# Databases 2

Tommaso Brumani Andrea Alesani

POLITECNICO
MILANO 1863

# Index


- **Specifications**
  - Functional Analysis
- **Data Models**
  - ER Diagram
  - Logical Model
- **Trigger Design and Code**
  - Materialized Views
- **ORM Relationships**
- **Entities Code**
  - Materialized Views
- **List of Components**
- **UML Sequence Diagrams**

# Specifications (broad)

A telco company offers prepaid online services to web users.

Two client applications using the same database need to be developed.

# Functional Analysis

Pages (views)
View components
Actions
Events

## Consumer Application

The consumer application has a public Landing page with a form for login and a form for registration. Registration requires a username (which can be assumed as the unique identification parameter), a password and an email. Login leads to the Home page of the consumer application. Registration leads back to the landing page where the user can log in.

The user can log in before browsing the application or browse it without logging in. If the user has logged in, his/her username appears in the top right corner of all the application pages.

The Home page of the consumer application displays the service packages offered by the telco company.

A service package has an ID and a name (e.g., “Basic”, “Family”, “Business”, “All Inclusive”, etc). It comprises one or more services. Services are of four types: fixed phone, mobile phone, fixed internet, and mobile internet. The mobile phone service specifies the number of minutes and SMSs included in the package plus the fee for extra minutes and the fee for extra SMSs. The fixed phone service has no specific configuration parameters. The mobile and fixed internet services specify the number of Gigabytes included in the package and the fee for extra Gigabytes. A service package must be associated with one validity period. A validity period specifies the number of months (12, 24, or 36). Each validity period has a different monthly fee (e.g., 20€/month for 12 months, 18€/month for 24 months, and 15€ /month for 36 months). A package may be associated with one or more optional products (e.g., an SMS news feed, an internet TV channel, etc.). The validity period of an optional product is the same as the validity period that the user has chosen for the service package. An optional product has a name and a monthly fee independent of the validity period duration. The same optional product can be offered in different service packages.

# Functional Analysis

Pages (views)
View components
Actions
Events

## Consumer Application

From the Home page, the user can access a Buy Service page for purchasing a service package and thus creating a service subscription. The Buy Service page contains a form for purchasing a service package. The form allows the user to select one package from the list of available ones 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. The application displays a CONFIRMATION page that summarizes the details of the chosen service package, the validity period, the optional products and the total price to be pre-paid: (monthly fee of service package * number of months) + (sum of monthly fees of options * number of months).

If the user has already logged in, the CONFIRMATION page displays a BUY button. If the user has not logged in, the CONFIRMATION page displays a link to the login page and a link to the REGISTRATION page. After either logging in or registering and immediately logging in, the CONFIRMATION page is redisplayed with all the confirmed details and the BUY button.

When the user presses the BUY button, an order is created. The order has an ID and a date and hour of creation. It is associated with the user and with the service package, its validity period and the chosen optional products. It also contains the total value (as in the CONFIRMATION page) and the start date of the subscription. After creating the order, the application bills the customer by calling an external service. If the external service accepts the billing, the order is marked as valid and a service activation schedule is created for the user. A service activation schedule is a record of the services and optional products to activate for the user with their date of activation and date of deactivation.

If the external service rejects the billing, the order is put in the rejected status and the user is flagged as insolvent. When an insolvent user logs in, the home page also contains the list of rejected orders. The user can select one of such orders, access the CONFIRMATION page, press the BUY button and attempt the payment again. When the same user causes three failed payments, an alert is created in a dedicated auditing table, with the user Id, username, email, and the amount, date and time of the last rejection.

# Functional Analysis

Pages (views)
View components
Actions
Events

## Employee Application

The employee application allows the authorized employees of the telco company to log in. 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. The same page lets the employee create optional products as well.

A Sales Report page allows the employee to inspect 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.

# Specifications Interpretation

- We assumed that the same website could host both **consumer** and **employee** applications, forcing the client to **login** in order to access **restricted pages** (users have an attribute which determines their level of access).

- We considered an **employee** as a **normal user** with some **more privileges**. Therefore, an employee **can perform** every **customer action** in addition to viewing reports and creating packages.

# On Entities and Views

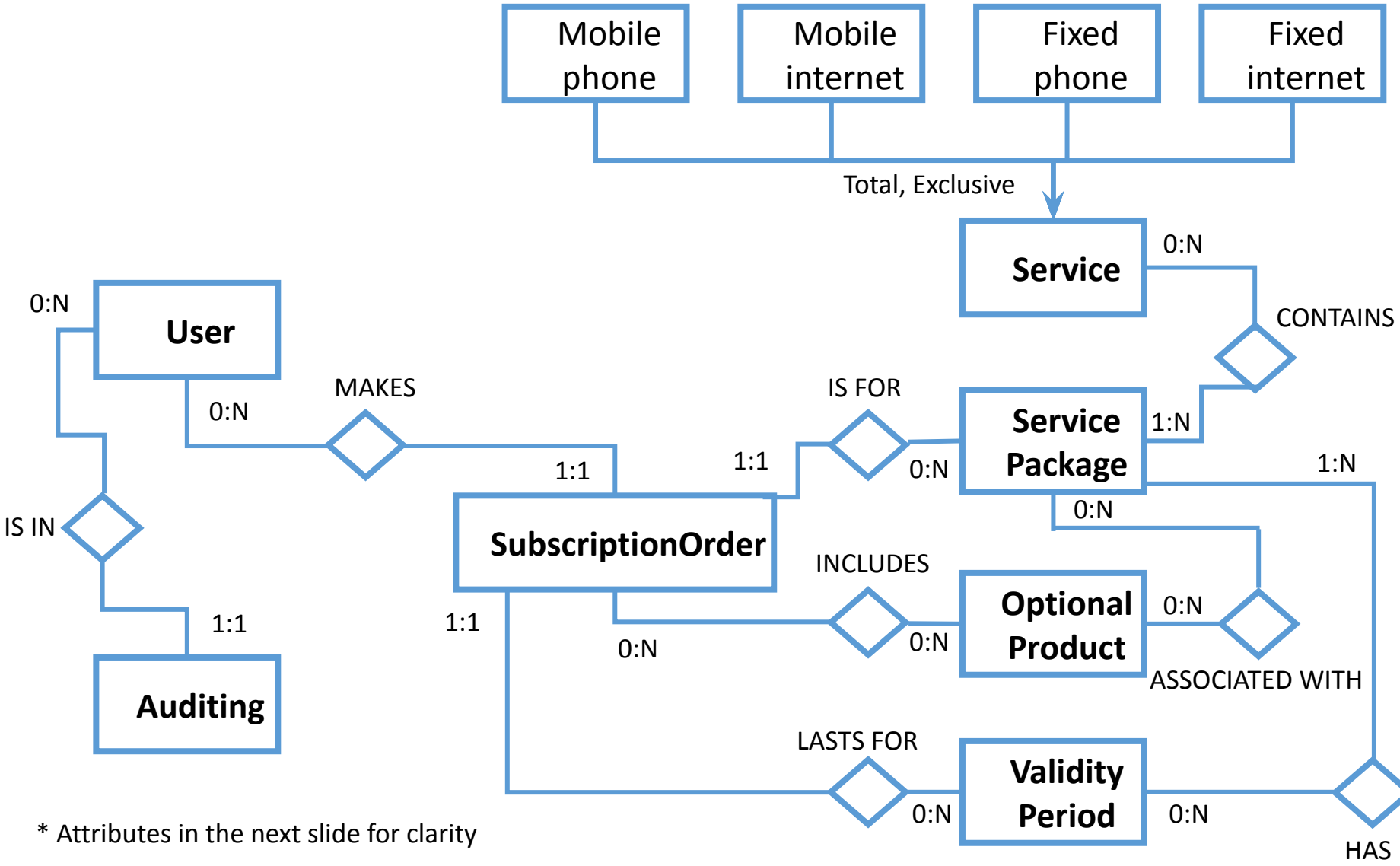
In the following sections we will treat **Entities** and **Materialized Views** as different concepts, and will describe them in **different sections**.

