Data bases 2

TSA - TELCO SERVICE APPLICATION

Documentation

Authors: Daniele Locatelli (10618397) & Xu Qiongjie (10757496)

https://github.com/ictar/TSA-DB2-Project-2022

Index

- Specification
  - Revision of the specifications
- Conceptual (ER) and logical data models
  - Explanation of the ER diagram
- Trigger design and code
- ORM relationship design with explanations
- Entities code
- Interface diagrams or functional analysis of the specifications
- List of components
  - Motivations of the components design
- UML sequence diagrams

Specifications

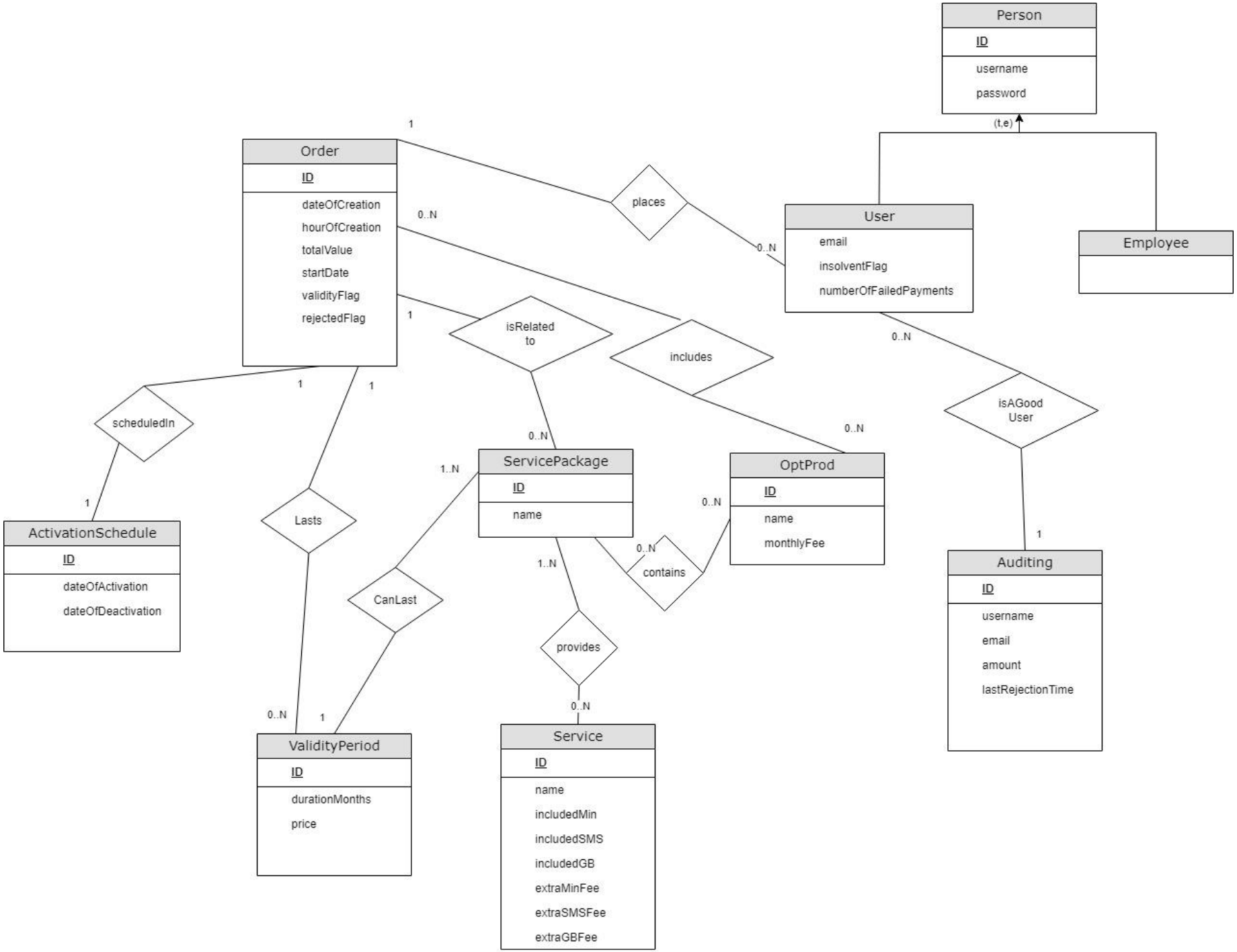
We implement an application for Telco that handles operations from both users and employees. They must be able to interact with offers that the company provides. The company provides service packages that are defined by services, validity periods and possible optional products.
Each person can browse the user application to see what is offered by the company and define the desired order, according to what is included in each specific service package. The user must be logged in to request that the order is issued, so the log in phase can be carried out before starting the browsing or after the order is defined.
The application provides a homepage to the user that allows the user to choose whether to define a new order or to see previous ones that were not yet paid and possibly complete them by performing a valid payment.
The application must also record users that consecutively fail payments and change the his/her status.

The employee application allows an already defined employee to log in.
The functionalities of this application are:

1. define new service packages and, with that, all its subparts (service, validity period, optional product)
2. access a sales report section that show some statistics about sales
    - Number of total purchases per package
    - Number of total purchases per package and validity period
    - Total value of sales per package with and without the optional products
    - Average number of optional products sold together with each service package
    - List of insolvent users, suspended orders and alerts
    - Best seller optional product

Specification interpretation

- Employee registration is not requested in specification, therefore, the data of the employee assumed to be already present in the database directly
- The application will reset to zero the count of failed payments of a user when he/she performs a correct payment, even if there are still unsolved orders
- The payment process is not implemented here but it is only handled by functions that simulate the possible outcomes and act accordingly.


Person
ID
username
password
(t,e)
Order
ID
dateOfCreation
hourOfCreation
totalValue
startDate
validityFlag
rejectedFlag
places
User
email
insolventFlag
numberOfFailedPayments
Employee
isRelated to
includes
scheduledIn
Lasts
ActivationSchedule
ID
dateOfActivation
dateOfDeactivation
ServicePackage
ID
name
OptProd
ID
name
monthlyFee
contains
isAGood User
Auditing
ID
username
email
amount
lastRejectionTime
CanLast
provides
ValidityPeriod
ID
durationMonths
price
Service
ID
name
includedMin
includedSMS
includedGB
extraMinFee
extraSMSFee
extraGBFee
1
0..N
1..N

Motivations of the ER design

- Each validity period can belong to one and only one service package to obtain better flexibility on prices
- Activation schedule is the owner of the ActivationSchedule-Order relationship because it is more efficient: in fact every day the system can scan the ActivationSchedule table and see what orders must be activated or deactivated, without having to scan all the orders

Relational model

Primary key, foreign key

Employee (**ID**, username, password)
Auditing (**ID**, userId, username, email, amount, lastRejectionTime)
User (**ID**, username , email, password, insolventFlag, numFailedPayments)
ServicePkg (**ID**, name)
ServicesInPkg(**ServicePkgId**, **ServiceID**)
OptProdInPkg (**servicePkgId**, **optProdId**)
Service(**ID**, name, includedMin, includedSMS, includedGB, extraMinFee, extraSMSFee, extraGBFee)
OptProduct (**ID**, name, monthlyFee)
ValidityPeriod(**ID**, monthDuration, price, servicePkgId)
Orders (**ID**, userId, servicePkgId, dateOfCreation, hourOfCreation, validityPeriodId, totalValue, startDate, validityFlag, rejectedFlag)
ActivationSchedule (**ID**, dateOfActivation, dateOfDeactivation, orderId)
ChosenOptProd(**OrderProductsID**, **OptProdID)**

*View tables will be described in following slides

Employee (**ID**, username, password)

Auditing (**ID**, userId, username, email, amount, lastRejectionTime)

User (**ID**, username , email, password, insolventFlag, numFailedPayments)

ServicePkg (**ID**, name)

ServicesInPkg(**servicePkgId**, **ServiceID**)

OptProdInPkg (**servicePkgId**, **optProdId**)

Service(**ID**, name, includedMin, includedSMS, includedGB, extraMinFee, extraSMSFee, extraGBFee)

OptProduct (**ID**, name, monthlyFee)

ValidityPeriod(**ID**, monthDuration, price, servicePkgId)

Orders (**ID**, userId, servicePkgId, dateOfCreation, hourOfCreation, validityPeriodId, totalValue, startDate, validityFlag, rejectedFlag)

ActivationSchedule (**ID**, dateOfActivation, dateOfDeactivation, orderId)

ChosenOptProd(**OrderProductsID**, **OptProdID**)

Trigger design & code

Auditing trigger

Event: when user performs failed payment

Condition: performed third failed payment

Action: add new row in auditing

Code:

```
CREATE TRIGGER `checkInsolventUserAfterUpdate`
AFTER UPDATE ON `user`
FOR EACH ROW
BEGIN
        DECLARE totalAmount float;
        DECLARE currentTime timestamp;
        set currentTime = current_timestamp();
        IF (new.numFailedPayments = 3 and new.numFailedPayments <>
        old.numFailedPayments) THEN
                set totalAmount =
                        (SELECT sum(totalValue)
                        FROM orders WHERE userId = new.id and rejectedFlag = 1);
                insert into telcoservicedb.auditing (userId, username, email, amount,
                lastRejectionTime)
                values (new.id, new.username, new.email, totalAmount, currentTime);

        END IF;
```

