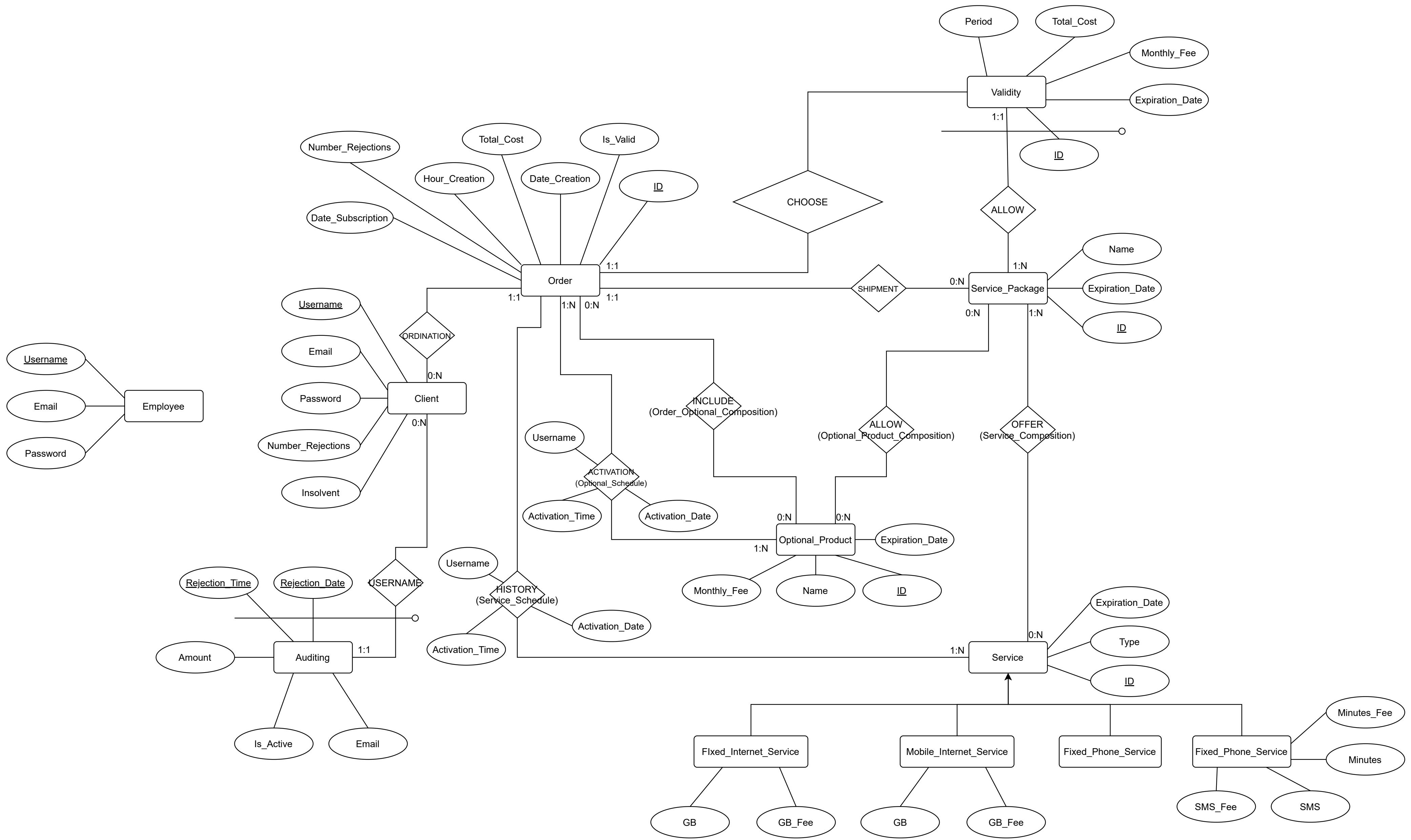
TelcoService

D’Amico (10661846) – Didoni (10623061)

Specification interpretation and assumptions

- The specifications document didn’t specify anything about the employee registration. For this reason we assumed that an employee can be registered only modifing directly the database. Future implementations could introduce an application (still hidden to the employee) to register an employee without accessing the database directly.

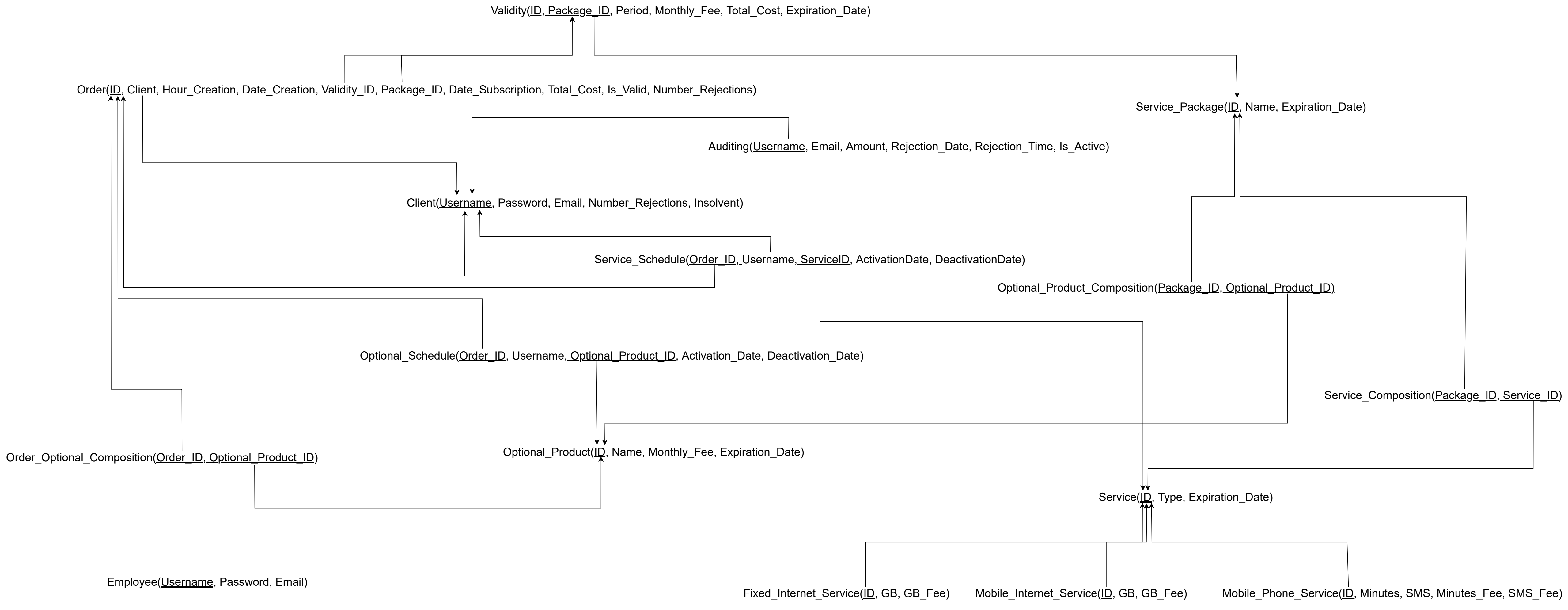
Entity Relationship



Motivation of the ER design

- One of the most critical points of the design was the Validity entity that has been modelled as a weak entity with respect to ServicePackage. The reason for this choice is that a validity, as it is, has little meaning and the monthly fee depends on the ServicePackage, thus relating the Validity with its ServicePackage is necessary.