Materialized Views had to be implemented using Tables and Entities, as MySql does not support regular Materialized Views by default.

The tables that represent Materialized Views always present a **foreign primary key** (which is the primary key of a different table), and are populated exclusively through **triggers**.

An exceptional case is that of the **Auditing** table, which despite being initially thought of, and treated as, a regular Entity, is nevertheless populated solely through triggers like a Materialized View.

# Entity Relationship Schema




* Attributes in the next slide for clarity

# Entities Attributes

### User

- id
- username (varchar)
- password (varchar)
- email (varchar)
- insolvent (bit)
- rejected_payments (int)
- employee (bit)

### Auditing

- id
- rejection_ts (datetime)
- rejected_amount (decimal)

### SubscriptionOrder

- id
- creation_ts (datetime)
- valid (bool)
- start_date_ts (datetime)
- total_value (decimal)

### Validity Period

- id
- months (int)
- monthly_fee (decimal)

### Optional Product

- id
- name (varchar)
- monthly_fee (decimal)

### Service

- id
- type (varchar)
- gb (int)
- extra_gb_fee (decimal)
- minutes (int)
- extra_min_fee (decimal)
- sms (int)
- extra_sms_fee (decimal)

### Service Package

- id
- name (varchar)

# Motivations of the ER design (1)

- We assumed that a **single order** can purchase one **single service package**, and that if the user desires more than one service package they can make two different purchases.

- We decided that the **total value** of an **order** would be useful to have **stored in the database**, despite the fact that it could be calculated from other attributes and entities (we chose to **assign its value using triggers**).

- We decided to **group all service types** into one **single entity** made up of optional attributes. We used triggers to ensure that certain attributes have non-zero value only for specific service types.

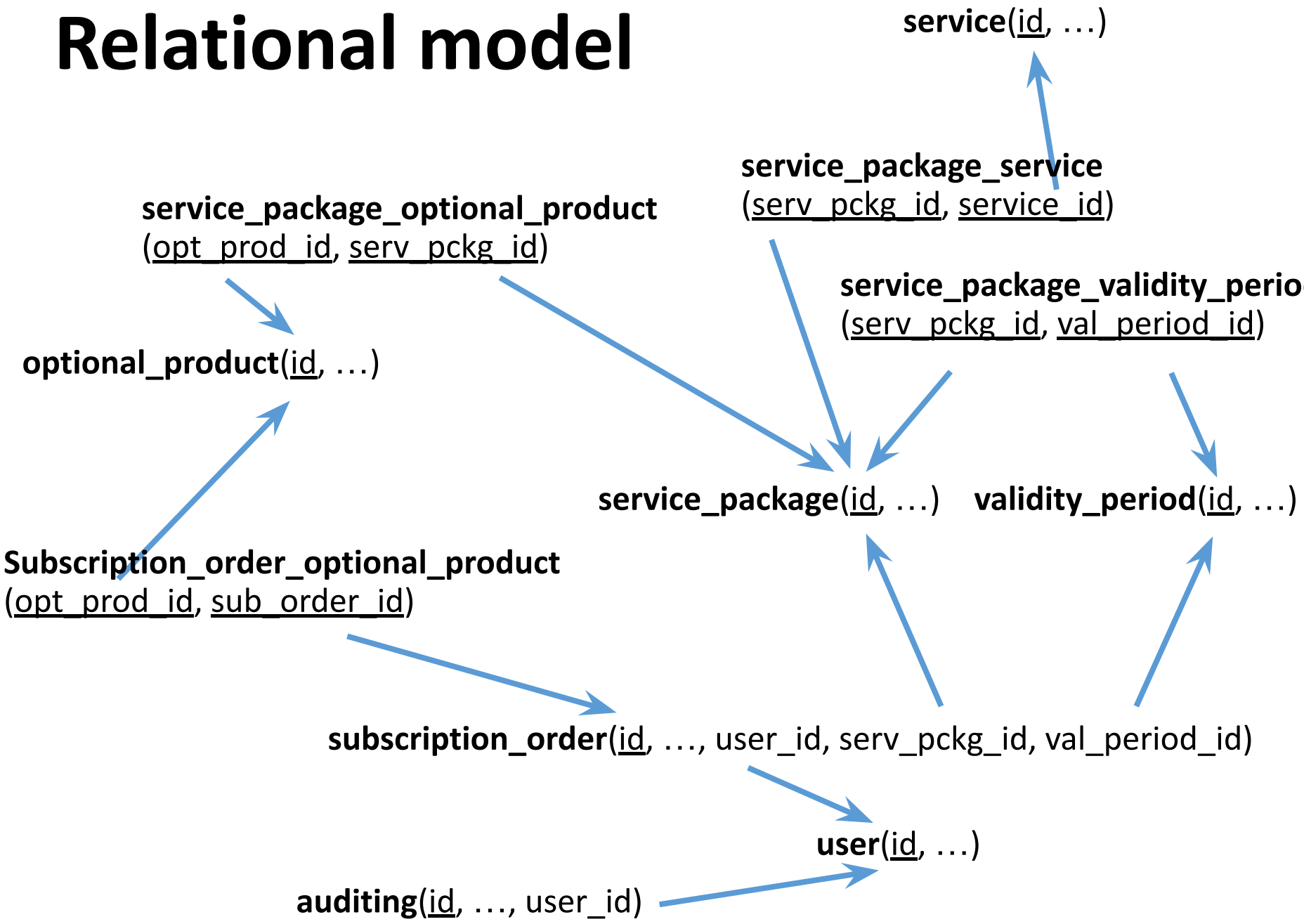
# Motivations of the ER design (2)

- We decided to associate **Service Packages** with **Validity Periods** and **Optional Products** through Many-To-Many relationships to determine which could be chosen when creating a new **Order**.

- We decided to keep track of whether a **User** is **insolvent**, and of the number of **rejected payments** since the last alert generation, through attributes in the Entity. These attributes are **kept** in a **consistent** state **through triggers**.

- **Materialized Views** were **omitted** as they will be described at a later section.

# Relational model



# Motivations of the logical design

- In order to **improve** the diagram's **readability**, we **omitted attributes** that do not function as either primary or foreign keys (they can be seen in the entity diagram already).

- **Associative tables** were **instantiated automatically** by JPA to map Many-To-Many relationships.

- **Materialized Views** were again **omitted** as they will be described in a later section.

# Trigger Design and Code

Because MySql Workbench does **not allow multiple triggers** to be defined on the **same event and table**, all triggers had to be **grouped** based on those factors in the actual code implementation (for example, all “AFTER UPDATE” triggers for table “subscription_order” had to be put inside the same trigger).

Beyond “regular” triggers, some triggers were specifically used to populate and update Materialized View tables: these are described in a s**eparate section**.