Trigger design motivation:
In this case row or statement doesn't make any difference because each statement will modify only one row. We chose AFTER update to maintain a logical order in what is performed but in this case both BEFORE and AFTER work properly. The trigger surely terminates because it doesn't fire any other trigger

Sales Report trigger

Event: when an order becomes valid

Condition: `validityFlag` == 1

Action:

- Increase the number of purchase for the servicepackage associated to the order (`PurchasePerSP`)
- Increase the number of purchase for the servicepackage and validity period associated to the order (`PurchasePerSPVP`)
- Calculate the average number of optional products sold together with the servicepackage associated to the order (`AvgProdSalesPerSP`)
- If there are optional products sold together in the order, increase the value of sales for the package with products (`ServicePkgSaleWithProd`), and increase the number of sales for each product.
- Otherwise, increase the value of sales for the package without products. (`ServicePkgSaleNoProd`)

Sales Report trigger

logical schema of materialized view tables

The database schema defines the following materialized view tables:

- PurchasePerSP(spid, ordercnt)
  - Number of total purchases per service package
- PurchasePerSPVP(spid, vpid, ordercnt)
  - Number of total purchases per service package and validity period
- AvgProdSalesPerSP(spid, avgProdcnt)
  - Average optional products sold together with each service package
- ServicePkgSaleWithProd(spid, valSale)
  - Total value of sales per package with optional products
- ServicePkgSaleNoProd(spid, valSale)
  - Total value of sales per package without optional products
- ProdSale(pid, saleCnt)
  - Number of total purchases per product

Sales Report trigger

SQL code of materialized view tables

The content of the materialized view tables can be specified with the following definitions

```
create table PurchasePerSP as
    select o.servicePkgId as spid, count(distinct o.id) as ordercnt
    from orders o
    where o.validityFlag = 1
    group by o.servicePkgId;

create table PurchasePerSPVP as
    select o.servicePkgId as spid, o.validityPeriodId as vpid, count(distinct o.id) as ordercnt
    from orders o
    where o.validityFlag = 1
    group by o.servicePkgId, o.validityPeriodId;

create table AvgProdSalesPerSP as
    select o.servicePkgId as spid, count(p.optProdId)/count(distinct o.id) as avgProdcnt
    from orders o
    left join chosenOptProd p on o.id = p.orderId
    where o.validityFlag = 1
    group by o.servicePkgId;
```

Sales Report trigger

SQL code of materialized view tables (CONTINUE)

```
create table ServicePkgSaleWithProd as
        select o.servicePkgId as spid, sum(o.totalValue) as valSale
        from orders o
        where o.validityFlag = 1 and o.id in (select distinct(orderId) from chosenOptProd)
        group by o.servicePkgId;


create table ServicePkgSaleNoProd as
        select o.servicePkgId as spid, sum(distinct o.totalValue) as valSale
        from orders o
        left join chosenOptProd p on o.id = p.orderId
        where o.validityFlag = 1 and p.optProdId is null
        group by o.servicePkgId;


create table prodSale as
        select optProdId as pid, count(orderId) as saleCnt
        from chosenOptProd p
        join orders o on o.id = p.orderId
        where o.validityFlag = 1 -- sold
        group by optProdId;
```

Sales Report trigger

Code

```
CREATE TRIGGER after_order_valid
AFTER UPDATE ON orders
FOR EACH ROW
BEGIN
        -- number of optional products associated with this order
        DECLARE prodCount INT;
        -- Average number of optional products sold together with each service package.
        DECLARE avgProdCount FLOAT;
        -- temp product id
        DECLARE tmpid INT;
        -- End flag variable （The default is 0）
        DECLARE done INT DEFAULT 0;
        -- cursor for optional products associated with this order
        DECLARE prod_cur CURSOR for SELECT optProdId FROM chosenOptProd WHERE orderId=new.id;
        DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

        IF (new.validityFlag != old.validityFlag and new.validityFlag = 1) THEN
                SELECT count(optProdId) INTO prodCount FROM chosenOptProd WHERE orderId=new.id;
                select count(p.optProdId)/count(distinct o.id) into avgProdCount from orders o left join chosenOptPro
                d p on o.id = p.orderId where o.validityFlag = 1 and o.servicePkgId = new.servicePkgId;
```

Sales Report trigger

Code (Continue)

```
-- Update: Number of total purchases per package.
IF (exists
        (select * from PurchasePerSP where spid=new.servicePkgId)) THEN
        update PurchasePerSP
        set ordercnt=ordercnt+1
        where spid=new.servicePkgId;
ELSE
        insert into PurchasePerSP (spid, ordercnt)
        values (new.servicePkgId, 1);
END IF;
-- Update: Number of total purchases per package and validity period.
IF (exists
        (select * from PurchasePerSPVP where spid=new.servicePkgId and vpid=new.validityPeriodId)) THEN
        update PurchasePerSPVP
        set ordercnt=ordercnt+1
        where spid=new.servicePkgId and vpid=new.validityPeriodId;
ELSE
        insert into PurchasePerSPVP (spid, vpid, ordercnt)
        values (new.servicePkgId, new.validityPeriodId, 1);
END IF;
-- Update: Average number of optional products sold together with each service package.
IF (exists
        (select * from AvgProdSalesPerSP where spid=new.servicePkgId)) THEN
        update AvgProdSalesPerSP
        set avgProdcnt = avgProdCount
        where spid=new.servicePkgId;
ELSE
        insert into AvgProdSalesPerSP (spid, avgProdcnt)
        values (new.servicePkgId, prodCount);
END IF;
```

Sales Report trigger

Code (Continue)

```
IF (prodCount > 0) THEN
        -- update total value of sales per package with products
        IF (exists
                (select * from ServicePkgSaleWithProd where spid=new.servicePkgId)) THEN
                update ServicePkgSaleWithProd set valSale = valSale + new.totalValue
                where spid=new.servicePkgId;
        ELSE -- no exist
                insert into ServicePkgSaleWithProd (spid, valSale) values (new.servicePkgId, new.totalValue);
        END IF;
        -- update the product sales count
        OPEN prod_cur;
        prod_loop: LOOP
                FETCH prod_cur into tmpid;
                IF done = 1 THEN
                        LEAVE prod_loop;
                END IF;
                -- update
                IF (exists
                        (select * from prodSale where pid=tmpid)) THEN
                        update prodSale set saleCnt = saleCnt + 1 where pid = tmpid;
                ELSE
                        insert into prodSale (pid, saleCnt) values (tmpid, 1);
                END IF;
        END LOOP;
        close prod_cur;
```

Sales Report trigger

Code (End)

```
            ELSE -- update total value of sales per package without products
                  IF (exists
                        (select * from ServicePkgSaleNoProd where spid=new.servicePkgId)) THEN
                        update ServicePkgSaleNoProd
                        set valSale = valSale + new.totalValue
                        where spid=new.servicePkgId;
                  ELSE -- no exist
                        insert into ServicePkgSaleNoProd (spid, valSale)
                        values (new.servicePkgId, new.totalValue);
                  END IF;
            END IF;
      END IF;
END;
```

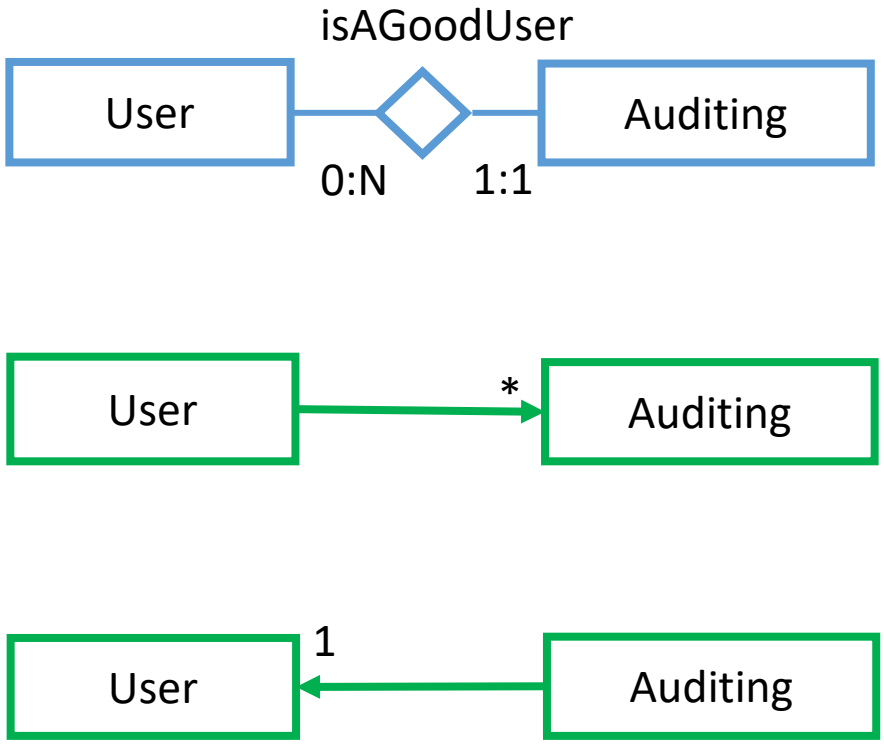
Sales Report trigger

Design Motivation

In this case row or statement doesn’t make any difference because each statement will modify on ly one row. We choose AFTER update to maintain a logical order in what is performed. Since the modifications are not done on the same table, there is no performance issue. The
trigger surely terminates because it doesn’t fire any other trigger