Relational Model



Validity(ID, Package_ID, Period, Monthly_Fee, Total_Cost, Expiration_Date)

Order(ID, Client, Hour_Creation, Date_Creation, Validity_ID, Package_ID, Date_Subscription, Total_Cost, Is_Valid, Number_Rejections)

Service_Package(ID, Name, Expiration_Date)

Auditing(Username, Email, Amount, Rejection_Date, Rejection_Time, Is_Active)

Client(Username, Password, Email, Number_Rejections, Insolvent)

Service_Schedule(Order_ID, Username, ServiceID, ActivationDate, DeactivationDate)

Optional_Product_Composition(Package_ID, Optional_Product_ID)

Optional_Schedule(Order_ID, Username, Optional_Product_ID, Activation_Date, Deactivation_Date)

Service_Composition(Package_ID, Service_ID)

Order_Optional_Composition(Order_ID, Optional_Product_ID)

Optional_Product(ID, Name, Monthly_Fee, Expiration_Date)

Service(ID, Type, Expiration_Date)

Employee(Username, Password, Email)

Fixed_Internet_Service(ID, GB, GB_Fee)

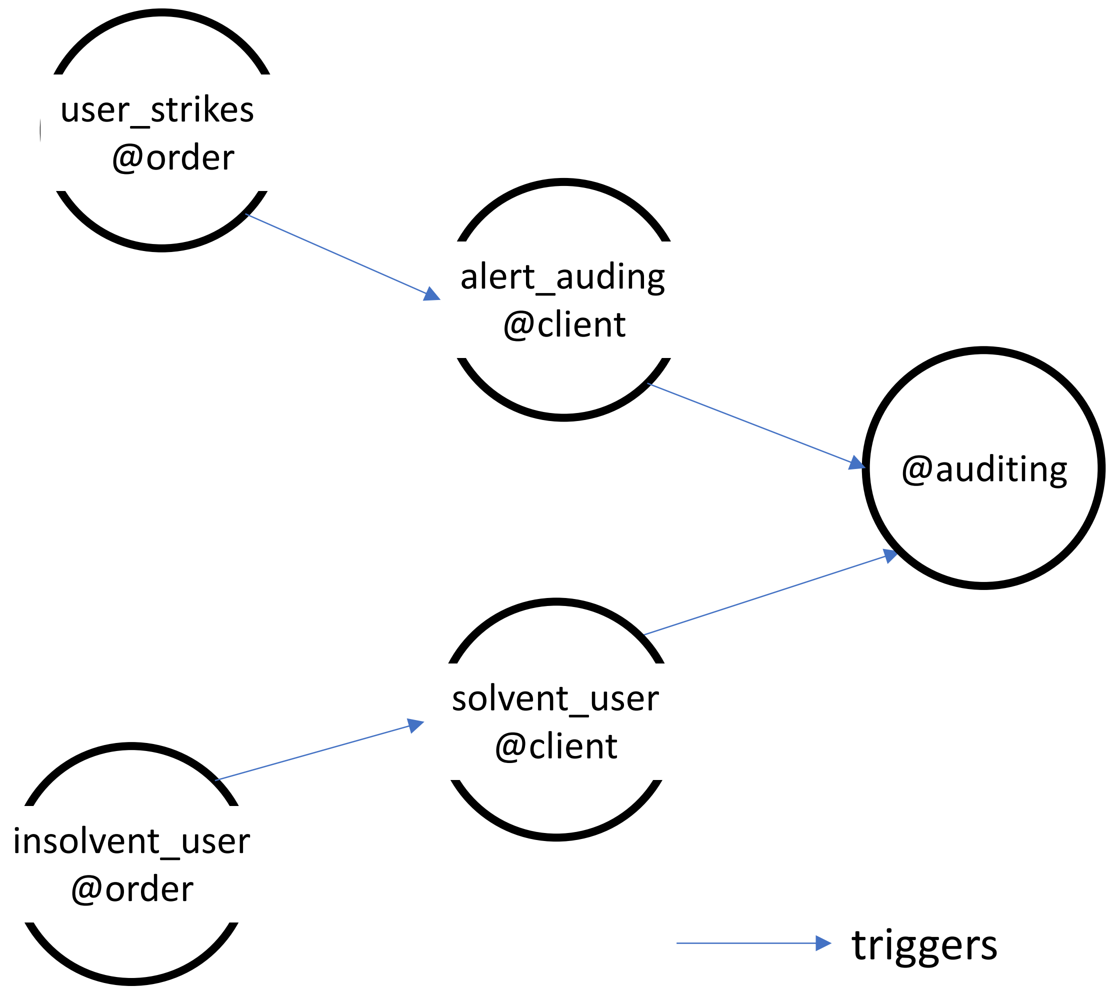
Mobile_Internet_Service(ID, GB, GB_Fee)

Mobile_Phone_Service(ID, Minutes, SMS, Minutes_Fee, SMS_Fee)

Motivation of the logical design

- The Service inheritance has been translated using multiple tables. In particular the Service table has all the common attributes and contains all services (of all types). Services that have exclusive attributes are also stored in one of the auxiliary tables (FixedInternetService, MobileInternetService, obilePhoneService) with the same id of the Service table. This choice has been done because having all types of services in a single table would have resulted in a table with many, nullable attributes.
- Another important thing to notice with respect to Service is that FixedInternetService and MobileInternetService have the same attributes. This has been done to easier map entities in Java.
- The FixedPhoneService entity has been merged with the Service entity since the former has no exclusive attributes.

Auditing update



- **user_strikes**: when the number of rejections of an order increases, the number of rejections of the client associated with such order changes accordingly.
- **solvent_user**: when a user becomes solvent all the auditing entries with its username are marked as inactive.
- **alert_auditing**: if the number of rejections of the user goes from 2 to 3, then a tuple is inserted in the auditing table.
- **insolvent_user**: when the validity state of an order is updated two actions can be done
  - if the state goes from DEFAULT to REJECTED the client is set as insolvent.
  - If the state goes from REJECTED to ACCEPTED the client's number_rejection is decremented by the number of times that order was rejected (i.e. if order 4 was rejected 2 times and has been paid, the client's rejections should be the current value - 2). If number_rejection goes to 0 then the client is flagged as SOLVENT.

User_Strikes

- **Event:** After update on Order
- **Condition:** The number of rejections (number_rejections) changes. The number of rejections of an order can only increases.
- **Action:** Update the number of rejections of the client whose order has been rejected adding 1 to the current number of rejections (client.number_rejections).
- **Code:**

```
create trigger solvent_user
after update on telcoservice_db.`order`
for each row
BEGIN
    IF ( old.number_rejections <> new.number_rejections ) THEN
            UPDATE telcoservice_db.client C
                    SET IS_C.number_rejections = C.number_rejections + 1
                    WHERE C.username = new.client
    END IF;
END;
```

Alert_Auditing

- **Event:** After update on Client
- **Condition:** The number of rejections changes.
- **Action:**
  - If the number of rejections goes from 2 to 3 a insert a tuple in the auditing table with the username and the mail of the client, the value of the order and the current date and time.
  - If the number changes and there was already one tuple associated to the updated client, modify the total amount of the tuple.
- **Code:** (on next slide)

```
create trigger alert_auditing
after update on telcoservice_db.client
for each row
begin
    declare total_value DECIMAL(6, 2);
    declare in_auditing int default (
        select count(*) from telcoservice_db.auditing A
        where A.USERNAME = new.USERNAME and A.IS_ACTIVE = true
    );

   if ( old.number_rejections = 2 AND new.number_rejections = 3 ) then
       set total_value = (
           SELECT sum(O.TOTAL_COST)
           FROM telcoservice_db.`order` O
           WHERE O.CLIENT = new.USERNAME AND O.IS_VALID = 'REJECTED'
            group by O.client
       );

      insert into auditing values ( new.username, new.EMAIL, total_value, current_date, current_time, true );
    elseif ( new.NUMBER_REJECTIONS <> old.NUMBER_REJECTIONS and in_auditing > 0 ) then
        set total_value = (
            SELECT sum(O.TOTAL_COST)
            FROM telcoservice_db.`order` O
            WHERE O.CLIENT = new.USERNAME AND O.IS_VALID = 'REJECTED'
            GROUP BY O.CLIENT
        );



        if ( total_value is null ) then set total_value = 0 end if;

        update telcoservice_db.auditing A
            set A.AMOUNT = total_value
            where A.USERNAME = new.USERNAME and A.IS_ACTIVE = true;

   end if;
end;
```

Insolvent_User

- **Event:** After update on Order
- **Condition:** The validity field (is_valid) changes
- **Action:** If the validity goes from “DEFAULT” to rejected then mark the client as “INSOLVENT”
- **Action:** If the validity goes from “REJECTED” to “ACCEPTED”, then reduce the number of rejections of the client by the number of rejections of the order (client.number_rejections=4, order.number_rejections=2, then client.number_rejections=2). If the number of rejections of the client goes to 0 then the client is flagged as “ SOLVENT ”
- **Code:** (on next slide)

```
create trigger insolvent_user
after update on telcoservice_db.order
for each row
BEGIN
   DECLARE new_client_rejections int;
   DECLARE old_client_rejections int;

   if ( old.IS_VALID = 'DEFAULT' and new.IS_VALID = 'REJECTED' ) then
        -- If the first payment is rejected
      update telcoservice_db.client
         set insolvent = 'insolvent'
            where strcmp (username, new.client) = 0;

   elseif ( old.IS_VALID = 'REJECTED' and new.IS_VALID = 'ACCEPTED' ) then
       -- When the order is finally payed
       SET old_client_rejections = (
            SELECT C.NUMBER_REJECTIONS
            FROM client C
            WHERE NEW.CLIENT = C.USERNAME
        );

       SET new_client_rejections = (
            SELECT C.NUMBER_REJECTIONS - O.NUMBER_REJECTIONS
            FROM telcoservice_db.order O, client C
            WHERE O.id = new.id AND NEW.CLIENT = C.USERNAME
       );

      update telcoservice_db.client C
         set C.NUMBER_REJECTIONS = new_client_rejections
          where NEW.CLIENT = C.USERNAME;

       if ( old_client_rejections > 0 and new_client_rejections = 0 ) then
            update telcoservice_db.client C
            set C.INSOLVENT = 'SOLVENT'
            where NEW.CLIENT = C.USERNAME AND C.INSOLVENT = 'INSOLVENT';
        end if;
   end if;
END;
```