# SERVICE_TYPE_CONSTRAINTS_BEFORE_INSERT

- **Event**:

  Before insert into SERVICE

- **Condition**:

  Based on TYPE, if the wrong attributes have non-zero values

- **Action**:

  ROLLBACK

- **SQL code**:

```
CREATE TRIGGER `SERVICE_TYPE_CONSTRAINTS_BEFORE_INSERT`
BEFORE INSERT ON `service`
FOR EACH ROW
BEGIN
        IF      (new.`type` = "MOBILE_PHONE" OR new.`type`="FIXED_PHONE")
                AND (new.gb != 0 OR new.extra_gb_fee != 0.0)
        THEN    SIGNAL sqlstate '45001' set message_text = "This type of service cannot offer internet navigation!";
        END IF;
        IF      (new.`type` = "FIXED_PHONE" OR new.`type` = "FIXED_INTERNET" OR new.`type` = "MOBILE_INTERNET")
                AND (new.minutes != 0 OR new.extra_min_fee != 0.0)
        THEN    SIGNAL sqlstate '45001' set message_text = "This type of service cannot offer phone call time!";
        END IF;
        IF      (new.`type` = "FIXED_PHONE" OR new.`type` = "FIXED_INTERNET" OR new.`type` = "MOBILE_INTERNET")
                AND (new.sms != 0 OR new.extra_sms_fee != 0.0)
        THEN    SIGNAL sqlstate '45001' set message_text = "This type of service cannot offer sms messages!";
         END IF;
END
```

# SET_CREATION_DATE_BEFORE_INSERT

- **Event**:

  Before insert into SUBSCRIPTION_ORDER

- **Action**:

  Set CREATION_TS to NOW()

- **SQL code**:

```
CREATE TRIGGER `SET_CREATION_DATE_BEFORE_INSERT`
BEFORE INSERT ON `subscription_order`
FOR EACH ROW
BEGIN
        SET new.creation_ts = now();
END
```

- **Motivation**:

  This trigger was created to ensure the creation date of an order is always accurate.

# SET_SERVICE_TOTAL_VALUE_BEFORE_INSERT

- **Event**:

  Before insert on SUBSCRIPTION_ORDER

- **Action**:

  Set TOTAL_VALUE to (VAL_PERIOD_ID.MONTHLY_FEE * VAL_PERIOD_ID.MONTHS)

- **SQL code**:

```
CREATE TRIGGER `SET_SERVICE_TOTAL_VALUE_BEFORE_INSERT`
BEFORE INSERT ON `subscription_order`
FOR EACH ROW
BEGIN
        SET     new.total_value =
                ((      SELECT monthly_fee
                        FROM validity_period
                        WHERE id = new.val_period_id )
                *
                (       SELECT months
                        FROM validity_period
                        WHERE id = new.val_period_id ));
END
```

- **Motivation**:

  This trigger is the first of 2 used to compute an order's TOTAL_VALUE. Since JPA inserts the order's optional products associations into the associative table only after completing the insert into this table, the trigger adding their value to the total has been placed under the subscription_order_optional_product table (seen in the following slide).

# SET_PRODUCTS_TOTAL_VALUE_AFTER_INSERT

- **Event**:

  After insert on SUBSCRIPTION_ORDER_OPTIONAL_PRODUCT

- **Action**:

  Set TOTAL_VALUE on SUBSCRIPTION_ORDER with ID equal to SUB_ORDER_ID to

  SUB_ORDER_ID.TOTAL_VALUE + (MONTHLY_FEE * SUB_ORDER_ID.VAL_PERIOD_ID.MONTHS)

- **SQL code**:

```
CREATE TRIGGER `SET_PRODUCTS_TOTAL_VALUE_AFTER_INSERT`
AFTER INSERT ON `subscription_order_optional_product`
FOR EACH ROW
BEGIN
        SET     @vp_id =
                (       SELECT val_period_id
                        FROM subscription_order
                        WHERE id = new.sub_order_id  );
        SET     new.total_value =
                ((      SELECT monthly_fee
                        FROM validity_period
                        WHERE id = @vp_id )
                *
                (       SELECT months
                        FROM validity_period
                        WHERE id = @vp_id     ));
END
```

# SET_USER_INSOLVENT_AFTER_UPDATE

- **Event**:

  After update on SUBSCRIPTION_ORDER

- **Condition**:

  If new.VALID = 0

- **Action**:

  Set INSOLVENT = 1 to the USER with ID equal to USER_ID

- **SQL code**:

```
CREATE TRIGGER `SET_USER_INSOLVENT_AFTER_UPDATE`
AFTER UPDATE ON `subscription_order`
FOR EACH ROW
BEGIN
        IF      new.valid = 0
        THEN    UPDATE `user`
                SET insolvent = 1
                WHERE id = new.user_id;
        END IF;
END
```

# SET_USER_SOLVENT_AFTER_UPDATE

- **Event**:

  After update on SUBSCRIPTION_ORDER

- **Condition**:

  If new.VALID = 1 and for all other SUBSCRIPTION_ORDERs belonging to USER with ID USER_ID: VALID != 0

- **Action**:

  Set USER_ID.INSOLVENT = 0

- **SQL code**:

```
CREATE TRIGGER `SET_USER_SOLVENT_AFTER_UPDATE`
AFTER UPDATE ON `subscription_order`
FOR EACH ROW
BEGIN
	IF	new.valid = 1
		AND NOT EXISTS (	SELECT *
					FROM subscription_order
					WHERE valid = 0 AND user_id = new.user_id	)
	THEN	UPDATE `user`
		SET insolvent = 0
		WHERE id = new.user_id;
	END IF;
END
```

# INCREASE_REJECTED_PAYMENTS_AFTER_UPDATE

- **Event**:

  After update on SUBSCRIPTION_ORDER

- **Condition**:

  If new.VALID = 0 and USER_ID.REJECTED_PAYMENTS < 2

- **Action**:

  Set USER_ID.REJECTED_PAYMENTS = USER_ID.REJECTED_PAYMENTS + 1

- **SQL code**:

```
CREATE TRIGGER `INCREASE_REJECTED_PAYMENTS_AFTER_UPDATE`
AFTER UPDATE ON `subscription_order`
FOR EACH ROW
BEGIN
        IF      new.valid = 0
                AND ( SELECT rejected_payments
                FROM `user`
                WHERE id = new.user_id ) < 2
        THEN    UPDATE `user`
                SET rejected_payments = rejected_payments +1
                WHERE id = new.user_id;
        END IF;
END
```

- **Motivation**:

  This trigger is tasked with increasing a User's rejected payments counter up to the number 2. When the number 3 is reached, a different trigger is fired, which brings the counter back down to zero and creates a record in the AUDITING table (see next slide).

# GENERATE_AUDITING_AFTER_UPDATE

- **Event**:

  After update on SUBSCRIPTION_ORDER

- **Condition**:

  If new.VALID = 0 and USER_ID.REJECTED_PAYMENTS >= 2

- **Action**:

  Set USER_ID.REJECTED_PAYMENTS = 0 and create tuple in AUDITING

- **SQL code**:

```
CREATE TRIGGER `SUBSCRIPTION_ORDER_AFTER_UPDATE`
AFTER UPDATE ON `subscription_order`
FOR EACH ROW
BEGIN
        IF      new.valid = 0
                AND ( SELECT rejected_payments
                FROM `user`
                WHERE id = new.user_id ) >= 2
        THEN    UPDATE `user`
                SET rejected_payments = 0
                WHERE id = new.user_id;

                INSERT INTO auditing (rejected_amount, rejection_ts, user_id)
                VALUES(new.total_value, now(), new.user_id);
        END IF;
END
```

# Materialized Views

In the following section are described the **Materialized Views** developed for the application.

The records in these tables are **inserted**, **updated** and **deleted** through **triggers**, and they always present "sales_report" at the beginning of their name, as well as **foreign primary keys** which reference the application's other entities' primary keys.

They will be listed describing the composition of their tables, as well as the code of the triggers used to populate them.

# SALES_REPORT_PACKAGES

- **TABLE DESCRIPTION**:

  This materialized view contains several sales statistics regarding each of the application's service packages. Records are inserted with all values set to zero immediately upon creation of the service package, and are subsequently updated each time a new order is made for that package.