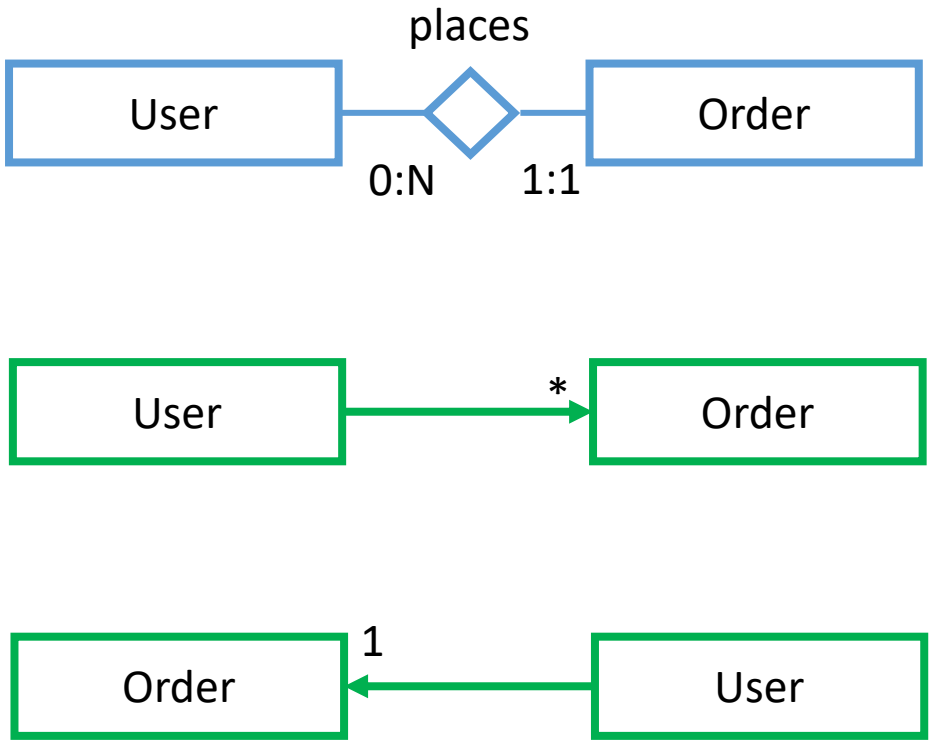
ORM design

Relationship “isAGoodUser”



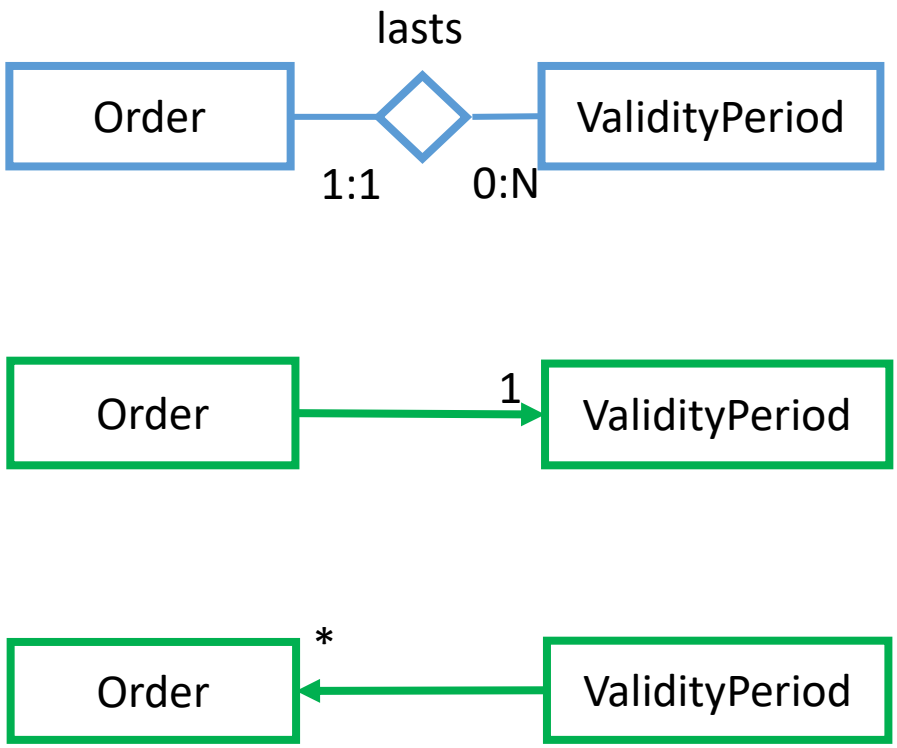
- User → Auditing
  @OneToMany not used, but added for possible future uses
  - FetchType is LAZY, because we don’t always need the User’s Auditing
  - No cascade
- Auditing → User @ManyToOne is necessary to get the user related to each alert
  - FetchType EAGER because we usually would like to know who is the insolvent user
  - No cascade because no operation on User should be allowed from the Auditing class
- Owner = Auditing

Relationship “places”



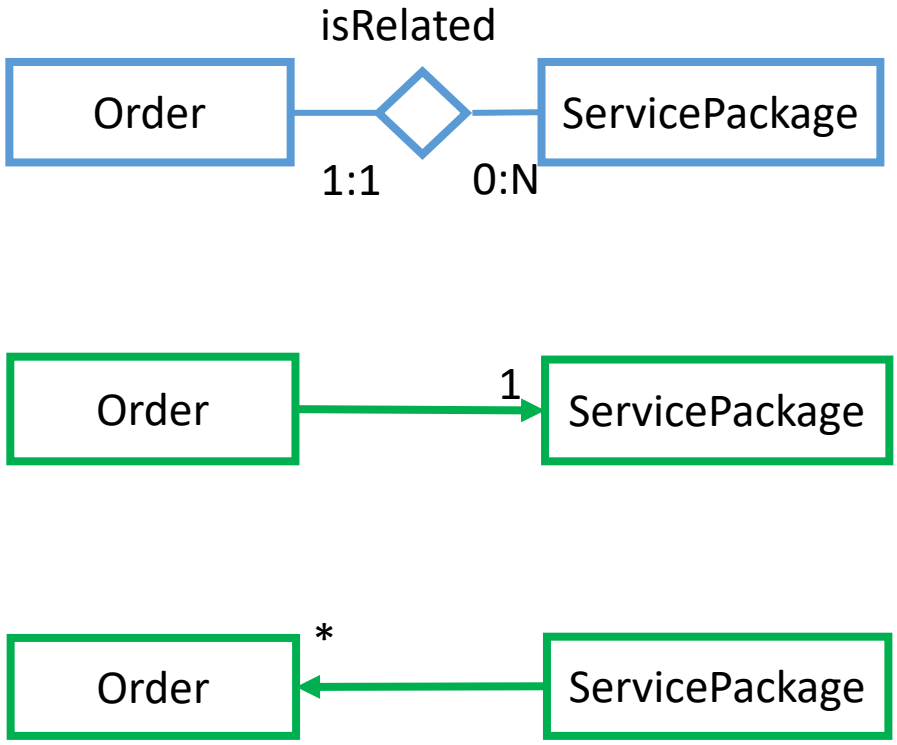
- User → Order
  @OneToMany is necessary to get the orders of the logged in user.
  - FetchType can be EAGER to let the client access the orders of the user via relationship navigation
  - No cascade is needed because the modification made on User should not affect the Order
- Order→ User
  @ManyToOne, useful relation to set who purchased an Order
  - FetchType is EAGER (for each Order, only need to retrieve one User)
  - No cascade is needed because the modification made on Order should not affect the User
- Owner = Order

Relationship “lasts”



