID)

UserServices

- public User checkCredentials(String usrn, String pwd)
- public void createUser(String name, String email, String password)
- public void updateUserStatus(User user)

ServicesServices

- public List<FixedInternet> findAllFixInt()
- public List<FixedPhone> findAllFixPho()
- public List<MobilePhone> findAllMobPho()
- public List<MobileInternet> findAllMobInt
- public FixedPhone findFPByID(int fP_ID)
- public FixedInternet findFIByID(int fI_ID)
- public MobileInternet findMIByID(int mI_ID)
- public MobilePhone findMPByID(int mP_ID)

- EmployeeServices:
  - public Employee checkCredentials(int usrn, String pwd);
- OptionalProductServices:
  - public OptionalProduct getFromID(int ID);
  - public List<OptionalProduct> findAllOptProd();
  - public void createOptionalProduct(String name_prod, BigDecimal monFee, List<Integer> servPackList_int);
- OrderServices:
  - public List<Order> rejectedOrdersFromUserID(String user_ID);
  - public Order getFromID(int ID);
  - public String loadAndPayOrder (Order order);
- SalesReportServices:
  - public List<SalesReport1> getSalesReport1();
  - public List<SalesReport2> getSalesReport2();
  - public List<SalesReport3> getSalesReport3();
  - public List<InsolventUsers> getInsolventUsers();
  - public List<SuspendedOrders> getSuspendedOrders();
  - public List<Alert> getAlerts();
- ServicePakcageServices:
  - public List<ServicePackage> findAllServPackages();
  - public ServicePackage findServPackByID(int servPackID);
  - public void createServPack (String name, int fP_ID, int mP_ID, int fI_ID, int mI_ID, List<Integer> optProdList_ID, List<Integer> valPeriodList_ID);
  - public void addOptProd(ServicePackage servPack, OptionalProduct optProd);

Motivations of the components design

- The EJB is stateless because business method calls act independently and do not need the preservation of the session state
- All business methods use default transaction behaviour, the transaction is started when the methods is invoked by the (Web) client and terminates when the method execution ends

UML sequence diagrams – Order (logged User)