  - serv_pckg_id: the ID of the referenced SERVICE_PACKAGE (foreign primary key)
  - purchases (int): number of times the package has been purchased
  - sales_value_no_products (decimal): total amount of sales without counting the associated optional products
  - sales_value_with_products (decimal): total amount of sales counting optional products
  - avg_num_products (decimal): average number of products sold with the service package

- **INSERT**:

```
AFTER INSERT ON `service_package`
FOR EACH ROW
BEGIN
        INSERT INTO sales_report_optional_product
                (purchases, sales_value_no_products, sales_value_with_products, avg_num_products)
        VALUES (0, 0.0, 0.0, 0);
END
```

# SALES_REPORT_PACKAGES

- **UPDATE**:

```
AFTER UPDATE ON `subscription_order`
FOR EACH ROW
BEGIN
        IF
                new.valid = 1 AND  (old.valid IS null OR old.valid = 0)
        THEN
                UPDATE sales_report_packages AS srp
                SET     srp.purchases = srp.purchases + 1,
                        srp.sales_value_no_prod = srp.sales_value_no_prod + ( SELECT months*monthly_fee
                                                                              FROM validity_period AS vp
                                                                              WHERE vp.id = new.val_period_id ),
                        srp.sales_value_with_products = srp.sales_value_with_products + new.total_value,
                        srp.avg_num_products =          (SELECT coalesce(avg(package.prod_count), 0)
                                                         FROM (
                                                                SELECT count(*) AS prod_count
                                                                FROM subscription_order_optional_product AS soop
                                                                JOIN subscription_order AS so
                                                                ON so.id = soop.sub_order_id
                                                                WHERE so.serv_pckg_id = new.serv_pckg_id
                                                                AND so.valid = 1
                                                                GROUP BY so.id          )          package          )
                WHERE srp.id = new.serv_pckg_id;
        END IF;
END
```

# SALES_REPORT_VALIDITY_PACKAGES

- **TABLE DESCRIPTION**:

  This materialized view contains sales statistics for each validity period when associated with a specific service package. Records are inserted upon creation of the package, and subsequently updated with each sale.

  - serv_pckg_id: the ID of the referenced SERVICE_PACKAGE (foreign composite primary key)
  - Val_period_id: the ID of the referenced VALIDITY_PERIOD (foreign composite primary key)
  - purchases (int): the amount of times the service package/validity period combination has been purchased

- **INSERT**:

```
AFTER UPDATE ON `service_package_validity_period`
FOR EACH ROW
BEGIN
        INSERT INTO sales_report_validity_packages (serv_pckg_id, val_period_id, purchases)
        VALUES (new.serv_pckg_id, new.val_period_id, 0);
END
```

- **UPDATE**:

```
AFTER UPDATE ON `service_package_validity_period`
FOR EACH ROW
BEGIN
        IF      new.valid = 1
        THEN    UPDATE sales_report_validity_packages
                SET purchases = purchases + 1
                WHERE serv_pckg_id = new.serv_pckg_id
                AND val_period_id = new.val_period_id;
        END IF;
END
```

# SALES_REPORT_INSOLVENT_USERS

- **TABLE DESCRIPTION**:

  This materialized view contains the IDs of the users who are currently flagged as insolvent.
  Records are added when the INSOLVENT flag of a user is set to 1, and removed when it is set back to 0.

  - user_id: the ID of the referenced USER (foreign primary key)

- **INSERT**:

```
AFTER UPDATE ON `user`
FOR EACH ROW
BEGIN
        IF      new.insolvent != old.insolvent
                AND new.insolvent = 1
        THEN    INSERT INTO sales_report_insolvent_users (user_id)
                VALUES (new.id);
        END IF;
END
```

- **DELETE**:

```
AFTER UPDATE ON `user`
FOR EACH ROW
BEGIN
        IF      new.insolvent != old.insolvent
                AND new.insolvent = 0
        THEN    DELETE FROM sales_report_insolvent_users
                WHERE user_id = new.id;
        END IF;
END
```

# SALES_REPORT_SUSPENDED_ORDERS

- **TABLE DESCRIPTION**:

  This materialized view contains the IDs of the orders for which the payment has failed.
  As orders are initially inserted with the VALID flag set to NULL, and the flag is updated with positive or negative value based on the result of the payment, the update of this flag is used to discriminate between when a record should be added or removed.

  - sub_order_id: the ID of the referenced SUBSCRIPTION_ORDER (foreign primary key)

- **INSERT**:

```
AFTER UPDATE ON `subscription_order`
FOR EACH ROW
BEGIN
        IF      new.valid = 0 AND (old.valid IS null OR old.valid = 1)
        THEN    INSERT INTO sales_report_suspended_orders (sub_order_id)
                VALUES (new.id);
        END IF;
END
```

- **DELETE**:

```
AFTER UPDATE ON `subscription_order`
FOR EACH ROW
BEGIN
        IF      new.valid = 1 AND (old.valid IS null OR old.valid = 0)
        THEN    DELETE FROM sales_report_suspended_orders
                WHERE sub_order_id = new.id;
        END IF;
END
```

# SALES_REPORT_PRODUCT_SALES

- **TABLE DESCRIPTION**:

  This materialized view contains the total sales amount for each OPTIONAL_PRODUCT.
  Records are added with the value set to zero upon creation of the product, and subsequently updated with each purchase.
  The table is queried by the application only to extract its best seller product, but the table keeps records of all products for convenience of update.

  - opt_prod_id: the ID of the referenced OPTIONAL_PRODUCT (foreign primary key)

- **INSERT**:

```
AFTER UPDATE ON `optional_product`
FOR EACH ROW
BEGIN
        INSERT INTO sales_report_product_sales (opt_prod_id, total_sales)
        VALUES (new.id, 0);
END
```

# SALES_REPORT_PRODUCT_SALES

- **UPDATE**:

```
AFTER INSERT ON `subscription_order_optional_product`
FOR EACH ROW
BEGIN
        UPDATE sales_report_product_sales
        SET total_sales = total_sales + (       SELECT vp.months
                                                FROM validity_period AS vp
                                                JOIN subscription_order AS so
                                                ON vp.id = so.val_period_id
                                                WHERE so.id = new.sub_order_id )
                                        *
                                        (       SELECT monthly_fee
                                                FROM optional_product
                                                WHERE id = new.opt_prod_id )
        WHERE opt_prod_id = new.opt_prod_id;
END
```

# ORM Design

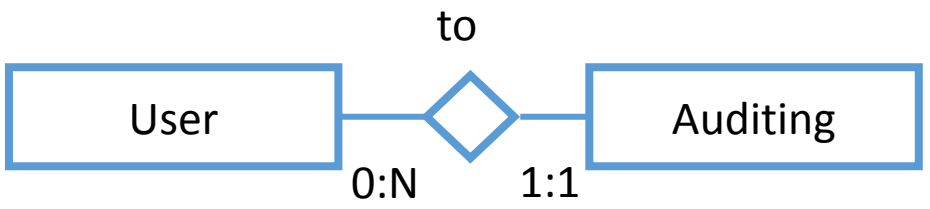
- When approaching the ORM design of the application, we decided to **map all sides of all relationships**, in order to make it easier to visualize and navigate them both in the Java code and through triggers.

- **Cascading policy** was **always** left to the **NONE** default, as the necessity to cascade persistence operations to an Entity's attributes never presented itself.

- Fetch policy was set to **EAGER** in those instances where an Entity would **usually** be **displayed along** some of its **attributes**.

- Materialized Views were omitted (no mappings).