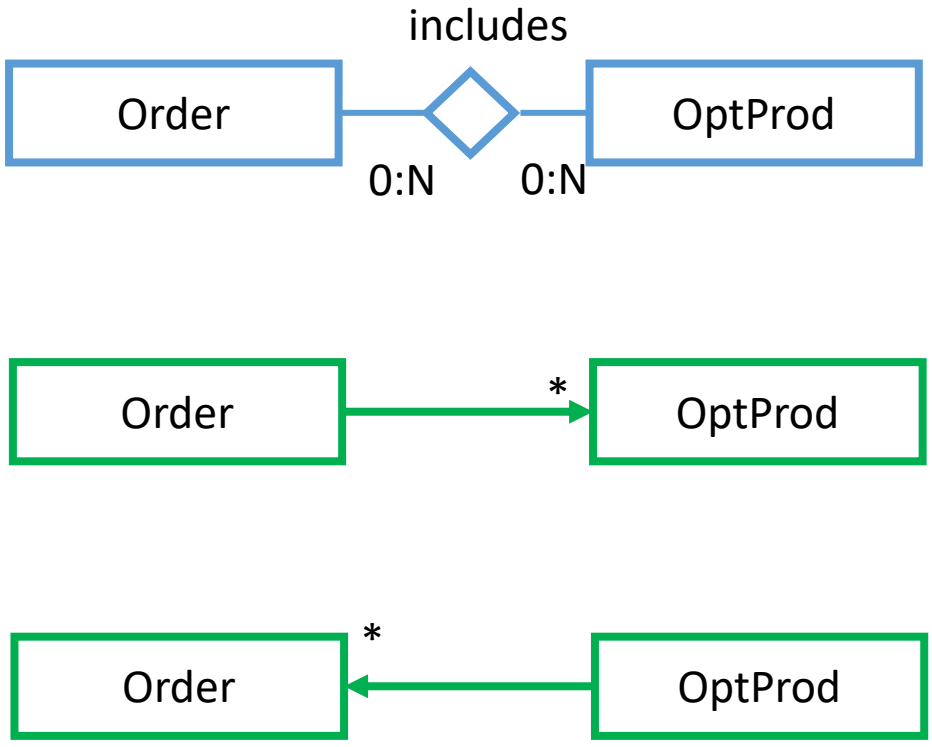
- Order → ValidityPeriod
  @ManyToOne is necessary to get the validity period for the order shown in the CONFIRMATION PAGE
  - FetchType must be EAGER (when the Order is retrieved, the validity period must be retrieved)
  - No cascade is needed because the modification made on Order should not affect the ValidityPeriod
- ValidityPeriod→ Order
  @OneToMany, since it is not used right now the configurations are:
  - FetchType = LAZY
  - No cascade performed
- Owner = Order

Relationship “isRelated”



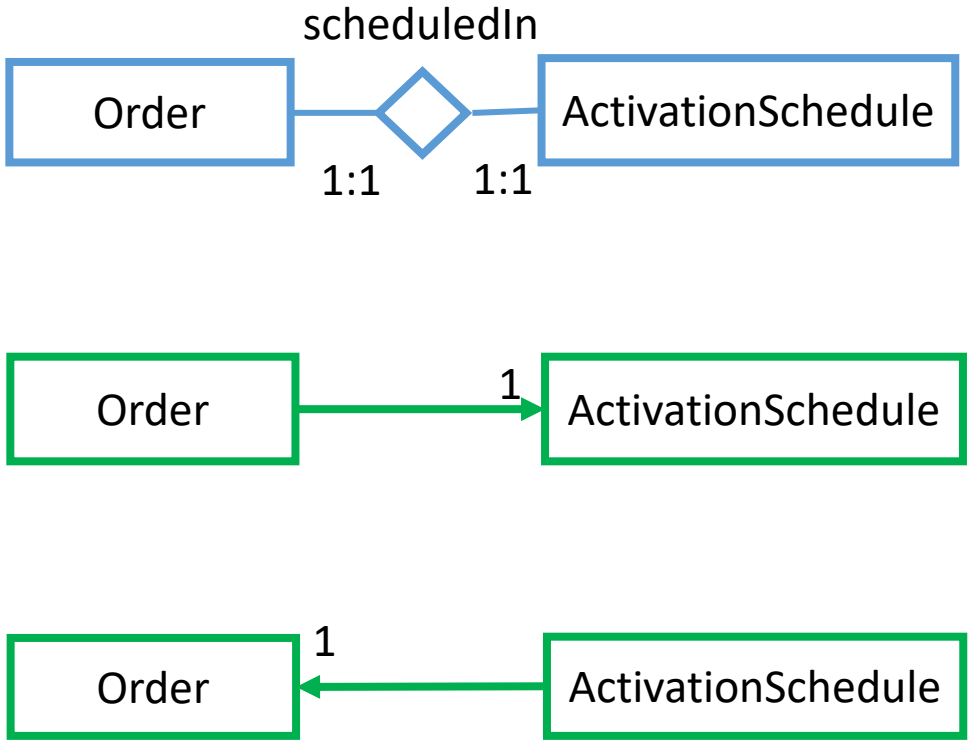
- Order → ServicePackage @ManyToOne is necessary to get the service
  - FetchType = EAGER (when the Order is retrieved, the service package must be retrieved)
  - No cascade is needed because the modification made on Order should not affect the ServicePackage
- ServicePackage→ Order @OneToMany is not requested by the specification, mapped for consistency. In fact:
  - FetchType = LAZY (no access from service package to order)
  - No cascade
- Owner = Order

Relationship “includes”



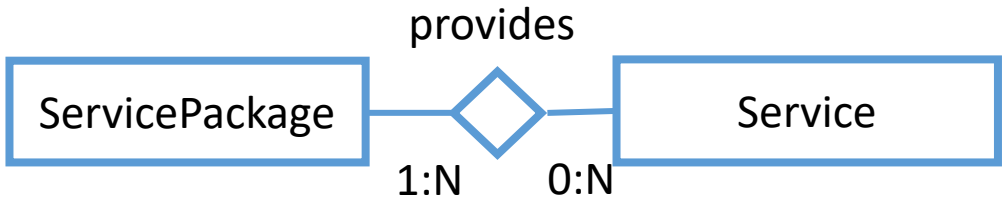
- Order → OptProd
  @ManyToMany is necessary to show the list of optional products associated to the Order
  - FetchType must be EAGER (when the Order is retrieved, the optional products must always be retrieved)
  - No cascade is needed because the modification made on Order should not affect the OptProd
- OptProd → Order
  @ManyToMany non necessary, can be mapped for consistency. In fact:
  - FetchType is LAZY (There may be lots of Orders per product)
  - No cascade
- Owner = Order

Relationship “scheduledIn”



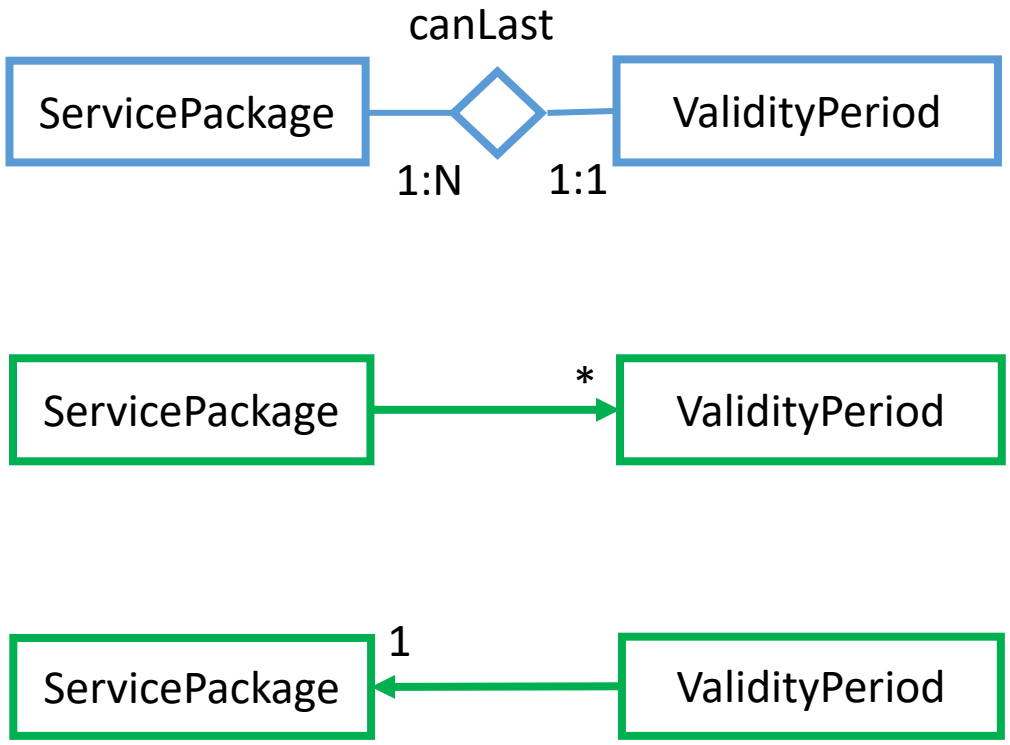
- Order → ActivationSchedule @OneToOne, useful to assign an ActivationSchedule to the order
  - Fetch Type = LAZY, usually we don’t need ActivationSchedule when retrieving the order
  - PERSIST are cascaded, to store the order along with its ActivationSchedule
- ActivationSchedule → Order @OneToOne is necessary to get the order to activate
  - Fetch Type can be EAGER by default since it is single-valued relationship
  - Don’t cascade: no operation can be done on Order from ActivationSchedule
- Owner = ActivationSchedule