Solvent_User

- **Event:** After update on Client
- **Condition:** The client becomes solvent (”INSOLVENT” -> “SOLVENT”)
- **Action:** Set all entries in the auditing table with the username of the solvent client as inactive (is_active = false)
- **Code:**

```
create trigger solvent_user
after update on telcoservice_db.client
for each row
BEGIN
    IF ( old.INSOLVENT = ‘INSOLVENT’ AND new.INSOLVENT = ‘SOLVENT’ ) THEN
            UPDATE telcoservice_db.auditing
                    SET IS_ACTIVE = 0
                    WHERE auditing.USERNAME = new.USERNAME
    END IF;
END;
```

Expiration dates

- Expiration dates allow an employee to specify if a certain Package, Validity or Service (let us call them assets) can be bought just for a limited period of time or indefinitely (if the date is null). Expiration dates have been added so that one can invalidate an asset without removing it from the database. This action is critical because an asset that can’t be bought could still be used by some clients that have bought a package before its expiration.

trigger_Expiration_date_package

- **Event:** After insert on Service_Composition
- **Condition:** Always
- **Action:**
  - If the expiration date of the new package is bigger than the minimum service expiration date , then set the package expiration date to the minimum service date
- **Code:** (on next slide)

trigger_Expiration_date_package

```
create trigger Expiration_Date_Consistency_Package
-- after, So to check also the latest added service
after insert on service_composition
for each row
BEGIN
   DECLARE min_date date default ( SELECT min(S.EXPIRATION_DATE)
                           FROM service S JOIN service_composition SC on S.ID = SC.SERVICE_ID
                           WHERE  new.PACKAGE_ID = SC.PACKAGE_ID );

   DECLARE package_date date default ( SELECT SP.EXPIRATION_DATE
                                        FROM service_package SP
                                        WHERE  SP.ID = new.PACKAGE_ID );

   if ( min_date  is not null  AND  package_date > min_date ) THEN
      update service_package SP
         SET SP.EXPIRATION_DATE = min_date
            WHERE ( new.PACKAGE_ID = SP.ID);
   elseif ( min_date is null AND package_date is null ) THEN
        update service_package SP
            SET SP.EXPIRATION_DATE = min_date
            WHERE ( new.PACKAGE_ID = SP.ID);
   end if;
END;
```

Schedule

- Schedules are handled with triggers that fire every time an order is accepted.

update_optional_schedule

- **Event:** After update on Order
- **Condition:** The order’s validity (is_valid) goes from not ‘ACCEPTED’ to ’ACCEPTED’
- **Action:** Insert a tuple in the optional_schedule table with the order’s id, and subscription date and the client’s username.
- **Code:**

```
CREATE TRIGGER update_optional_schedule
AFTER UPDATE ON telcoservice_db.`order`
FOR EACH ROW
BEGIN
    DECLARE _period INT;

    -- if the order goes from not accepted to accepted
    IF ( old.IS_VALID <> 'ACCEPTED' AND new.IS_VALID = 'ACCEPTED' ) THEN
        -- add all the optionals in the order to the schedule

        -- get the validity period and save it in a variable
        SET _period = (
            SELECT V.PERIOD
            FROM telcoservice_db.validity V
            WHERE V.PACKAGE_ID = NEW.PACKAGE_ID AND V.ID = NEW.VALIDITY_ID
        );

        -- insert the optional_ids decorated with the name of the user and the starting and ending date
        INSERT INTO telcoservice_db.optional_schedule (
            SELECT NEW.ID, NEW.CLIENT, C.OPTIONAL_PRODUCT_ID, NEW.DATE_SUBSCRIPTION, DATE_ADD(NEW.DATE_SUBSCRIPTION, INTERVAL _period MONTH)
            FROM telcoservice_db.order_optional_composition C
            WHERE C.ORDER_ID = NEW.ID
        );
    END IF;
END;
```

update_service_schedule

- **Event:** After update on Order
- **Condition:** The order’s validity (is_valid) goes from not ‘ACCEPTED’ to ’ACCEPTED’
- **Action:** Insert a tuple in the optional_service table with the order’s id, and subscription date and the client’s username.
- **Code:**

```
CREATE TRIGGER update_service_schedule
AFTER UPDATE ON telcoservice_db.`order`
FOR EACH ROW
BEGIN
    DECLARE _period INT;

    -- if the order goes from not accepted to accepted
    IF ( old.IS_VALID <> 'ACCEPTED' AND new.IS_VALID = 'ACCEPTED' ) THEN
        -- add all the services in the order to the schedule

        -- get the validity period and save it in a variable
        SET _period = (
            SELECT V.PERIOD
            FROM telcoservice_db.validity V
            WHERE V.PACKAGE_ID = NEW.PACKAGE_ID AND V.ID = NEW.VALIDITY_ID
        );

        -- insert the service_ids decorated with the name of the user and the starting and ending date
        INSERT INTO telcoservice_db.service_schedule (
            SELECT NEW.ID, NEW.CLIENT, C.SERVICE_ID, NEW.DATE_SUBSCRIPTION, DATE_ADD(NEW.DATE_SUBSCRIPTION, INTERVAL _period MONTH)
            FROM telcoservice_db.service_composition C
            WHERE C.PACKAGE_ID = NEW.PACKAGE_ID
        );
    END IF;
END;
```

Sales report update

- -- Total value of sales per package with optional products

```
CREATE TABLE value_per_package_op (
   PACKAGE_ID int,
   TOTAL bigint
);
```

- -- Total value of sales per package without optional products

```
CREATE TABLE value_per_package_without(
   PACKAGE_ID int,
   TOTAL bigint
);
```

- -- Number of total purchases per package and validity period

```
CREATE TABLE purchase_per_package_validity (
   PACKAGE_ID int,
    VALIDITY_ID int,
    PURCHASES int
);
```

- -- Number of total purchases per package

```
CREATE TABLE purchase_per_package (
   PACKAGE_ID int,
    PURCHASES int
);
```

- -- Average number of optional products sold together with each service package

```
create table IF NOT EXISTS average_OpProducts_per_ServPackage (
      PACKAGE_ID int,
        AVERAGE_PRODUCTS float
);
```

- -- The best_optional_product contains the id of the optional product that has generated the highest value of sales and the relative value of sales.

```
CREATE TABLE `best_optional_product` (
    `ID` INT,
    `VALUE_OF_SALES` DECIMAL(6, 2)
);
```

- -- Volume of sale for each optional product

```
CREATE TABLE `optional_product_volume_of_sales` (
    `ID` INT,
    `VALUE_OF_SALES` DECIMAL(6, 2)
);
```

Sales report update

Here’s the list of triggers used to update the materialised views used for the sales reports.