# Relationship “IS IN”





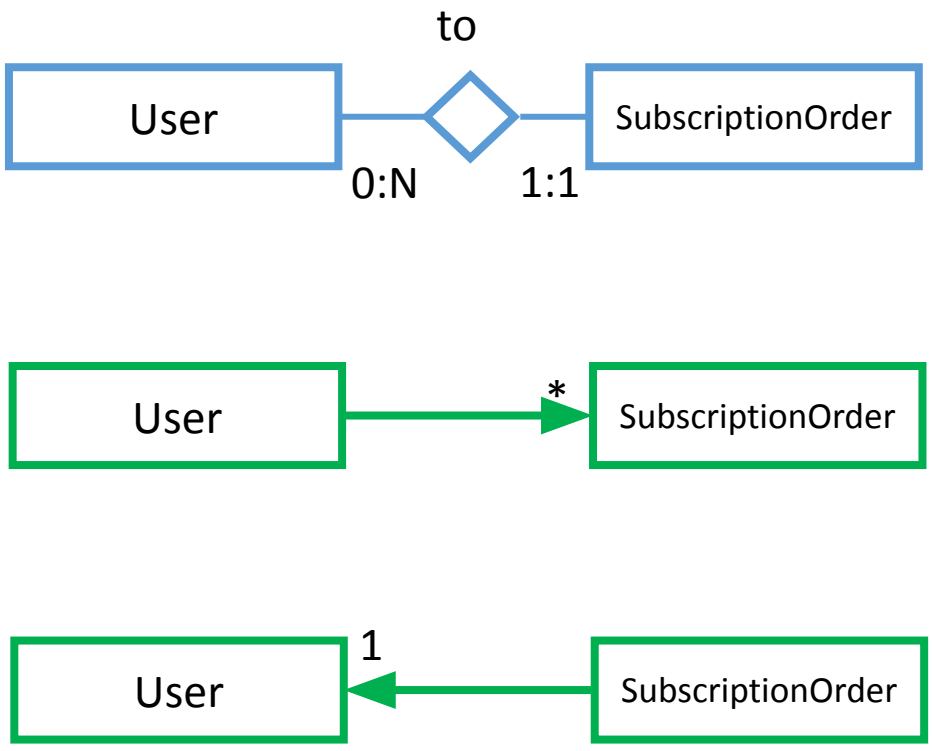

**User ➔ Auditing**

@OneToMany (LAZY fetch, cascade NONE): not necessary, but mapped and not used for simplicity

**Auditing ➔ User**

@ManyToOne (EAGER fetch, cascade NONE): necessary to show information for the audited User

# Relationship “MAKES”



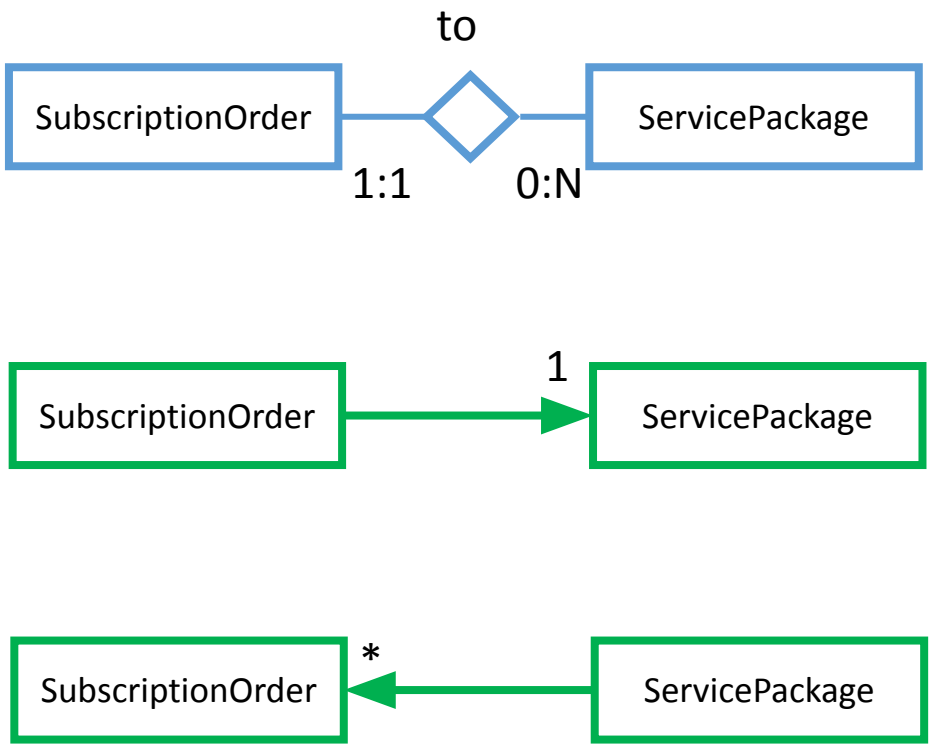

**User ➔ SubscriptionOrder**

@OneToMany (LAZY fetch, cascade NONE): needed to show the User’s unpaid Orders (kept LAZY to avoid always fetching all orders when logging in).

**SubscriptionOrder ➔ User**

@ManyToOne (LAZY fetch, cascade NONE): not necessary, but mapped and not used for simplicity

# Relationship “IS FOR”



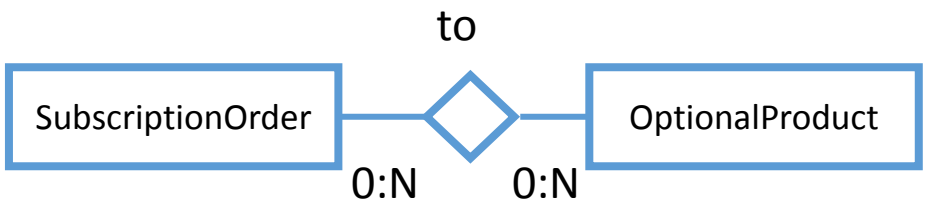

**SubscriptionOrder ➔ ServicePackage**

@ManyToOne (EAGER fetch, cascade NONE): needed to show the Order’s Service Package.

**ServicePackage ➔ SubscriptionOrder**

@OneToMany (LAZY fetch, cascade NONE): not necessary, but mapped and not used for simplicity

# Relationship “INCLUDES”





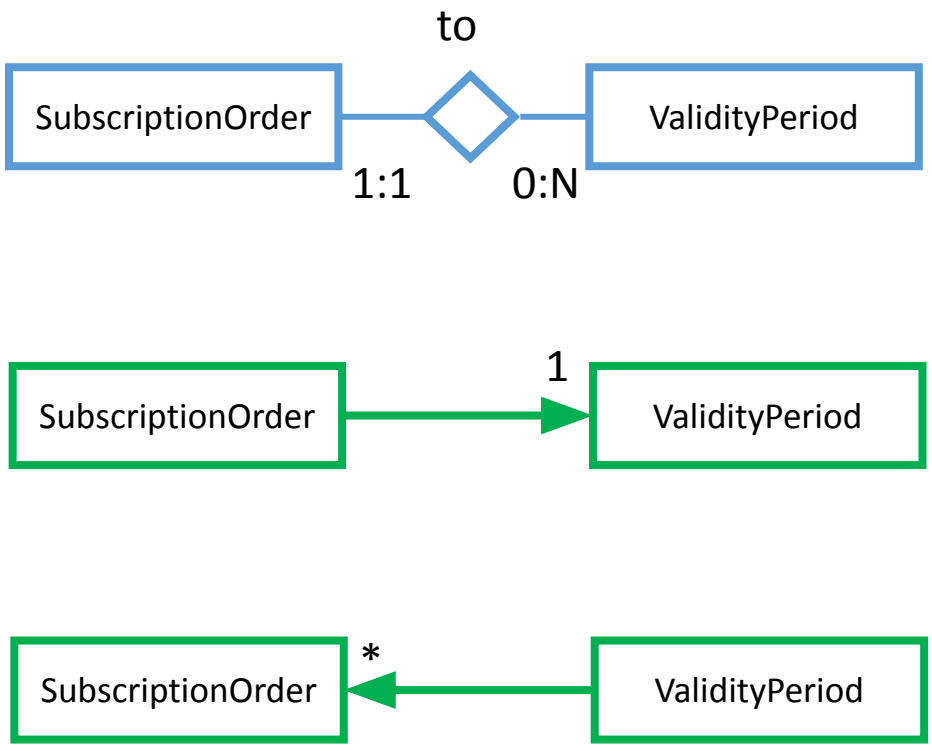

**SubscriptionOrder ➔ OptionalProduct**

@ManyToMany (EAGER fetch, cascade NONE): needed to show the Order’s selected Optional Products.