Relationship “provides”



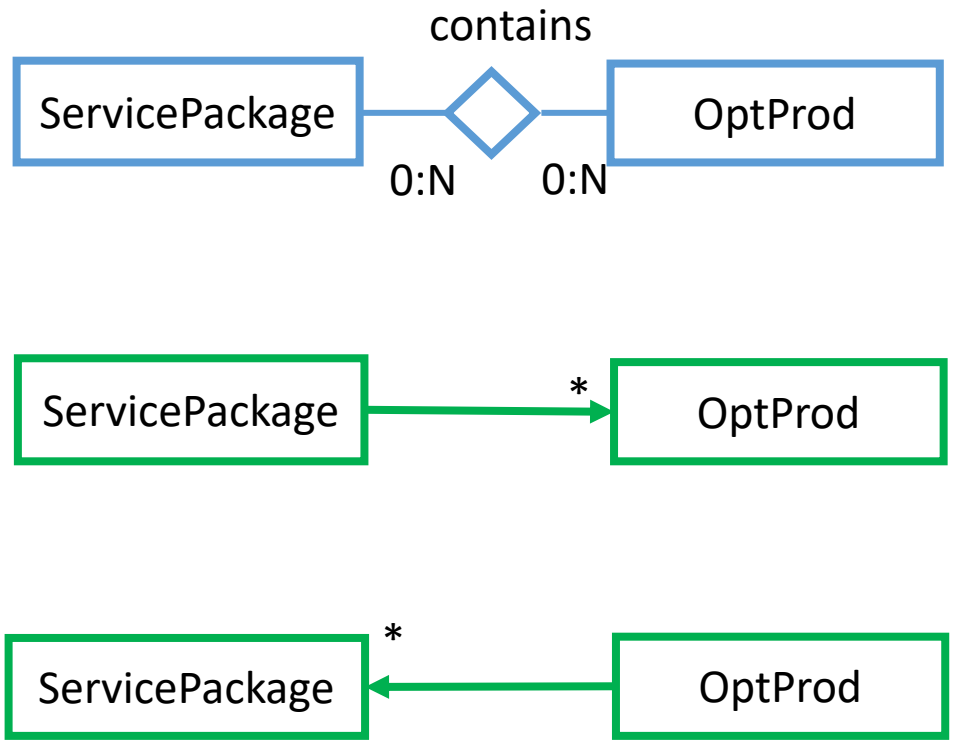
- ServicePackage → Service
  @ManyToMany is necessary to show the services associated to the service package in Buy Service Page
  - FetchType is EAGER (when the service package is retrieved, the services must be retrieved).
  - No cascade is needed because modifications made on ServicePackage should not affect Service
- Service → ServicePackage
  @ManyToMany is not requested, but mapped for simplicity
  - FetchType = LAZY (there may be lots of service package per service)
  - No cascade is needed because the modification made on Service should not affect the ServicePackage
- Owner = ServicePackage

Relationship “canLast”



- ServicePackage → ValidityPeriod @OneToMany is necessary to show the validity periods associated to the service package in Buy Service Page
  - FetchType is EAGER (when the service package is retrieved, the validity period must be retrieved)
- ValidityPeriod → ServicePackage @ManyToOne is not requested, but mapped for simplicity. In fact:
  - FetchType = LAZY
  - No cascade
- Owner = ValidityPeriod

Relationship “contains”



- ServicePackage → OptProd
  @ManyToMany is necessary to show the list of optional products contained in the ServicePackage in BuyService Page
  - FetchType = EAGER (when the service package is retrieved, the optional products must be retrieved)
  - No cascade is needed because the modification made on ServicePackage should not affect the OptProd
- OptProd → ServicePackage
  @ManyToMany non necessary, can be mapped for consistency
  - FetchType = LAZY (There may be lots of service packages per product)
  - No cascade
- Owner = ServicePackage

Entities code

In the following slides we report all the Java classes defined to represent the entities.
We reported only relevant attributes to understand the behaviour and role of the class, omitting getters, setters and contructors which are present in the actual classes.
All attributes have getters and setters, except id attribute that only has the getter method.
All classes have constructors which are parameterless.

Entity ActivationSchedule

```
@Entity
public class ActivationSchedule {

    @Id
    @GeneratedValue(
    strategy = GenerationType.IDENTITY)
    private int id;
    private LocalDate dateOfActivation;
    private LocalDate
    dateOfDeactivation;

    @OneToOne(fetch= FetchType.EAGER)
    @JoinColumn(name="orderId")
    private Order order;

    public LocalDate getDateOfAct() {
        return dateOfActivation;
    }

    public void setDateOfAct(LocalDate
    dateOfAct) {
        this.dateOfActivation =
        dateOfAct;
    }

    public LocalDate getDateOfDeact() {
        return dateOfDeactivation;
    }

    public void setDateOfDeact(
    LocalDate dateOfDeact) {
        this.dateOfDeactivation =
        dateOfDeact;
    }

    public void setOrder(Order order) {
        this.order = order;
    }
}
```

Entity Auditing

```
@Entity
@NamedQuery(name="Auditing.findAll", query="SELECT a from Auditing a")

public class Auditing implements
Serializable {

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

    private String username;
    private String email;
    private float amount;
    private Timestamp lastRejectionTime;

    @ManyToOne
    @JoinColumn(name="userID")
    private User user;

    public void setUser(User user) {
        this.user = user;
        this.username =
        user.getUsername();
        this.email = user.getEmail();
    }
}
```

Entity Employee

```
@Entity
@NamedQuery(name = "Employee.checkCredenetials",
    query = "SELECT e From Employee e WHERE e.username=?1 and e.password=?2")
public class Employee implements
Serializable {

    @Id
    @GeneratedValue
    private int id;

    private String username;
    private String password;

    public int getId() {
        return id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String
    username) {
        this.username = username;
    }

    public String getPassword() {
    return password;
    }

    public void setPassword(String
    password) {
    this.password = password;
    }
}
```

Entity OptProduct

```
@Entity
@Table(name="optproduct")
@NamedQuery(name = "OptProduct.findOne",
    query = "SELECT op FROM OptProduct op WHERE op.id=?1")
public class OptProduct implements
Serializable {

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

    private String name;
    private float monthlyFee;

    // relationships

    // optProd -> servicePackage
    @ManyToMany(
        mappedBy="availableOptProds",
        fetch=FetchType.LAZY)
    private Set<ServicePackage>
    servicePkgs;

    // optProduct -> Order
    @ManyToMany(
        mappedBy="chosenOptProds",
        fetch=FetchType.LAZY)
    private Set<Order> orders;

}
```

Entity Order (1/2)

```
@Entity
@Table(name = "orders")
@NamedQueries ({
    @NamedQuery(name = "Order.getSuspended",
        query = "SELECT o From Order o WHERE o.rejectedFlag = true"),
    @NamedQuery(name = "Order.getUserOrders",
        query = "SELECT o FROM Order o WHERE o.user=?1")
})

public class Order {
    @Id
    @GeneratedValue(
    strategy =
    GenerationType.IDENTITY)
    private int id;

    private float totalValue;
    private LocalDate startDate;
    private LocalDate dateOfCreation;
    private int hourOfCreation;
    private boolean validityFlag;
    private boolean rejectedFlag;
```

```
    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(
        name = "validityPeriodId")
    private ValidityPeriod validityPeriod;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "userId")
    private User user;

    @OneToOne(cascade =
    CascadeType.PERSIST,
    mappedBy = "order“, fetch =
    FetchType.LAZY)
    private ActivationSchedule
    activationSchedule;
```

Entity Order (2/2)

```
    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name="servicePkgId")
    private ServicePackage servicePackage;

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(
        name = "chosenOptProd",
        joinColumns= @JoinColumn(name ="orderId"),
        inverseJoinColumns = @JoinColumn(name ="optProdId")
    )
    private Set<OptProduct> chosenOptProds;

    public float computeTotalValue() {
        totalValue = validityPeriod.getPrice();
        totalValue = chosenOptProds.stream().map(product ->
        product.getMonthlyFee()).reduce(totalValue,
        (a, b) -> a + b);
        totalValue *= validityPeriod.getMonthDuration();
        return totalValue;
    }
}

//Class Order finishes here
```

Entity Service

```
@Entity
public class Service {

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

    private String name;
    private int includedMin;
    private int includedSMS;
    private int includedGB;
    private float extraMinFee;
    private float extraSMSFee;
    private float extraGBFee;

    @ManyToMany(mappedBy="availableServices",fetch = FetchType.LAZY)
    private Set<ServicePackage> servicePackageRelated;
}
```