- new_package_opProduct
- new_purchase_order_opProduct
- update_volume_of_sales
- update_optional_product_sales
- update_best_optional_product
- new_package_available
- new_purchase
- new_package_validity
- new_purchase_validity
- new_package_value
- new_purchase_value
- new_package_value_op
- new_purchase_value_op

new_package_opProduct

- **Event:** After insert on ServicePackage
- **Condition:** Always (for consistency every new package is immediately added)
- **Action:** Insert a tuple in the average_OpProducts_per_ServPackage materialised view. The tuple has the id of the inserted ServicePackage and average 0
- **Code:**

```
create trigger new_package_opProduct
after insert on service_package
for each row
begin
   insert into average_OpProducts_per_ServPackage
   value (new.ID, 0);
end;
```

new_purchase_order_opProduct

- **Event:** After update on Order
- **Condition:** Order’s validity (is_valid) goes from not ‘ACCEPTED’ to ‘ACCEPTED’
- **Action:** Update the average of average_OpProducts_per_ServPackage materialised view of the package purchased in the Order
- **Code:**

```
create trigger new_purchase_order_opProduct
after update on telcoservice_db.order
for each row
begin

   if ( old.is_valid <> 'ACCEPTED' and new.is_valid = 'ACCEPTED' ) then

      update average_OpProducts_per_ServPackage AOPS
         set AVERAGE_PRODUCTS = ( SELECT count(OPTIONAL_PRODUCT_ID) / count( distinct O.ID )
                                  FROM telcoservice_db.order O
                                      LEFT JOIN order_optional_composition OpComp on O.ID = OpComp.ORDER_ID
                                  WHERE O.PACKAGE_ID = new.PACKAGE_ID )
         where AOPS.PACKAGE_ID = new.PACKAGE_ID;

   end if;
end;
```

update_volume_of_sales

- **Event:** After insert on Optional_Product
- **Condition:** Always (for consistency every new optional product is immediately added)
- **Action:** Insert a new tuple in optional_product_volume_of_sales materialised view with the inserted optional product's id
- **Code:**

```
CREATE TRIGGER update_volume_of_sales
AFTER INSERT ON telcoservice_db.optional_product
FOR EACH ROW
BEGIN
    INSERT INTO telcoservice_db.optional_product_volume_of_sales
    VALUES (new.ID, 0.0);
END;
```

update_optional_product_sales

- **Event:** After update on Order
- **Condition:** The Order’s validity (’is_valid’) goes from not ‘ACCEPTED’ to ‘ACCEPTED’
- **Action:** Uptade the total value sold for each optional product in the inserted Order. The optional products and their total value are stored in the optional_product_volume_of_sales materialised view.
- **Code:** (on next slide)

```sql
-- when an order is created the optional products of the order update the table
-- optional_product_volume_of_sales.
CREATE TRIGGER update_optional_product_sales
    AFTER UPDATE ON telcoservice_db.order
    FOR EACH row
BEGIN
    -- the duration of the order in number of months
    DECLARE duration INT;

    IF ( old.is_valid <> 'ACCEPTED' and new.is_valid = 'ACCEPTED' ) THEN

        SET duration = (
            SELECT period
            FROM validity
            WHERE PACKAGE_ID = new.package_Id AND id = new.validity_Id
        );

        -- update the product in the table
        UPDATE telcoservice_db.optional_product_volume_of_sales VOS
        SET VALUE_OF_SALES = VALUE_OF_SALES + (
            SELECT OP.monthly_fee * duration
            FROM telcoservice_db.Order_Optional_Composition OOC JOIN telcoservice_db.optional_product as OP
                                                                 ON OOC.optional_Product_Id = OP.id
            WHERE OOC.order_Id = new.id and VOS.ID = OP.ID
        )
        WHERE VOS.id IN (
            SELECT OOC.OPTIONAL_PRODUCT_ID
            FROM telcoservice_db.Order_Optional_Composition OOC
            WHERE OOC.ORDER_ID = new.ID
        );

    END IF;
END;
```

update_best_optional_product

- **Event:** After update on optional_product_volume_of_sales
- **Condition:** Always
- **Action:** Replace the tuple in best_optional_product with the optional_product with the highest volume of sales
- **Code:**

```
CREATE TRIGGER update_best_optional_product
    AFTER UPDATE ON telcoservice_db.optional_product_volume_of_sales
    FOR EACH ROW
    BEGIN
        DELETE FROM best_optional_product;
        INSERT INTO best_optional_product (
            SELECT id, VALUE_OF_SALES
            FROM telcoservice_db.optional_product_volume_of_sales
            WHERE VALUE_OF_SALES >= ALL (
                SELECT VALUE_OF_SALES
                FROM telcoservice_db.optional_product_volume_of_sales
            )
        );
END;
```

new_package_available

- **Event:** After insert on service_package
- **Condition:** Always (for consistency every new package is immediately added)
- **Action:** Insert a new tuple in purchase_per_package with the id of the inserted service_package
- **Code:**

```
create trigger new_package_available
after insert on service_package
for each row
begin
   insert into purchase_per_package value ( new.ID, 0 );
end;
```

new_purchase

- **Event:** After update on Order
- **Condition:** The Order’s validity (is_valid) goes from not ‘ACCEPTED’ to ‘ACCEPTED’
- **Action:** Increment the counter relative to the order’s package in the purchase_per_package materialised view.
- **Code:**

```
create trigger new_purchase
after update on telcoservice_db.order
for each row
begin
    if ( old.is_valid <> 'ACCEPTED' and new.is_valid = 'ACCEPTED' ) then
      update purchase_per_package PPP
         set PPP.PURCHASES = PPP.PURCHASES + 1
         where PPP.PACKAGE_ID = new.PACKAGE_ID;
   end if;
end;
```

new_package_validity

- **Event:** After insert on Validity
- **Condition:** Always (for consistency every new validity is immediately added)
- **Action:** Insert a new tuple in the purchase_per_package_validity materialised view. The new tuple has the same key (i.e. package_id, validity_id) of the validity added to Validity.
- **Code:**

```
create trigger new_package_validity
after insert on validity
for each row
begin
   insert into purchase_per_package_validity
   value (new.PACKAGE_ID, new.ID, 0);
end;
```

new_purchase_validity

- **Event:** After update on Order
- **Condition:** If the Order’s validity (is_valid) goes from not ‘ACCEPTED’ to ‘ACCEPTED
- **Action:** Update the purchase_per_package_validity materialised view is updated. In particular increment the purchases’ counter of the rows with same validity_id and package_id of the inserted order.
- **Code:**

```
create trigger new_purchase_validity
after update on telcoservice_db.order
for each row
begin
    if ( old.is_valid <> 'ACCEPTED' and new.is_valid = 'ACCEPTED' ) then
            update purchase_per_package_validity PPPV
            set PPPV.purchases = PPPV.purchases + 1
            where PPPV.PACKAGE_ID = new.PACKAGE_ID and
                  PPPV.VALIDITY_ID = new.VALIDITY_ID;
    end if;
end;
```

new_package_value

- **Event:** After insert on Service_Package
- **Condition:** Always (for consistency every new package is immediately added)
- **Action:** Insert a new tuple in the value_per_package_without materialised view. The new tuple has the id of the of the new service package
- **Code:**

```
create trigger new_package_value
after insert on service_package
for each row
begin
   insert into value_per_package_without value ( new.ID, 0 );
end;
```

new_purchase_value

- **Event:** After update on Order
- **Condition:** The order’s validity (is_valid) goes from not ‘ACCEPTED’ to ‘ACCEPTED’
- **Action:** Update the total value in the value_per_package_without materialised view. Only rows with same package_id and validity_id of the new Order are effected
- **Code:**

```
create trigger new_purchase_value
after update on telcoservice_db.order
for each row
begin
    if ( old.is_valid <> 'ACCEPTED' and new.is_valid = 'ACCEPTED' ) then
      update value_per_package_without VPP
         set VPP.TOTAL = VPP.TOTAL + (
              SELECT V.MONTHLY_FEE * V.PERIOD
              FROM telcoservice_db.order O join validity V
              on (O.VALIDITY_ID, O.PACKAGE_ID) = (V.id, V.PACKAGE_ID)
              WHERE O.id = new.id and o.PACKAGE_ID = new.PACKAGE_ID )
         where VPP.PACKAGE_ID = new.PACKAGE_ID;
   end if;
end;
```

new_package_value_op

- **Event:** After insert on Service_Package
- **Condition:** Always (for consistency every new package is immediately added)
- **Action:** Insert a new tuple in the value_per_package_op materialised view with the same id of the new service_package.
- **Code:**

```
create trigger new_package_value_op
after insert on service_package
for each row
begin
   insert into value_per_package_op value ( new.ID, 0 );
end;
```

new_purchase_value_op

- **Event:** After update on Order
- **Condition:** The order's validity (is_valid) goes from not 'ACCEPTED' to 'ACCEPTED'
- **Action:** Update the total values in the value_per_package_op materialised view. Only rows with same id as the one of the order's package are affected
- **Code:**

```
create trigger new_purchase_value_op
after update on telcoservice_db.order
for each row
begin
    if ( old.is_valid <> 'ACCEPTED' and new.is_valid = 'ACCEPTED' ) then
      update value_per_package_op VPPO
         set VPPO.TOTAL = VPPO.TOTAL + (
              SELECT O.TOTAL_COST
              FROM telcoservice_db.order O
              WHERE O.id = new.id )
         where VPPO.PACKAGE_ID = new.PACKAGE_ID;
   end if;
end;
```

ORM Design

Order – Client

ManyToOne

Owner: Order

Description: An Order can be done only one Client but a Client can do many Orders.