**OptionalProduct ➔ SubscriptionOrder**

@ManyToMany (LAZY fetch, cascade NONE): not necessary, but mapped and not used for simplicity

# Relationship “LASTS FOR”



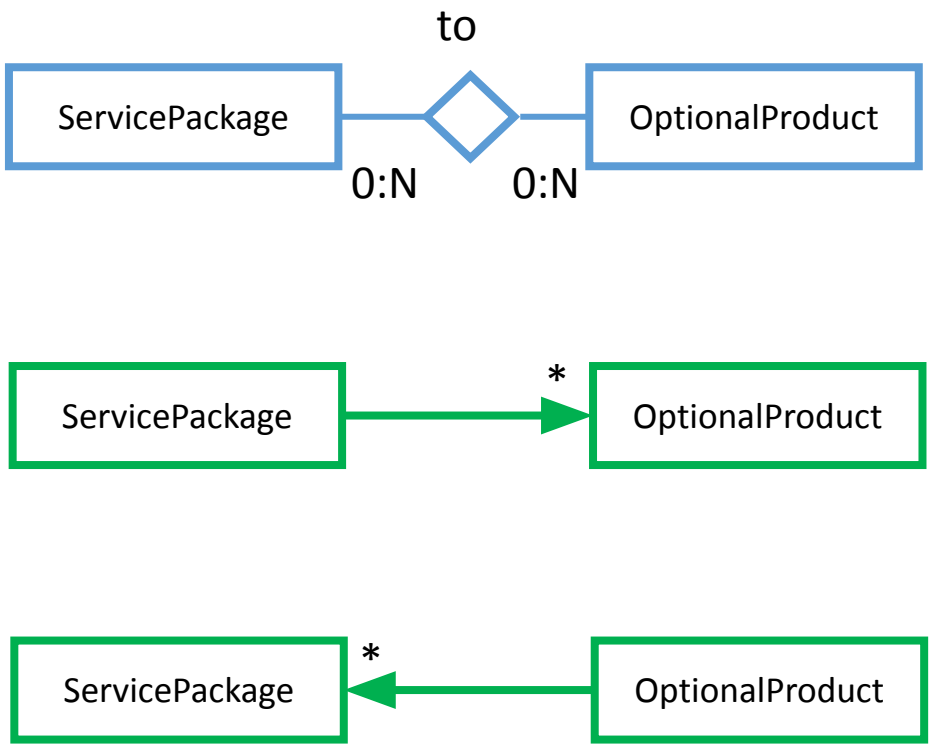

**SubscriptionOrder ➔ ValidityPeriod**

@ManyToOne (EAGER fetch, cascade NONE): needed to show the Order’s Validity Period.

**ValidityPeriod ➔ SubscriptionOrder**

@OneToMany (LAZY fetch, cascade NONE): not necessary, but mapped and not used for simplicity

# Relationship “ASSOCIATED WITH”



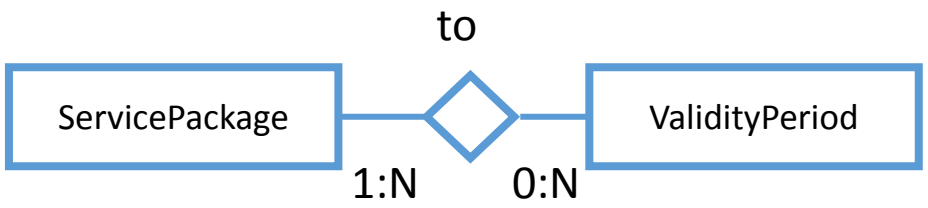

**ServicePackage ➔ OptionalProduct**

@ManyToMany (EAGER fetch, cascade NONE): needed to show the Package’s associated Optional Products.

**OptionalProduct ➔ ServicePackage**

@ManyToMany (LAZY fetch, cascade NONE): not necessary, but mapped and not used for simplicity

# Relationship “HAS”





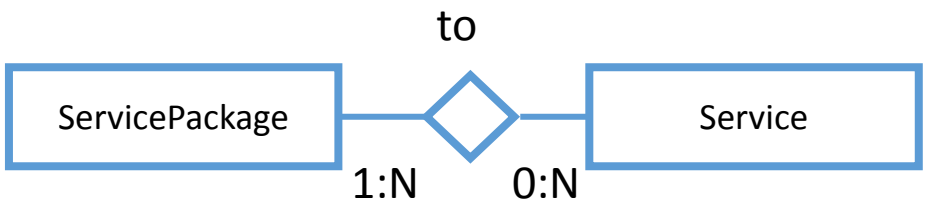

**ServicePackage → ValidityPeriod**

@ManyToMany (EAGER fetch, cascade NONE): needed to show the Package’s associated Validity Periods.

**ValidityPeriod → ServicePackage**

@ManyToMany (LAZY fetch, cascade NONE): not necessary, but mapped and not used for simplicity

# Relationship “CONTAINS”






**ServicePackage ➔ Service**

@ManyToMany (EAGER fetch, cascade NONE): needed to show the Package’s associated Services.

**Service ➔ ServicePackage**

@ManyToMany (LAZY fetch, cascade NONE): not necessary, but mapped and not used for simplicity

# Entities Code

**Named queries** were employed to perform **operations** that are **impossible** or onerous to achieve **through** the **ORM mappings**.

These were added to the top corresponding Entity's code for clarity.

Entities used to access Materialized Views were included for completeness.

# Entity User

```
@Entity
@Table(name = "user")
@NamedQueries({
      @NamedQuery(name = "User.checkCredentials", query =
            "SELECT r FROM User r  WHERE r.username = ?1 and r.password = ?2"),
      @NamedQuery(name = "User.existsUsername", query =
            "SELECT r FROM User r  WHERE r.username = ?1")})

public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    @Column(nullable = false, unique = true)
    private String username;
    @Column(nullable = false)
    private String password;
    @Column(nullable = false, unique = true)
    private String email;
    @Column(nullable = false)
    private boolean insolvent;
    @Column(name = "rejected_payments", nullable = false)
    private int rejectedPayments;
    @Column(name = "employee", nullable = false)
    private boolean employee;

    @OneToMany(
            fetch = FetchType.LAZY,
            mappedBy = "user"
    )
    private List<Auditing> auditing;
    @OneToMany(
            fetch = FetchType.EAGER,
            mappedBy = "user"
    )
    private List<SubscriptionOrder> orders;
}
```

# Entity Auditing

```
@Entity
@Table(name = "auditing")
@NamedQueries({
      @NamedQuery(name = "Auditing.findAllAuditings", query =
            "SELECT a FROM Auditing a")})

public class Auditing {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @Column(
            nullable = false,
            name = "rejection_ts"
    )
    private Timestamp rejectionTs;
    @Column(
            nullable = false,
            name = "rejected_amount"
    )
    private BigDecimal rejectedAmount;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(
            nullable = false,
            name = "user_id"
    )
    private User user;
}
```