Entity ServicePackage (1/2)

```
@Entity
@Table(name = "ServicePkg")
@NamedQueries({
    @NamedQuery(name = "ServicePackage.findAll",
        query = "SELECT sp FROM ServicePackage sp"),
    @NamedQuery(name = "ServicePackage.findOne",
        query = "SELECT sp FROM ServicePackage sp WHERE sp.id=?1")})
public class ServicePackage {
    @Id
    @GeneratedValue(strategy =
    GenerationType.IDENTITY)
    private int id;
    private String name;

    @OneToMany(
    mappedBy = "servicePackage",
    fetch = FetchType.EAGER)
    private Set<ValidityPeriod>
    validityPeriods;
```

```
    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(name = "servicesInPkg",
    joinColumns = @JoinColumn(name =
    "servicePkgId"), inverseJoinColumns
    = @JoinColumn(name = "serviceId"))
    private Set<Service>
    availableServices;

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

Entity ServicePackage (2/2)

```
    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(name = "optprodinpkg", joinColumns = { @JoinColumn(name =
    "servicePkgId") },
    inverseJoinColumns = {
    @JoinColumn(name = "optProdId") })
    private Set<OptProduct> availableOptProds;


    public void addValidityPeriod(ValidityPeriod vp) {
        if (validityPeriods == null) {
            validityPeriods = new HashSet<ValidityPeriod>();
        }
        validityPeriods.add(vp);
        vp.setServicePackage(this);
    }

}
//ServicePackage class finishes here
```

Entity User (1/2)

```
@Entity
@NamedQueries({
    @NamedQuery(name = "User.checkCredentials",
        query = "SELECT u From User u WHERE u.username=?1 and u.password=?2",
        hints =@QueryHint(name= QueryHints.REFRESH, value= HintValues.TRUE)),
    @NamedQuery(name = "User.checkDuplicateUsername",
        query = "SELECT u From User u WHERE u.username=?1"),
    @NamedQuery(name = "User.getInsolvents",
        query = "SELECT u From User u WHERE u.insolventFlag = true")
})
public class User implements
Serializable {

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

    private String username;
    private String email;
    private String password;
    private boolean insolventFlag;
    private int numFailedPayments;

    // relationships
    // user -> order
    @OneToMany(fetch=FetchType.EAGER,
    mappedBy="user")
    private List<Order> orders;

    // user -> auditing
    @OneToMany(fetch=FetchType.LAZY,
    mappedBy = "user")
    private List<Auditing> audits;
```

Entity User (2/2)

```
public int failedPayment() {
    insolventFlag = true;
    numFailedPayments++;
    return numFailedPayments;
}

public void addOrder(Order order) {
    orders.add(order);
}

public void addAudit(Auditing audit)
{
    audit.setUser(this);
    audits.add(audit);
}
```

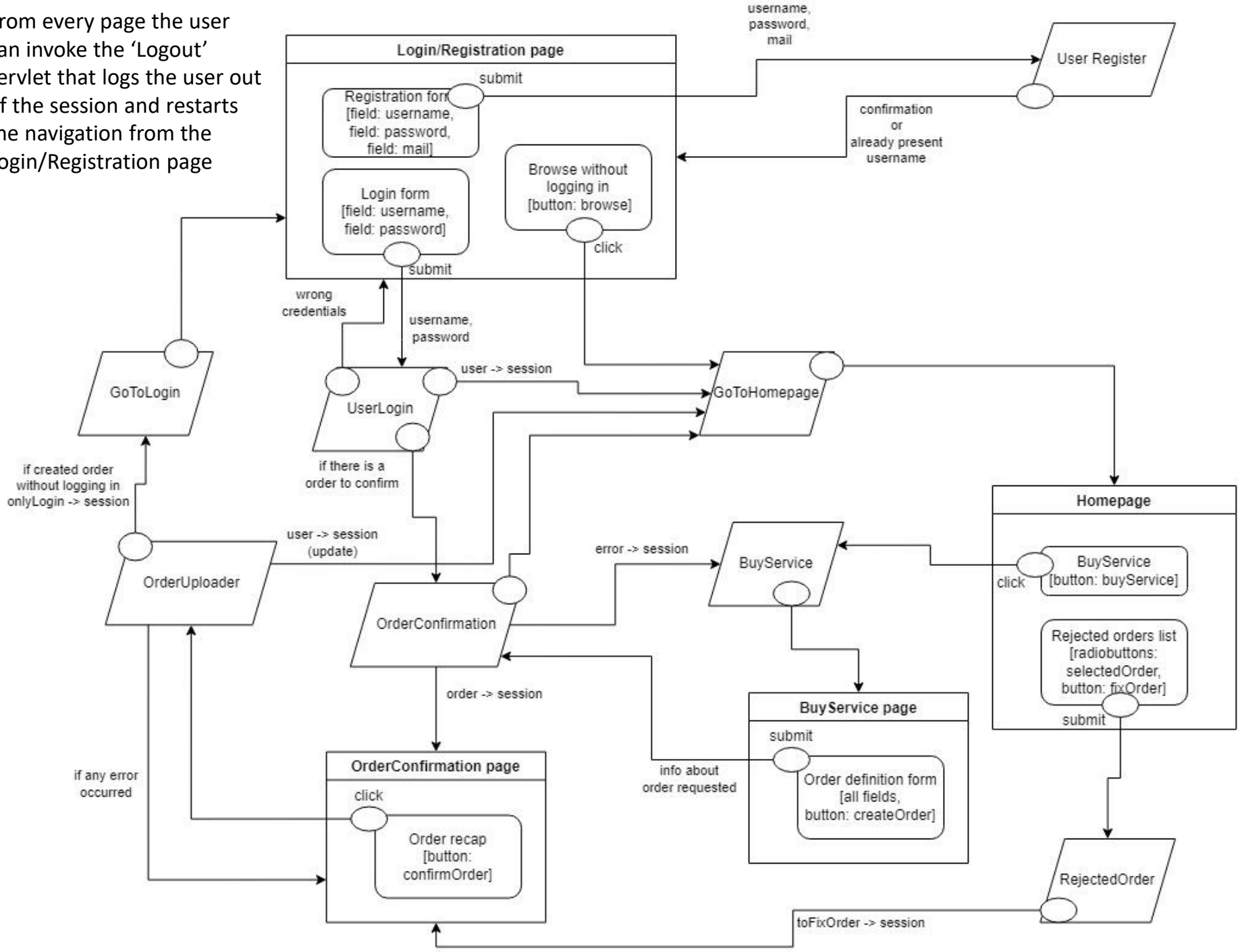
```
    public void decreaseFailedPayments()
    {
        boolean hasRejectedOrder =
        false;

        numFailedPayments =0;
        for(int i = 0;
            i<orders.size();
            i++) {
        if(orders.
            get(i).
            isRejectedFlag())
                hasRejectedOrder= true;
        }

        insolventFlag =
        hasRejectedOrder;
    }
}

//Class User finishes here
```

Entity ValidityPeriod

```
@Entity
@NamedQuery(name = "ValidityPeriod.getOne",
    query = "SELECT vp FROM ValidityPeriod vp WHERE vp.id=?1")

public class ValidityPeriod {


    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private int monthDuration;
    private float price;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name ="servicePkgId")
    private ServicePackage servicePackage;