- **Order -> Client**: An order is done by a client. (FetchType is EAGER since it is left to default)
- **Client -> Order**: added for consistency and because it’s useful to know, given a client a list of all its orders. Lazy because when we get a client we might not want to know her/his orders hence they might be too many, thus it might be costly to retreive all of them.

Order – OptionalProduct

ManyToMany

Owner: Order

Description: An Order can contain many OptionalProduct. An OptionalProduct can be in many different Orders.

- **Order -> OptionalProduct**: An Order contains many OptionalProducts. FetchType is EAGER because we are retrieving all the order’s details.
- **OptionalProduct -> Order**: Added for consistency. FetchType is LAZY

ServicePackage - OptionalProduct

- **ManyToMany**
- **Owner: ServicePackage**
- **Description**: An ServicePackage can possibly contain many OptionalProduct. An OptionalProduct can be associated to many different ServicePackages.

- **ServicePackage -> OptionalProduct**: A ServicePackage can offer many (or no) OptionalProducts. FetchType is EAGER because we are retrieving all the service package’s details.
- **OptionalProduct -> ServicePackage**: Added because given an OptionalProduct, it might be useful to know in which ServicePackage it’s offered. The fetch policy is LAZY since when retrieving an OptionalProduct we might not need the associated ServicePackages.

Order - Validity

- **ManyToOne**
- **Owner:** Order
- **Description**: An Order (actually the ServicePackage of an Order) is associated with a Validity but a Validity can be in different orders since different Orders can contain the same ServicePackage (and the key of the Validity is the id of the validity and the validity of the service package).

- **Order -> Validity**: An Order has a Validity (i.e. a validity period and a monthly fee). FetchType is EAGER (left to default) because we are retrieving all the order's details.
- **Validity -> Order**: Added because, given a Validity, it's useful to know the Orders in which it's chosen. The fetch policy is LAZY because when retrieving a Validity we might not immediately want the Orders in which it appears.

Order - ServicePackage

- **ManyToOne**
- **Owner:** Order
- **Description**: An Order is associated with a ServicePackage but a ServicePackage can be in different orders since different Orders can contain the same ServicePackage.

- **Order -> ServicePackage**: An Order contains a ServicePackage. FetchType is EAGER (left to default) because we are retrieving all the order's details.
- **ServicePackage -> Order**: A ServicePackage can be bought in different Orders. This relationship is added because, given a ServicePackage, it's useful to know the Orders in which it's bought. The fetch policy is LAZY because when retrieving a ServicePackage we might not immediately want the Orders in which it appears.

ServicePackage - Service

- **ManyToMany**
- **Owner:** ServicePackage
- **Description**:

- **ServicePackage -> Service**: A ServicePackage is composed of many different Services. FetchType is EAGER because we are retrieving all the ServicePackage’s details.
- **Service -> ServicePackage**: A Service can be offered in different ServicePackages. This relationship is added because, given a Service, it’s useful to know the ServicePackages in which it’s bought. The fetch policy is LAZY.

OrderServiceSchedule - Order

- **ManyToOne**
- **Owner:** OrderServiceSchedule
- **Description**:

- **OrderServiceSchedule -> Order**: An OrderServiceSchedule refers to one Order only.
- **Order -> OrderServiceSchedule**: An Order can be in different OrderServiceSchedules since an order may be composed of different services. The fetch policy is LAZY since most of the times we don’t need the Schedules associated with an Order. Still the relation is kept since it might be useful when consulting the schedule (in a future implementation).

OrderServiceSchedule - Service

- **ManyToOne**
- **Owner:** OrderServiceSchedule
- **Description**:

- **OrderServiceSchedule -> Service**: An OrderServiceSchedule refers to one Service only.
- **Service -> OrderServiceSchedule**: A Service can be in different OrderServiceSchedules since a Service can be offered in different Orders. The fetch policy is LAZY since most of the times we don’t need the Schedules associated with a Service. Still the relation is kept since it might be useful when consulting the schedule (in a future implementation).

OrderOptionalSchedule - Order

- **ManyToOne**
- **Owner:** OrderServiceSchedule
- **Description**:

- **OrderOptionalSchedule -> Order**: An OrderOptionalSchedule refers to one Order only.
- **Order -> OrderOptionalSchedule**: An Order can be in different OrderOptionalSchedules since an client might have bought different optionals with an Order. The fetch policy is LAZY since most of the times we don’t need the schedules associated with an Order. Still the relation is kept since it might be useful when consulting the schedule (in a future implementation).

OrderOptionalSchedule - OptionalProduct

- **ManyToOne**
- **Owner:** OrderOptionalSchedule
- **Description**:

- **OrderOptionalSchedule -> OptionalProduct**: An OrderOptionalSchedule refers to one OptionalProduct only.
- **OptionalProdcut -> OrderOptionalSchedule**: An OptionalProduct can be in different OrderOptionalSchedules since the same optional can be offered in different Orders. The fetch policy is LAZY since most of the times we don’t need the schedules associated with an OptionalProduct. Still the relation is kept since it might be useful when consulting the schedule (in a future implementation).

Validity - ServicePackage

- **ManyToOne**
- **Owner:** Validity
- **Description**:

- **Validity -> ServicePackage**: A Validity refers only to one ServicePackage because it is a weak entity. The fetch policy is EAGER (left to default) since it is a single object to be retrieved.
- **ServicePackage -> Validity**: A ServicePackage offers a Client different Validities to choose among. FetchType is EAGER because we are retrieving all the ServicePackage's details.

ENTITIES

MobileInternetService

```
@Entity
@Table(name = "mobile_internet_service")
@DiscriminatorValue("MOBILE_INTERNET")
public class MobileInternetService extends Service {
    private static final long serialVersionUID = 1L;

    @Column(name = "gb")
    private Integer gigaByte;

    @Column(name = "gb_fee", precision = 2)
    private BigDecimal gigaByteFee;
}
```

Client

```
@Entity
@NamedQuery(name = "Client.withCredentials", query = "SELECT r FROM
Client r  WHERE r.username = ?1 and r.password = ?2")
@NamedQuery(name = "Client.insolvent", query = "SELECT c FROM
Client c WHERE c.insolvent =
it.polimi.db2.telcoservice_sc42.entities.UserStatus.INSOLVENT" )
public class Client implements Serializable {
    private static final long serialVersionUID = 1L;

    // TODO: consider adding an id.

    @Id
    private String username;
    private String password;
    private String email;

    @Column(name = "NUMBER_REJECTIONS")
    private Integer numberOfRejections;

    @Column(columnDefinition = "ENUM('SOLVENT', 'INSOLVENT')")
    @Enumerated(EnumType.STRING)
    private UserStatus insolvent;

    // A client can do many orders.
    @OneToMany(mappedBy="client", fetch=FetchType.LAZY)
    private List<Order> orders;

    @
```

```
OneToMany(mappedBy = "username", fetch = FetchType.L)
private List<Auditing> auditings;

    public Client() {
        this.numberOfRejections = 0;
        this.insolvent = UserStatus.SOLVENT;
    }

    public Client(String username, String email, String password) {
        this();
        this.username = username;
        this.email = email;
        this.password = password;
    }
}
```

OptionalProduct

```java
@Entity
@Table(name = "optional_product")
@NamedQuery(name = "OptionalProduct.all", query = "SELECT p FROM
OptionalProduct p")
@NamedQuery(name = "OptionalProduct.valid", query = "SELECT p FROM
OptionalProduct p WHERE p.expirationDate = null OR p.expirationDate >=
current_date ")
public class OptionalProduct implements Serializable, Representable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;

    @Column(name = "monthly_fee", precision = 2)
    private BigDecimal monthlyFee;

    @Temporal(TemporalType.DATE)
    @Column(name = "expiration_date")
    private Date expirationDate;


    @ManyToMany(mappedBy = "optionals", fetch=FetchType.LAZY)
    private List<Order> orders;

    @ManyToMany(mappedBy = "products", fetch=FetchType.LAZY)
    private List<ServicePackage> packages;

    @OneToMany(mappedBy = "optional", fetch = FetchType.LAZY)
    private List<OrderOptionalSchedule> optionalSchedules;

    public OptionalProduct() {
        this("");
    }

    public OptionalProduct(String name) {
        this.name = name;
        this.monthlyFee = BigDecimal.ZERO;
        this.expirationDate = null;
    }

    public OptionalProduct(String name, BigDecimal fee, Date
expirationDate) {
        this(name);
        this.monthlyFee = fee;
        this.expirationDate = expirationDate;
    }
}
```