# Entity Optional Product

```
@Entity
@Table(name = "optional_product")
@NamedQueries({
      @NamedQuery(name = "OptionalProduct.findAllOptionalProducts", query =
            "SELECT op FROM OptionalProduct op"),
      @NamedQuery(name = "OptionalProduct.findNumOptionalProducts", query =
            "SELECT count(op) FROM OptionalProduct op")})

public class OptionalProduct {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @Column(nullable = false)
    private String name;
    @Column(
            nullable = false,
            name = "monthly_fee"
    )
    private BigDecimal monthlyFee;

    @ManyToMany(
            fetch = FetchType.LAZY,
            mappedBy = "optionalProducts"
    )
    private Set<ServicePackage> servicePackages;
    @ManyToMany(
            fetch = FetchType.LAZY,
            mappedBy = "optionalProducts"
    )
    private Set<SubscriptionOrder> subscriptionOrder;
}
```

# Entity Service

```
@Entity
@Table(name = "service")
@NamedQueries({
      @NamedQuery(name = "Service.findAllServices", query =
            "SELECT s FROM Service s"),
      @NamedQuery(name = "Service.findNumServices", query =
            "SELECT count(s) FROM Service s")})

public class Service {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    @Enumerated(EnumType.STRING)
    @Column(nullable = false)
    private ServiceType type;
    @Column
    private int gb;
    @Column(name = "extra_gb_fee")
    private BigDecimal extraGbFee;
    @Column
    private int minutes;
    @Column(name = "extra_min_fee")
    private BigDecimal extraMinFee;
    @Column
    private int sms;
    @Column(name = "extra_sms_fee")
    private BigDecimal extraSmsFee;

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

# Entity ServicePackage

```
@Entity
@Table(name = "service_package")
@NamedQueries({
      @NamedQuery(name = "ServicePackage.findAllServicePackages", query =
            "SELECT sp FROM ServicePackage sp")})

public class ServicePackage {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    @Column(nullable = false)
    private String name;

    @ManyToMany(
            fetch = FetchType.EAGER
    )
    @JoinTable(
            joinColumns = @JoinColumn(name = "serv_pckg_id"),
            inverseJoinColumns = @JoinColumn(name = "service_id"))
    private Set<Service> services;
    @ManyToMany(
            fetch = FetchType.EAGER
    )
    @JoinTable(
            joinColumns = @JoinColumn(name = "serv_pckg_id"),
            inverseJoinColumns = @JoinColumn(name = "val_period_id")
    )
    private Set<ValidityPeriod> validityPeriods;
    @ManyToMany(
            fetch = FetchType.EAGER
    )
    @JoinTable(
            joinColumns = @JoinColumn(name = "serv_pckg_id"),
            inverseJoinColumns = @JoinColumn(name = "opt_prod_id")
    )
    private Set<OptionalProduct> optionalProducts;
    @OneToMany(
            fetch = FetchType.LAZY,
            mappedBy = "servicePackage"
    )
    private Set<SubscriptionOrder> subscriptionOrders;
}
```

# Entity SubscriptionOrder

```
@Entity
@Table(name = "subscription_order")
@NamedQueries({
      @NamedQuery(name = "SubscriptionOrder.makePayment", query =
            "UPDATE SubscriptionOrder so SET so.valid = ?2, so.user = ?3 WHERE so.id = ?1")})

public class SubscriptionOrder {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @Column(name = "total_value")
    private BigDecimal totalValue;
    @Column(nullable = false,name = "creation_ts")
    private Timestamp creationTs;
    @Column(name = "start_date_ts")
    private Timestamp startDateTs;
    @Column()
    private Boolean valid;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(
            nullable = false,
            name = "serv_pckg_id"
    )
    private ServicePackage servicePackage;
    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(
            nullable = false,
            name = "val_period_id"
    )
    private ValidityPeriod validityPeriod;
    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(
            joinColumns = @JoinColumn(name = "sub_order_id"),
            inverseJoinColumns = @JoinColumn(name = "opt_prod_id")
    )
    private Set<OptionalProduct> optionalProducts;
    @ManyToOne(fetch = FetchType.LAZY, cascade = CascadeType.PERSIST)
    @JoinColumn(
            nullable = false,
            name = "user_id"
    )
    private User user;

}
```

# Entity ValidityPeriod

```
@Entity
@Table(name = "validity_period")
@NamedQueries({
      @NamedQuery(name = "ValidityPeriod.findAllValidityPeriods", query =
            "SELECT vp FROM ValidityPeriod vp"),
      @NamedQuery(name = "ValidityPeriod.findNumValidityPeriods", query =
            "SELECT count(vp) FROM ValidityPeriod vp")})

public class ValidityPeriod {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    @Column(nullable = false)
    private int months;
    @Column(
            nullable = false,
            name = "monthly_fee"
    )
    private BigDecimal monthlyFee;

    @ManyToMany(
            fetch = FetchType.LAZY,
            mappedBy = "validityPeriods"
    )
    private Set<ServicePackage> servicePackages;
    @OneToMany(
            fetch = FetchType.LAZY,
            mappedBy = "validityPeriod"
    )
    private Set<SubscriptionOrder> subscriptionOrder;
}
```

# Entity SalesReportInsolventUsers (Materialized View)

```
@Entity
@Table(name = "sales_report_insolvent_users")
@NamedQueries({
      @NamedQuery(name = "SalesReportInsolventUsers.findAllInsolvent", query =
            "SELECT sriu FROM SalesReportInsolventUsers sriu")})

public class SalesReportInsolventUsers {

    @Id
    @Column(
            nullable = false,
            name = "user_id"
    )
    private int userId;

    public int getUserId() {
        return userId;
    }
}
```

# Entity SalesReportPackages (Materialized View)

```
@Entity
@Table(name = "sales_report_packages")
@NamedQueries({
      @NamedQuery(name = "SalesReportPackages.findAllSalesReports", query =
            "SELECT srp FROM SalesReportPackages srp")})

public class SalesReportPackages {

    @Id
    @Column(
            nullable = false,
            name = "serv_pckg_id"
    )
    private int servPckgId;

    @Column(
            nullable = false
    )
    private int purchases;
    @Column(
            nullable = false,
            name = "sales_value_no_products")
    private BigDecimal salesValueNoProducts;
    @Column(
            nullable = false,
            name = "sales_value_with_products")
    private BigDecimal salesValueWithProducts;
    @Column(
            nullable = false,
            name = "avg_num_products")
    private BigDecimal avgNumProducts;
}
```

# Entity SalesReportProductSales (Materialized View)

```
@Entity
@Table(name = "sales_report_product_sales")
@NamedQueries({
        @NamedQuery(name = "SalesReportProductSales.findBestSeller", query =
                "" +
                "SELECT srps " +
                "FROM SalesReportProductSales srps " +
                "WHERE srps.totalSales = (" +
                "SELECT MAX(srps1.totalSales)" +
                "FROM SalesReportProductSales srps1" +
                ")")})

public class SalesReportProductSales {

    @Id
    @Column(
            nullable = false,
            name = "opt_prod_id"
    )
    private int optProdId;

    @Column(
            nullable = false,
            name = "total_sales"
    )
    private BigDecimal totalSales;
}
```

# Entity SalesReportSuspendedOrders (Materialized View)

```
@Entity
@Table(name = "sales_report_suspended_orders")
@NamedQueries({
      @NamedQuery(name = "SalesReportSuspendedOrders.findAllSuspended", query =
            "SELECT srso FROM SalesReportSuspendedOrders srso")})

public class SalesReportSuspendedOrders {

    @Id
    @Column(
            nullable = false,
            name = "sub_order_id"
    )
    private int subOrderId;
}
```

# Entity SalesReportValidityPackages (Materialized View)

```
@Entity
@Table(name = "sales_report_validity_packages")
@IdClass(SalesReportValidityPackagesId.class)
@NamedQueries({
      @NamedQuery(name = "SalesReportValidityPackages.findAllSalesReports", query =
            "SELECT srvp FROM SalesReportValidityPackages srvp")})

public class SalesReportValidityPackages {

    @Id
    @Column(
            nullable = false,
            name = "serv_pckg_id"
    )
    private int servPckgId;