    @OneToMany(mappedBy="validityPeriod", fetch = FetchType.LAZY)
    private Set<Order> actualOrders;
}
```

Functional analysis of the interaction

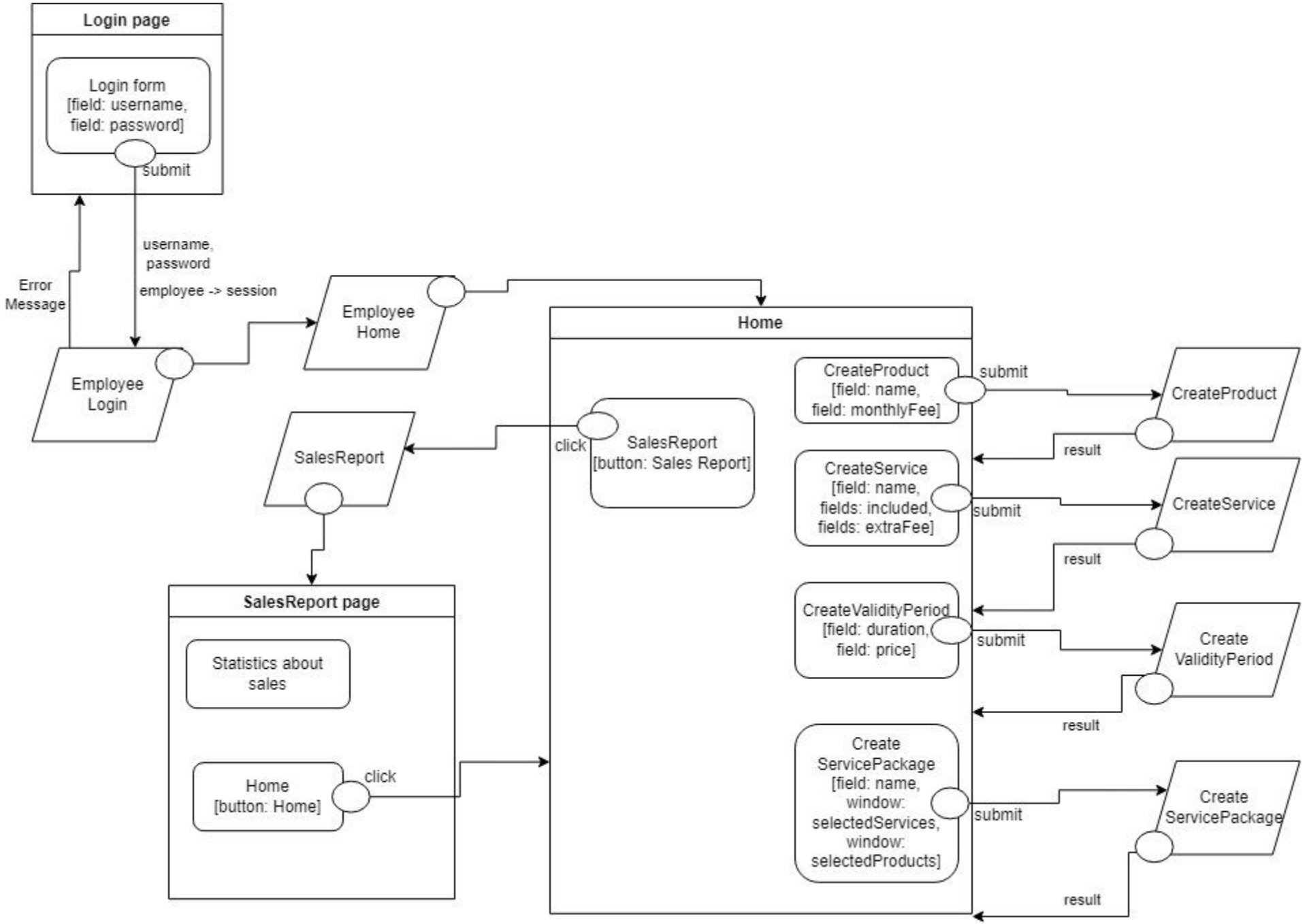
From every page the user can invoke the 'Logout' servlet that logs the user out of the session and restarts the navigation from the Login/Registration page



Customer Application

- The consumer application has a public LANDING page (LOGIN/REGISTRATION page) with a form(username, password) for login and a form(username, password, mail) for registration. Each form has a button, that allows to submit the form to the respective servlet. Click of the LOGIN button starts a credential verification and if they are correct this leads to the HOMEPAGE of the consumer application, to the LOGIN/REGISTRATION page otherwise. Click of REGISTRATION button creates and stores the new user, if the username is new, and when finished creates the LANDING page where the user can log in. From the LANDING page the user can click the browse button that allows him/her to see the available service packages list without logging in, by calling directly the GoToHomepage servlet that instantiates the HOMEPAGE.
- If the user has logged in, his/her username appears in the top right corner of all the application pages, above the LOGOUT button. Clicking LOGOUT button the application goes back to the LANDING page and the user is disconnected
- To get from the LANDING page to the HOMEPAGE the application calls the UserLogin servlet first, which assign the user to the session, then calls the GoToHomepage servlet that instantiates the actual HOMEPAGE.
- From the HOMEPAGE, the user can access a BUYSERVICE page to purchase a service package and thus creating a service subscription, by clicking the BUYSERVICE link. This click calls the BuyService servlet that creates the BUYSERVICE page. The BUYSERVICE page contains a form for purchasing a service package. The form allows the user to select one package from the list of available service packages and choose the validity period duration and the optional products to buy together with the chosen service. The form also allows the user to select the start date of his/her subscription. After choosing the service packages, the validity period and (0 or more) optional products, the user can press a CONFIRM button. This click takes to the OrderConfirmation servlet that add the defined order to the session and displays the CONFIRMATION page that summarizes the details of the chosen service package, the validity period, the optional products and the total price to be paid
- If the user has already logged in, the CONFIRMATION page displays a CANCEL button and one BUY button for each possible payment outcome. Clicking on CANCEL button cancels the operation on the order and takes back to the GoToHomepage servlet. Clicking the BUY button calls the OrderUploader servlet that stores the order and updates the user session variable, eventually calling the GoToHomepage servlet to restart the process.
- If the user has not logged in, the CONFIRMATION page displays a LOGIN/REGISTER Button to take the user to the LOGIN/REGISTRATION page. Clicking the button calls the OrderUploader servlet that sets a onlyLogin session variable and calls the GoToLogin servlet to create the LOGIN/REGISTRATION page. After either logging in or registering and immediately logging in, the UserLogin servlet calls the OrderConfirmation servlet to instantiate the CONFIRMATION page with all the details of the order, the CANCEL button and the BUY buttons.
- In the HOMEPAGE the user can also see the list of rejected orders. The user can select one of such orders and click the FIX button. This click calls the RejectedOrder servlet that adds a toFixOrder session variable and creates the CONFIRMATION page. From there the application flow is exactly the same as when creating a new order if logged in.

Pages (views), view components, events, actions


Login page
Login form
[field: username,
field: password]
submit
username,
password
Error
Message
employee -> session
Employee
Login
Employee
Home
Home
SalesReport
click
SalesReport
[button: Sales Report]
CreateProduct
[field: name,
field: monthlyFee]
submit
CreateProduct
result
CreateService
[field: name,
fields: included,
fields: extraFee]
submit
CreateService
result
CreateValidityPeriod
[field: duration,
field: price]
submit
Create
ValidityPeriod
result
Create
ServicePackage
[field: name,
window:
selectedServices,
window:
selectedProducts]
submit
Create
ServicePackage
result
SalesReport page
Statistics about
sales
Home
[button: Home]
click

Employee Application

- The employee application allows the authorized employees of the telco company to log in in the Login Page. Click of the Submit button starts a credential verification and if they are correct this leads to the Home Page of the employee application, to the Login page otherwise.