FixedInternetService

```
@Entity
@Table(name = "fixed_internet_service")
@DiscriminatorValue("FIXED_INTERNET")
public class FixedInternetService extends Service {
    private static final long serialVersionUID = 1L;

    @Column(name = "gb")
    private Integer gigaByte;

    @Column(name = "gb_fee", precision = 2)
    private BigDecimal gigaByteFee;
}
```

Order

```java
@Entity
@Table(name = "`order`")
@NamedQuery(name = "Order.rejected", query = "SELECT o FROM Order o WHERE o.status =
it.polimi.db2.telcoservice_sc42.entities.OrderStatus.REJECTED")
public class Order implements Serializable, Representable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;

    @Column(name = "HOUR_CREATION")
    private Time creationHour;

    @Column(name = "DATE_CREATION")
    private Date creationDate;

    @Column(name = "NUMBER_REJECTIONS")
    private Integer numberOfRejections;

    @Column(name = "DATE_SUBSCRIPTION")
    private Date subscriptionDate;

    @Column(name = "TOTAL_COST")
    private BigDecimal totalCost;

    @Enumerated(EnumType.STRING)
    @Column(name = "IS_VALID")
    private OrderStatus status;

    // an order refers to one validity period, but the same validity
    // period can be assigned to multiple orders.
    @ManyToOne
    @JoinColumns({
            @JoinColumn(name = "VALIDITY_ID", referencedColumnName = "ID"),
            @JoinColumn(name = "PACKAGE_ID", referencedColumnName = "PACKAGE_ID", updatable =
false, insertable = false )
    })
    private Validity validity;


    // an order refers to a package only, but the same package can be part
    // of multiple orders.
    @ManyToOne
    @JoinColumn(name = "PACKAGE_ID", referencedColumnName = "ID")
    private ServicePackage servicePackage;


    // An order has just one client but a client can do different orders.
    @ManyToOne
    @JoinColumn(name = "CLIENT")
    private Client client;

    @ManyToMany ( fetch = FetchType.EAGER )
    @JoinTable(
            name = "order_optional_composition",
            joinColumns = @JoinColumn(name="ORDER_ID"),
            inverseJoinColumns = @JoinColumn(name="OPTIONAL_PRODUCT_ID"))
    private List<OptionalProduct> optionals;

    @OneToMany (mappedBy = "order", fetch = FetchType.LAZY )
    private List<OrderServiceSchedule> serviceSchedules;

    @OneToMany (mappedBy = "order", fetch = FetchType.LAZY )
    private List<OrderOptionalSchedule> optionalSchedules;


    public Order(Client client, Validity validityId, ServicePackage packageId, Date
subscriptionDate, List<OptionalProduct> optionals ) {
        this();
        this.client = client;
        this.validity = validityId;
        this.servicePackage = packageId;
        this.subscriptionDate = subscriptionDate;

        BigDecimal optionalsFee = new BigDecimal(0);
        for ( OptionalProduct o: optionals ) {
            optionalsFee = optionalsFee.add(o.getMonthlyFee());
        }

        BigDecimal bigPeriod = new BigDecimal(validityId.getPeriod());

        this.totalCost = ( validityId.getMonthlyFee().add(optionalsFee) ).multiply(bigPeriod);
        this.optionals = new ArrayList<>(optionals);
    }}
```

MobilePhoneService

```
@Entity
@Table(name = "mobile_phone_service")
@DiscriminatorValue("MOBILE_PHONE")
public class MobilePhoneService extends Service {
    private static final long serialVersionUID = 1L;

    private Integer minutes;

    private Integer sms;

    @Column(name = "minutes_fee", precision = 2)
    private BigDecimal minutesFee;

    @Column(name = "sms_fee", precision = 2)
    private BigDecimal smsFee;
}
```

Employee

```
@Entity
@Table(name = "employee")
@NamedQuery(name = "Employee.withCredentials", query = "SELECT e FROM Employee e  WHERE
e.id = ?1 and e.password = ?2")
public class Employee implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @Column(name = "USERNAME", nullable = false)
    private String id;

    @Column(name = "PASSWORD", nullable = false, length = 31)
    private String password;

    @Column(name = "EMAIL", nullable = false)
    private String email;

}
```

Service

```
@Entity
@Table(name = "service")
@Inheritance(
        strategy = InheritanceType.JOINED
)
@DiscriminatorColumn(
        name="TYPE",
        discriminatorType = DiscriminatorType.STRING
)
@DiscriminatorValue("FIXED_PHONE")
@NamedQuery(name = "Service.valid", query = "SELECT s FROM
Service s WHERE ( s.expirationDate = NULL OR s.expirationDate
>= current_date )")
public class Service implements Serializable, Representable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(columnDefinition = "ENUM('FIXED_PHONE',
'MOBILE_PHONE', 'FIXED_INTERNET', 'MOBILE_INTERNET')")
    @Enumerated(EnumType.STRING)
    private ServiceType type;

    @Column(name = "expiration_date")
    private Date expirationDate;

    @ManyToMany (mappedBy = "services", fetch =
FetchType.LAZY )
    private List<ServicePackage> packages;

    @OneToMany (mappedBy="service", fetch = FetchType.LAZY )
    private List<OrderServiceSchedule> serviceSchedules;

    public Service() {}

    public Service(ServiceType type, Date expirationDate) {
        this.type = type;
        this.expirationDate = expirationDate;
    }

}
```

ServicePackage

```
@Entity
@Table(name = "service_package")
@NamedQuery(name = "ServicePackage.valid", query = "SELECT p FROM
ServicePackage p WHERE ( p.expirationDate >= current_date OR
p.expirationDate = null ) ")
public class ServicePackage implements Serializable, Representable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;

    @Temporal(TemporalType.DATE)
    @Column(name = "EXPIRATION_DATE")
    private Date expirationDate;

    @OneToMany(mappedBy = "servicePackage", fetch=FetchType.LAZY)
    private List<Order> orders;

    @ManyToMany ( fetch = FetchType.EAGER, cascade =
             CascadeType.PERSIST )
    @JoinTable(
            name = "service_composition",
            joinColumns = @JoinColumn(name="PACKAGE_ID"),
            inverseJoinColumns = @JoinColumn(name="SERVICE_ID"))
    private List<Service> services;

    @ManyToMany ( fetch = FetchType.EAGER , cascade =
             CascadeType.PERSIST )
    @JoinTable
            (name = "optional_product_composition",
                    joinColumns = @JoinColumn(name="package_id"),
                    inverseJoinColumns =
@JoinColumn(name="optional_product_id"))
    private List<OptionalProduct> products;

    @OneToMany(mappedBy = "servicePackage", fetch = FetchType.EAGER ,
cascade = CascadeType.ALL)
    private List<Validity> validities;

    public ServicePackage(){ }

    public ServicePackage(String name) {
        this.name = name;
        this.expirationDate = null;
    }

    public ServicePackage(String name, Date expirationDate) {
        this.name = name;
        this.expirationDate = expirationDate;
    }

}
```

Validity

```
@Entity
@IdClass(ValidityPrimaryKey.class)
public class Validity implements Serializable, Representable
{
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Id
    @ManyToOne
    @JoinColumn (name = "PACKAGE_ID")
    private ServicePackage servicePackage;

    private Integer period;

    @Column(name = "MONTHLY_FEE", precision = 2)
    private BigDecimal monthlyFee;

    @Column(name = "EXPIRATION_DATE")
    private Date expirationDate;

    @OneToMany(mappedBy="validity", fetch=FetchType.LAZY)
    private List<Order> orders;

    @Column(name = "TOTAL_COST")
    private BigDecimal totalCost;
```

```
    public Validity() {
    }

    public Validity(ServicePackage servicePackage, int
period, BigDecimal monthlyFee, Date expirationDate ){
            this(servicePackage, period, monthlyFee);
            this.expirationDate = expirationDate;
}

     public Validity(ServicePackage servicePackage, int
period, BigDecimal monthlyFee ){
            this.setServicePackage(servicePackage);
            this.period = period;
            this.monthlyFee = monthlyFee.setScale(2,
RoundingMode.HALF_UP);
            this.totalCost =
            monthlyFee.multiply(BigDecimal.valueOf(period)).se
tScale(2, RoundingMode.HALF_UP);
     }

}
```