    @Id
    @Column(
            nullable = false,
            name = "val_period_id"
    )
    private int valPeriodId;

    @Column(
            nullable = false
    )
    private int purchases;
}


public class SalesReportValidityPackagesId implements Serializable {

    private int servPckgId;
    private int valPeriodId;

    @Override
    public int hashCode() {
        return super.hashCode();
    }

    @Override
    public boolean equals(Object obj) {
        return super.equals(obj);
    }
}
```

# Components

## Views (Client tier)

- activation-schedule.html
- buy-service.html
- confirmation.html
- creation-result.html
- employee-home.html
- home.html
- payment-result.html
- sales-report.html

## Servlets (Web tier)

- CreateOptionalProduct.java
- CreateServicePackage.java
- GoToBuyServicePage.java
- GoToConfirmationPage.java
- GoToEmployeeHomePage.java
- GoToHomePage.java
- GoToLoginPage.java
- GoToSalesReportPage.java
- GoToSchedulePage.java
- InvalidPayment.java
- Login.java
- PrepareOrder.java
- RegisterNewUser.java
- RetryOrder.java
- ValidPayment.java

# Components

## EJBs (Business tier)

- AuditingService (@stateless)
  - List<Auditing> findAllAuditings()
- OptionalProductService (@stateless)
  - List<OptionalProduct> findAllOptionalProducts()
  - OptionalProduct findOptionalProductById(int id)
  - int findNumOptionalProducts()
  - OptionalProduct createOptionalProduct(String name, BigDecimal monthlyFee)
- ServicePackageService (@stateless)
  - List<ServicePackage> findAllServicePackages()
  - ServicePackage findServicePackageById(int id)
  - ServicePackage createServicePackage(String name, List<Integer> serviceIDs, List<Integer> validityPeriodIDs, List<Integer> optionalProductIDs)
- ServiceService (@stateless)
  - List<Service> findAllServices()
  - Service findServiceById(int id)
- SubscriptionOrderService (@stateless)
  - SubscriptionOrder findSubscriptionOrderById(int id)
  - void makePayment(int id, boolean valid, int userId)
  - SubscriptionOrder createOrder(int servicePackageID, int validityPeriodID, List<Integer> optionalProductIDs, Timestamp creationTs, Timestamp startDateTs)
- UserService (@stateless)
  - User findUserById(int id)
  - List<SubscriptionOrder> getUserOrders(int id)
  - User checkCredentials(String username, String password)
  - void registerNewUser(String username, String password, String email)
- ValidityPeriodService (@stateless)
  - List<ValidityPeriod> findAllValidityPeriods()
  - ValidityPeriod findValidityPeriodById(int id)
  - int findNumValidityPeriods()

# Components

## EJBs (Materialized Views)

- SalesReportInsolventUsersService (@stateless)
  - List<SalesReportInsolventUsers> findAllInsolvent()
- SalesReportPackagesService (@stateless)
  - List<SalesReportPackages> findAllSalesReports()
- SalesReportProductSalesService (@stateless)
  - SalesReportProductSales findBestSeller()
- SalesReportSuspendedOrdersService (@stateless)
  - List<SalesReportSuspendedOrders> findAllSuspended()
- SalesReportValidityPackagesService (@stateless)
  - List<SalesReportValidityPackages> findAllSalesReports()

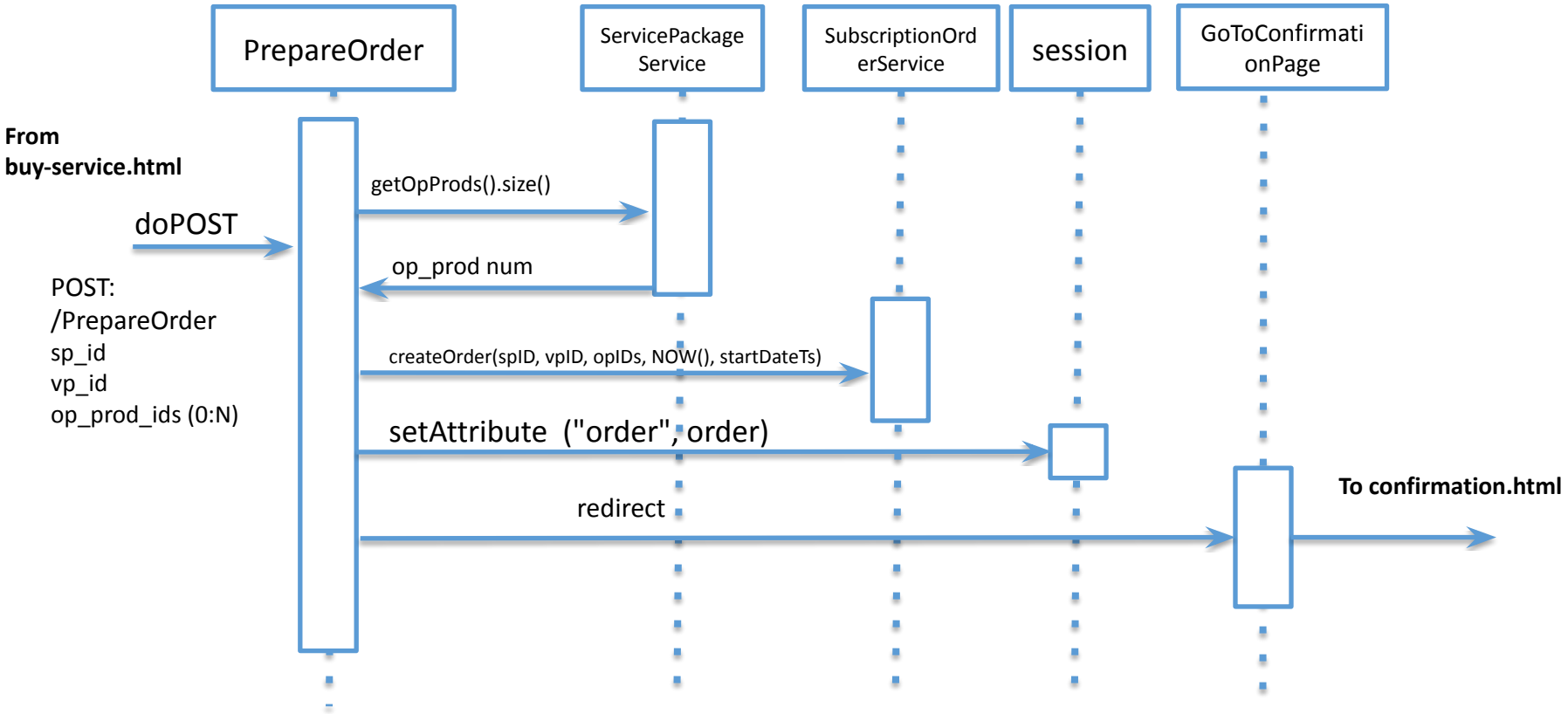
## Entities (Data tier)

- Auditing.java
- OptionalProduct.java
- Service.java
- ServicePackage.java
- SubscriptionOrder.java
- User.java
- ValidityPeriod.java

## Entities (Materialized Views)

- SalesReportInsolventUsers.java
- SalesReportPackages.java
- SalesReportProductSales.java
- SalesReportSuspendedOrders.java
- SalesReportValidityPackages.java

# From **buy-service.html** to **confirmation.html**




**NB** The first interaction with ServicePackageService is needed because the user can choose an arbitrary number of optional products (from 0 to the total number of products associated with the specific service package they chose) for one single order, so we need to know the maximum number of parameters they might have sent to successfully parse the request
**NB** The created order has the 'valid' attribute set to NULL (it will take a non-NULL value after the first payment attempt)