- In the Home page, a form allows the creation of service packages, with all the needed data and the possible optional products associated with them. Submitting will make the application check the data and create a new service package. The Home page lets the employee create optional products, services and validity periods as well. Each creation submitting will lead the applicaiton to check the data and create a new item.
- From the Home page, employee can click the link to access the Sales Report page, in which employee can see the essential data about the sales and about the users over the entire lifespan of the application.
  - Number of total purchases per package.
  - Number of total purchases per package and validity period.
  - Total value of sales per package with and without the optional products.
  - Average number of optional products sold together with each service package.
  - List of insolvent users, suspended orders and alerts.
  - Best seller optional product, i.e. the optional product with the greatest value of sales across all the sold service packages.
- In Sales Report page, employee can click the link to access the Home page.
- Pages (views), view components, events, actions

Components

Client tier

- **Servlets**
  - BuyService: creates the order (client side) that the user wants to buy
  - CreateProduct
  - CreateService
  - CreateServicePkg
  - CreateValidityPeriod
  - EmployeeHome: extracts the list of all available services/products/validity periods
  - EmployeeLogin: verifies credentials and store employee info in web session
  - GoToHomepage: extracts the list of service packages and rejected orders for the logged in user.
  - GoToLogin
  - Logout: invalidates the session
  - OrderConfirmation: allows the user to finalize order
  - OrderUploader: stores the order
  - RejectedOrder: handles the order to be fixed
  - SalesReport
  - UserLogin: verifies credentials and stores user info in web session
  - UserRegister
- **Html**
  - index.html: login form, registration form and link to employee
  - employee
    - home.html: forms for product/service/validity period/ service package creation
    - login.html: login form
    - salesreport.html: shows essential data about the sales and users
  - user
    - buyservice.html: displays the list of service packages and associated services/validity perios/ products
    - homepage.html: display all service packages, the list of rejected orders, logged user name and link to buy service
    - orderConfirmation.html: displays the order summary

Components

Business tier - Entities

- ActivationSchedule
- Auditing
- Employee
- OptProduct: optional product
- Order
- Service
- ServicePackage
- User
- ValidityPeriod

Components

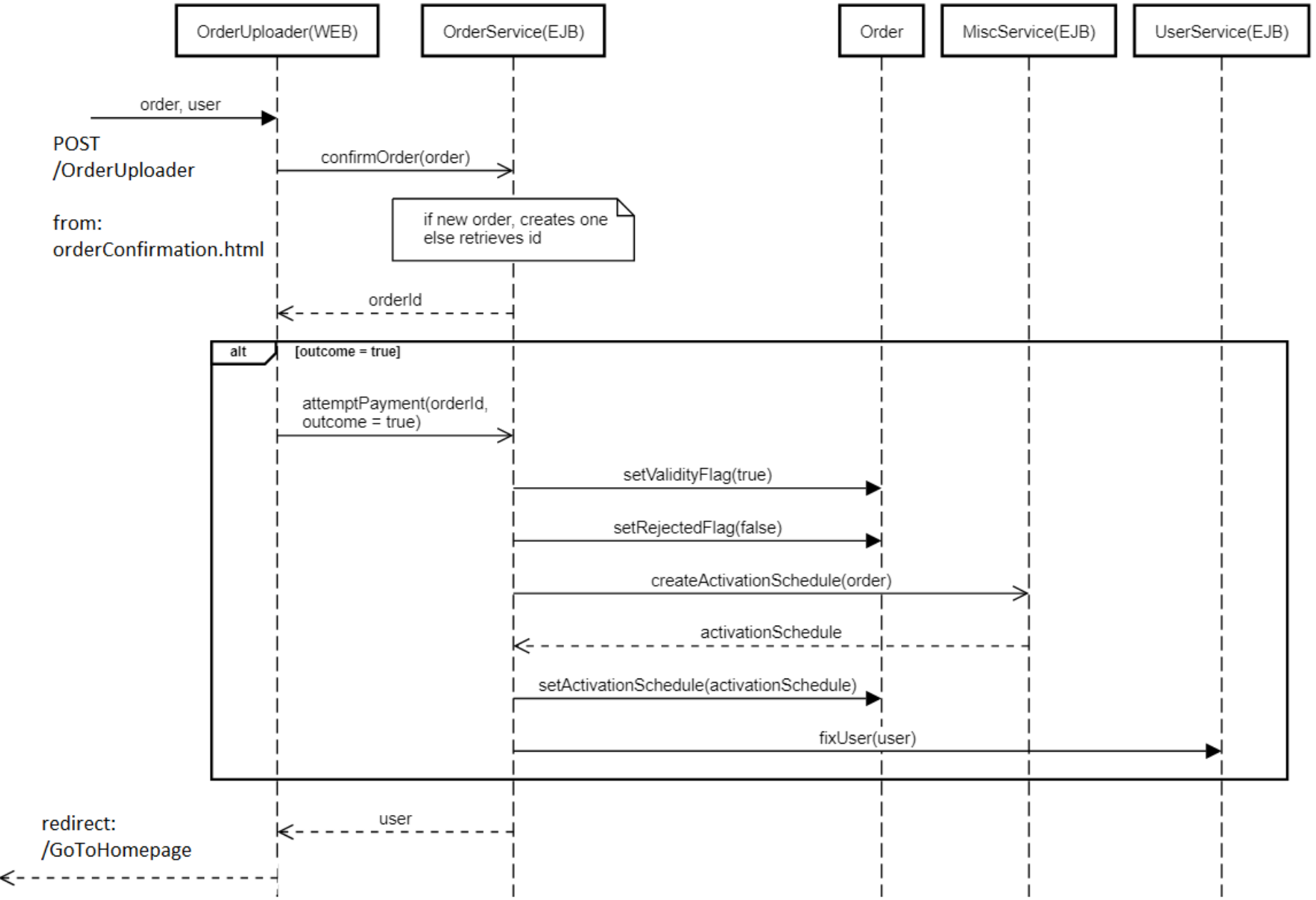
Business tier - Business Components (EJBs)

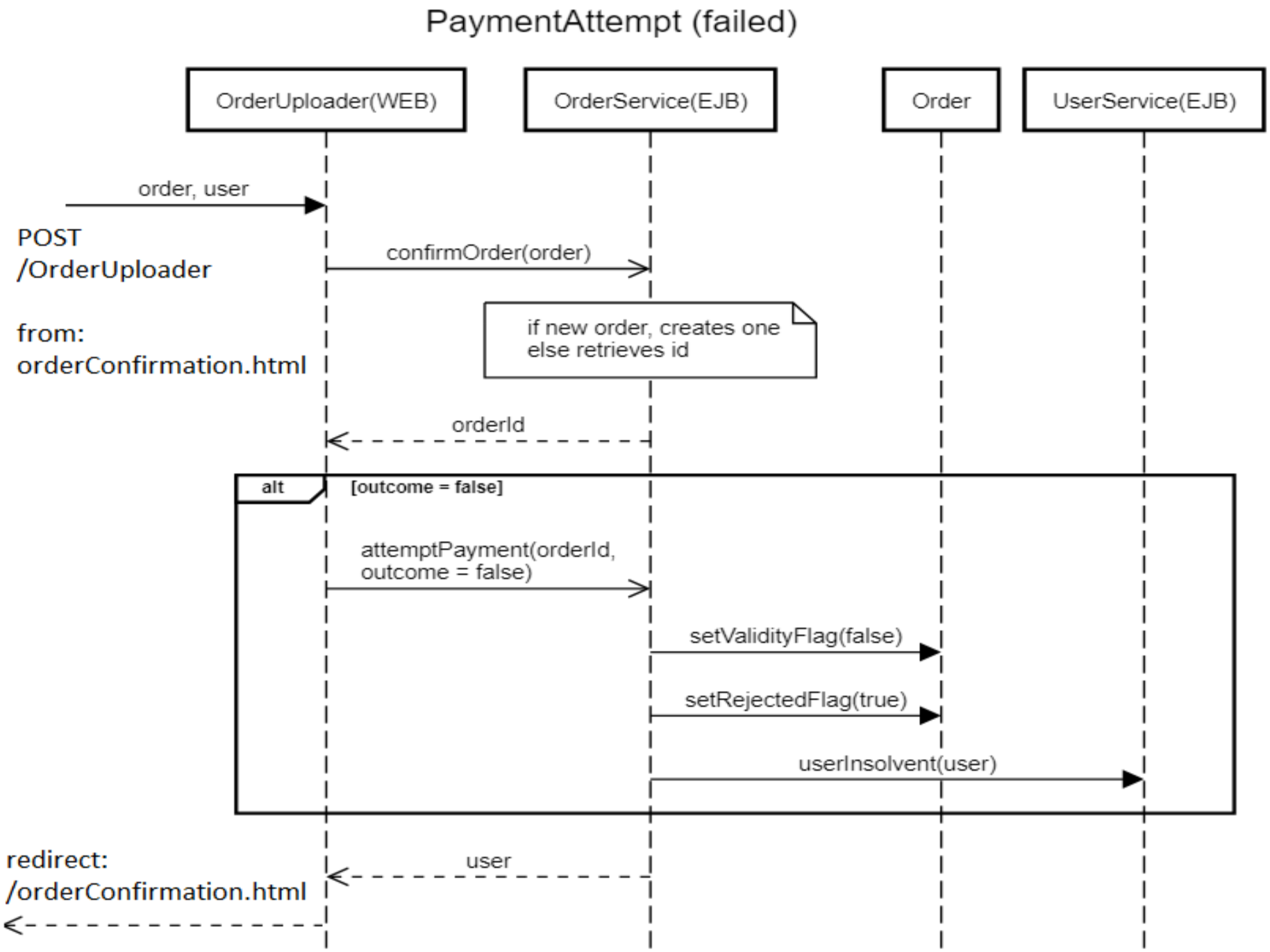
- @Stateless AuditingService
  - List<Auditing> getAllAuditings()
- @Stateless EmployeeService
  - Employee checkCredentials(String username, String password)
- @Stateless MiscService
  - ServicePackage retrieveServicePackage(**int** servicePackageId)
  - List<ServicePackage> findAllServicePackages()
  - ActivationSchedule createActivationSchedule(Order order)
  - **void** createAuditing(Order order, User user)
- @Stateless OrderService
  - Order getOrder(User user)
  - Order createOrder(User user, **int** chosenSP, **int** chosenVP, List<Integer> chosenOP, LocalDate startDate)
  - **int** confirmOrder(Order order)
  - User attemptPayment(**int** orderId, **boolean** activated)
  - List<Order> getAllSuspendedOrders()
- @Stateless SalesReportService
  - List getTotalPurchasePerSPandVP()
  - List getTotalPurchasePerSP()
  - List getAvgProdPerSP()
  - List getTotalSalesPerSPWithProd()
  - List getTotalSalesPerSPWithoutProd()
  - Object getBestSellerProduct()
- @Stateless ProductService
  - **void** createAProduct(String name, **float** fee)
- @Stateless ServicePkgService
  - **void** createAServicePkg(String name, **int**[] vpids, **int**[] sids, **int**[] pids)
- @Stateless ServiceService
  - Set<Service> findAllService()
  - **void** createAService(String name, **int** includedMin, **int** includedSMS, **int** includedGB, **float** extraMinFee, **float** extraSMSFee, **float** extraGBFee)
- @Stateless UserService
  - User checkCredentials(String usrn, String pwd)
  - **boolean** createUser(String username, String pwd, String email)
  - **void** userInsolvent(User user)
  - **void** fixUser(User user)
  - List<User> getInsolventUsers()
- @Stateless ValidityPeriodService
  - void createAValidityPeriod(**int** duration, **float** price)
  - Set<ValidityPeriod> findAllValidityPeriods()
  - Set<ValidityPeriod> findAllUnusedValidityPeriods()

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

PaymentAttempt (success)




PaymentAttempt (failed)
OrderUploader(WEB)
OrderService(EJB)
Order
UserService(EJB)
order, user
POST
/OrderUploader
confirmOrder(order)
from:
orderConfirmation.html
if new order, creates one
else retrieves id
orderId
alt
[outcome = false]
attemptPayment(orderId,
outcome = false)
setValidityFlag(false)
setRejectedFlag(true)
userInsolvent(user)
redirect:
/orderConfirmation.html
user