OrderServiceSchedule

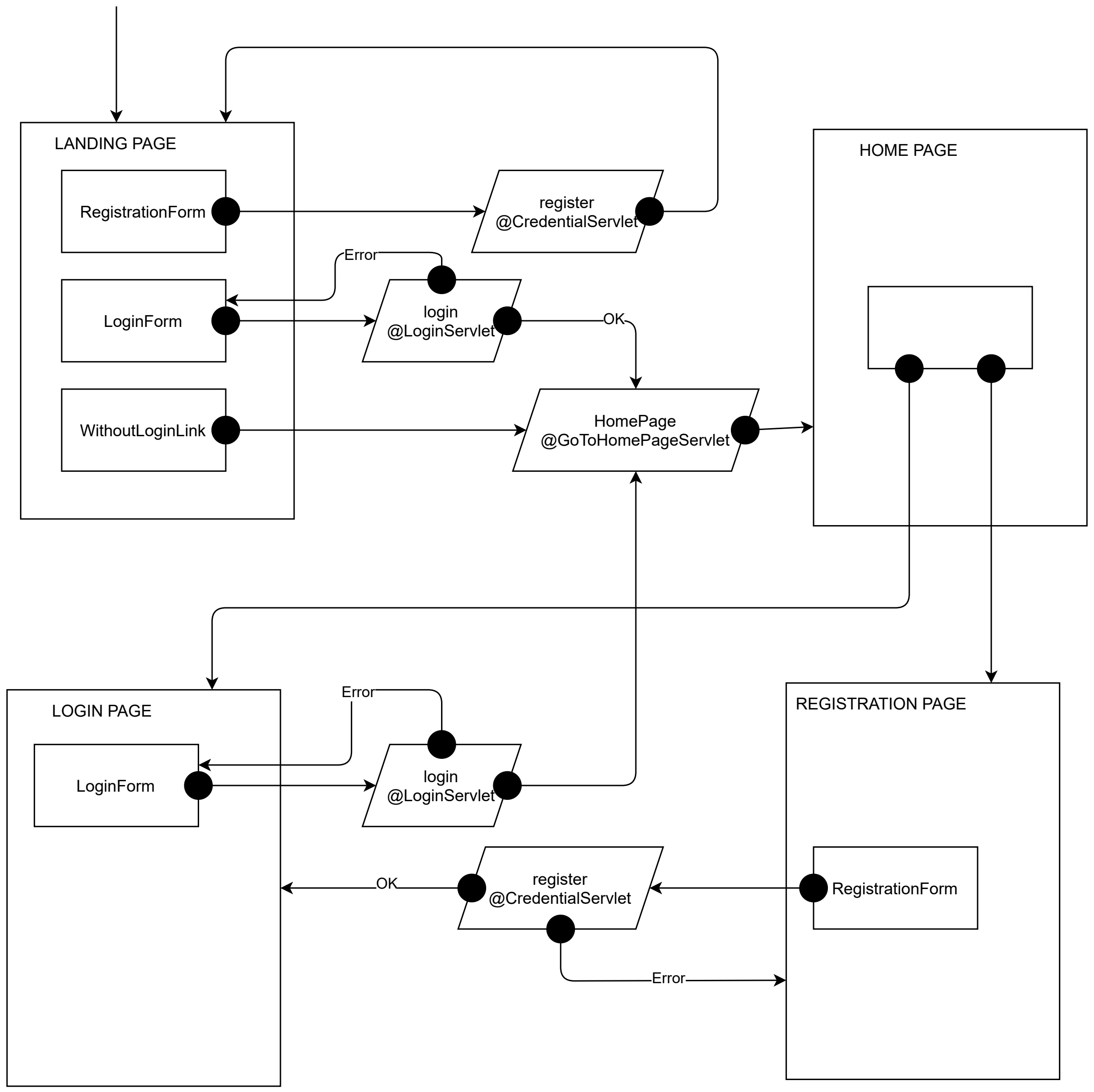
```
@Entity
@Table(name = "service_schedule")
@IdClass(OrderServiceSchedulePrimaryK
ey.class)
public class OrderServiceSchedule
implements Serializable {
    private static final long
serialVersionUID = 1L;

    @Id
    @ManyToOne ( cascade =
CascadeType.PERSIST )
    @JoinColumn (name = "ORDER_ID")
    private Order order;

    @Id
    @ManyToOne ( cascade =
CascadeType.PERSIST )
    @JoinColumn (name = "SERVICE_ID")
    private Service service;

    @Column(name = "username")
    private String username;

    @Column(name = "ACTIVATION_DATE")
    private Date activationDate;

    @Column(name =
"DEACTIVATION_DATE")
    private Date deactivationDate;


    public OrderServiceSchedule() {
    }}
```

OrderOptionalSchedule

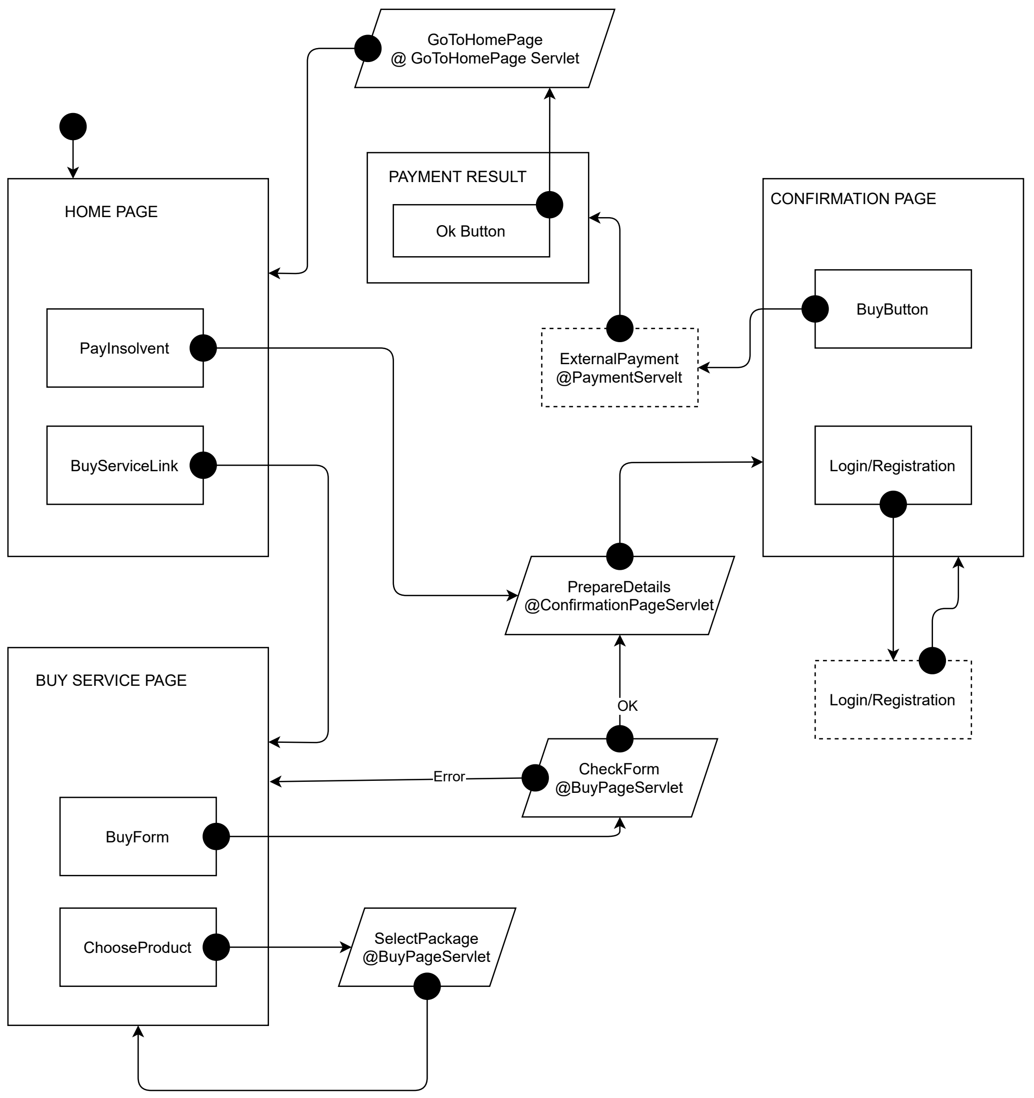
```
@Entity
@Table(name = "service_schedule")
@IdClass(OrderServiceSchedulePrimaryKey.class
)
public class OrderServiceSchedule implements
Serializable {
    private static final long
serialVersionUID = 1L;

    @Id
    @ManyToOne ( cascade =
CascadeType.PERSIST )
    @JoinColumn (name = "ORDER_ID")
    private Order order;

    @Id
    @ManyToOne ( cascade =
CascadeType.PERSIST )
    @JoinColumn (name =
"OPTIONAL_PRODUCT_ID")
    private OptionalProduct optional;

    @Column(name = "username")
    private String username;

    @Column(name = "ACTIVATION_DATE")
    private Date activationDate;

    @Column(name = "DEACTIVATION_DATE")
    private Date deactivationDate;


    public OrderServiceSchedule() {
    }

}
```

INTERACTION DIAGRAMS

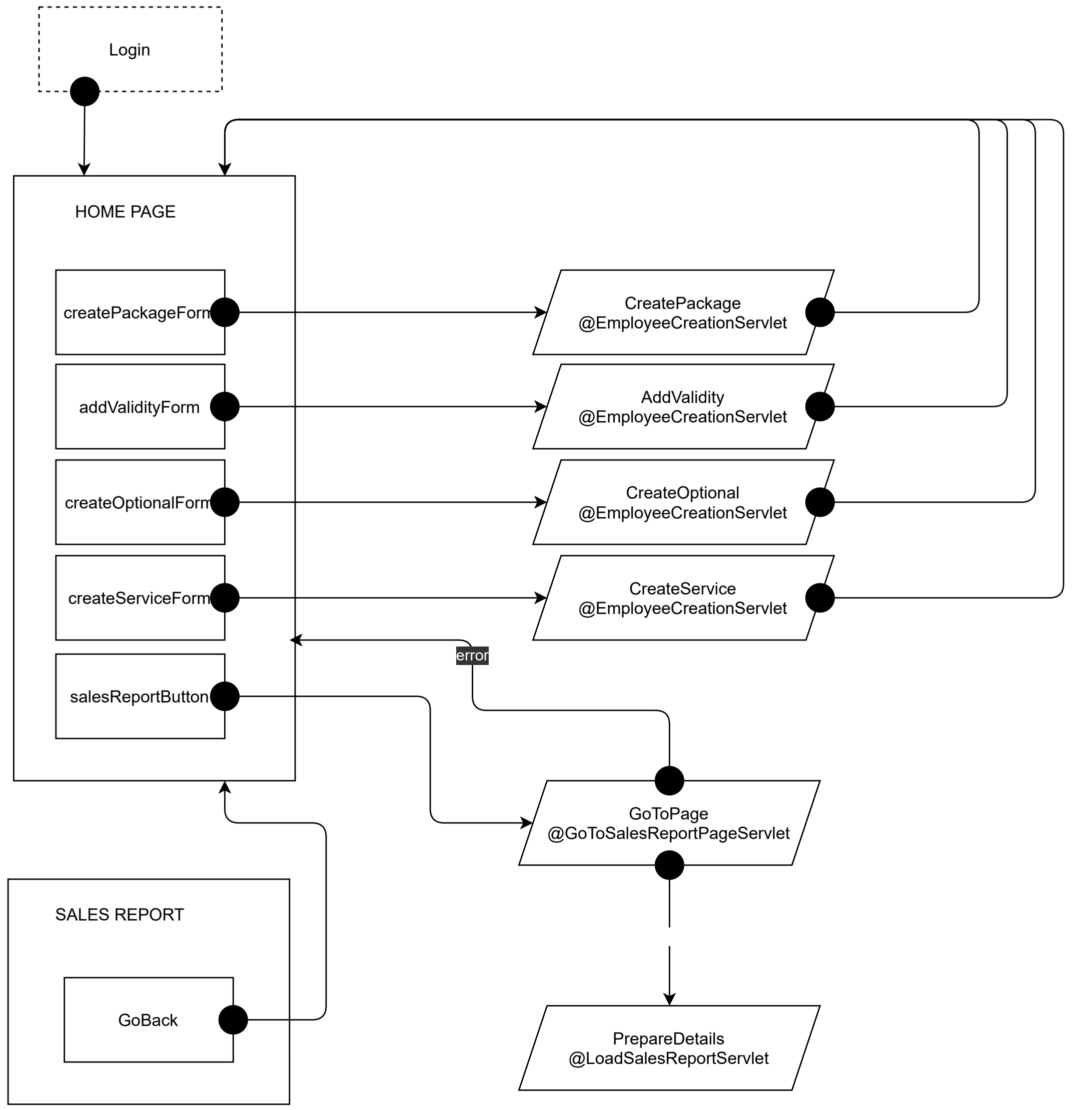
Login and Registration Diagram



Buy Diagram


GoToHomePage
@ GoToHomePage Servlet
PAYMENT RESULT
Ok Button
HOME PAGE
CONFIRMATION PAGE
BuyButton
PayInsolvent
ExternalPayment
@PaymentServelt
BuyServiceLink
Login/Registration
PrepareDetails
@ConfirmationPageServlet
BUY SERVICE PAGE
OK
Login/Registration
Error
CheckForm
@BuyPageServlet
BuyForm
ChooseProduct
SelectPackage
@BuyPageServlet

Employee Diagram


Login
HOME PAGE
createPackageForm
addValidityForm
createOptionalForm
createServiceForm
salesReportButton
CreatePackage @EmployeeCreationServlet
AddValidity @EmployeeCreationServlet
CreateOptional @EmployeeCreationServlet
CreateService @EmployeeCreationServlet
error
GoToPage @GoToSalesReportPageServlet
SALES REPORT
GoBack
PrepareDetails @LoadSalesReportServlet

Client Components

Client Components

- Servlets
  - BuyPageServlet
  - ConfirmationPageServlet
  - CredentialServlet
  - GoToHomePageServlet
  - LoginServlet
  - LogoutServlet
  - PaymentServlet
- Views
  - index.jsp
  - buyService.jsp
  - confirmation.jsp
  - home.jsp
  - login.jsp
  - paymentResult.jsp
  - registration.jsp

Back end components

- Entities
- EJB (Services)
  - ClientService
  - EmployeeService
  - OrderService
  - OptionalProductService
  - PackageService
  - SalesReportService
  - ServiceService
  - ValidityService

Employee Components

Client Components

- Servlets
  - CredentialServlet
  - EmployeeCreationServlet
  - GoToHomePageServlet
  - GoToSalesReportPageServlet
  - LoadOptionalsEmployeeServlet
  - LoadSalesReportServlet
  - LoginServlet
  - LogoutServlet
- Views
  - home.jsp
  - login.jsp
  - salesReport.jsp

Back end components

- Entities
- EJB (Services)
  - ClientService
  - EmployeeService
  - OrderService
  - OptionalProductService
  - PackageService
  - SalesReportService
  - ServiceService
  - ValidityService

Back end Components

Back end components

- ClientService
  - **Stateless**
  - Client addClient(String username, String email, String password)
  - Boolean isRegistered(String username)
  - Client checkCredentials(String username, String pwd)
- OptionalProductService
  - **Stateless**
  - OptionalProduct createOptionalProduct(String name, BigDecimal fee, Date expirationDate)
  - OptionalProduct findOptionalProductById(int id)
  - List<OptionalProduct> findValidOptionalProducts()

Back end components

- EmployeeService
  - **Stateless**
  - Employee checkCredentials(String username, String password)
- OrderService
  - **Stateless**
  - List<Order> findOrdersByClient(String username)
  - List<Order> findRejectedOrdersByClient(String username)
  - Order findOrderById(int orderId)
  - Integer createOrder (String clientUsername, int validityId, int packageId, Date dateSubscription, List<Integer> optionals)
  - void setOrderStatus(int id, OrderStatus status)

Back end Components

Back end components

- SalesReportService
  - **Stateless**
  - List<Map<String, String>> getAllPurchasesPerPackage()
  - List<Map<String, String>> getAllPurchasesPerPackageValidity()
  - List<Map<String, String>> getAllValuePerPackageWithOptionalProduct()
  - List<Map<String, String>> getAllValuePerPackageWithoutOp()
  - List<Map<String, String>> getAllAveragesOptionalProductsPerPackage()
  - Map<String, String> findBestOptionalProduct()
  - List<Map<String, String>> insolventUsers()
  - List<Map<String, String>> suspendedOrders()
  - List<Map<String, String>> getAlerts()

Back end components

- PackageService
  - **Stateless**
  - ServicePackage findServicePackageById(int packageId)
  - List<ServicePackage> findValidServicePackages()
  - ServicePackage createServicePackage(String name, Date expirationDate, List<Integer> serviceIds, List<Integer> optionalIds, List<IndependentValidityPeriod> periods)
- ServiceService
  - **Stateless**
  - Service createService(ServiceType type, Date expirationDate, BigDecimal gbFee, Integer gbs, BigDecimal smsFee, Integer sms, BigDecimal callFee, Integer minutes)
  - List<Service> findValidServices()