Salespoint 5

Technical Overview

Christopher Bellmann, Thomas Dedek, Stanley Förster,
Paul Henke, Hendrik Neubert, Hannes Weisbach

6th October 2011

Dresden University of Technology
Faculty of Computer Science
Institute of Software- and Multimedia- Technology
Professorship of Software Technology

Address
Faculty of Computer Science
Insitute of Software and Multimedia Technology
TU Dresden
01062 Dresden

Telephone: 0351 463 38442
Fax: 0351 463 38459
birgit.demuth@tu-dresden.de
http://tu-dresden.de

Contents

List of Figures

List of Tables

1 Introduction

The Salespoint Framework is intended to minimize developing effort of e-commerce solutions and point-of-sale applications. Salespoint 2010 users complained about complexity, missing features and bugs. Thus, the decision was made to re-design and re-implement the framework from scratch. Our development goal was an easy-to-use framework primarily targeted for educational purposes. As such, Salespoint 5 is not taylored to any specific application, but designed with a wide area of applications in mind.

Models and design patterns employed in Salespoint 5 are inspired by "Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML" by Jim Arlow [AN03]. An overview of the functionality of and new features in Salespoint 5 is detailed in this document.

We would like to thank all Salespoint users who submitted their feedback and encourage future users of Salespoint 5 to do the same.

2 Technical Background

One of the main reason to use a framework such as Salespoint for educational purposes is to teach students reusability. The Salespoint 5developers also adhere to that principle. Thus, Salespoint 5itself uses a number of frameworks and APIs, which are introduced briefly.

2.1 JPA - Java Persistence API

One of the key features of Salespoint 5is its integrated persistence layer. The goal is to allow data persistence, while minimising programming effort and training period as well as maximising flexibility for framework users.

The obvious choice was the Java Persistence API (JPA), a Java framework managing relational data in Java SE or EE applications. Salespoint 5uses JPA 2.0, developed under JSR 317 and finalised in Dec, 2009.

Additional to the API itself, which is defined in the `javax.persistence` package, JPA also consists of *Persistence Entities*, *ORM Metadata* and the *Java Persistence Query Language* (JPQL).

A persistence entity is usually a *Plain Old Java Object* (POJO), which is mapped to a single table in a database. A row in such a database table corresponds to a specific instance of such an entity. Relational data between entities (and therefor tables) may be specified in an XML descriptor file or as annotations in Java source code. Salespoint 5 uses annotations to provide object/relational metadata.

A query language similar to SQL - JPQL - is used to retrieve entity information from the database. In contrast to SQL, JPQL queries act on entity objects instead of database tables. JPA Implementations translate a JPQL statement to SQL statements at run time, thus it is possible to replace the DBMS, while keeping the Java classes. It is possible to interface directly with the DBMS using *Native Queries*. Salespoint 5 however, uses the *Criteria API* to facilitate type safe querying.

Multiple implementations of JPA 2.0 exist, for example TopLink [top], EclipseLink [ecl]. The open source persistence and ORM framework Hibernate [hib] also supports JPA 2.0. Salespoint 5uses the JPA 2.0 reference implementation, EclipseLink. No implementation specific code is used in Salespoint 5, therefore it should be possible to interchange EclipseLink with another JPA implementation.

2.2 Joda Time

Joda Time [jod] is a Java date and time API. It provides a quality replacement for the Java date and time classes. Salespoint 5incorporates Joda Time, because it is open

source, easy to use and offers better performance characteristics than Java date and time classes.

Key concepts from Joda Time used in Salespoint 5 are *Instant*, *Interval*, *Duration* and *Period*. From Joda Time documentation:

> The most frequently used concept in Joda-Time is that of the *instant*. An Instant is defined as *an instant in the datetime continuum specified as a number of milliseconds from 1970-01-01T00:00Z*. This definition of milliseconds is consistent with that of the JDK in `Date` or `Calendar`. Interoperating between the two APIs is thus simple.

Intervals are defined by two Instants. An Interval is *half-open*, that is to say the start is inclusive but the end is not. Also, the end is always greater or equal than the start.

Durations in Joda Time represent a duration in time, exact to the milisecond. Durations can be thought of as length of an Interval.

Periods also represent durations in time, but in a more abstract way. A Period may be a month, which may be 28, 29, 30 or 31 days, thus the length in miliseconds of those periods differ.

If you, for example have an Instant of February, 1st and add a Period of one month, the result will be an Instant of March, 1st. Adding instead a Period of 30 days to an Instant of February, 1st will result in an Instant of March, 2nd or March 3rd, depending if the Instant is in a leap year or not.

2.3 Spring

In contrast to earlier versions of the Salespoint Framework, Salespoint 5 obeys the MVC pattern. Salespoint 5 can be seen as the Model of an MVC application, no parts of the View or the Controller are incorporated in the Framework.

Salespoint 5 is designed as basis for development of web applications, using the *Spring Framework* [spr] to implement Views and Controllers. To further ease the development, Salespoint 5 includes property editors to convert string based representations to Salespoint 5 identifier types. Also, JSP tags to check if a user is logged in and if a user has a certain capability are included in the framework.

3 Salespoint 5 Components

Integrating a persistence layer into the Salespoint 5 Framework had a great influence on some design decisions made during development of Salespoint 5. Early on it became obvious that necessities of JPA could dictate the design and implementations of Salespoint 5.

To guard against the influence of JPA requirements on design decisions, Salespoint 5 strongly follows the *programming against interfaces* programming style. Although, creating an interface for almost every class violates the *KISS* principle, the developers deemed programming against interfaces necessary because Salespoint 5 is intrinsically tied to JPA. Using interfaces allowed us to cleanly define the behaviour of an object, without relying on a specific implementation.

Objects, which need to be persisted to safe the current state of an application, are called persistence entities. Usually, a persistence entity is a *Plain Old Java Object* (POJO). Every persistence entity class is an implementation of a corresponding Java interface. Each persistence entity has a manager class, which in turn is an implementation of a manager interface. The specific manager implementations in Salespoint 5 facilitate persisting the objects to a database. However, it is possible to implement every persistence entity and manager class in Salespoint 5 non-persistent, for example collection-based.

Salespoint 5 is designed to be developer-friendly. A crucial part of its easy-to-use feel is the consistency in interfaces, persistence entities and managers across the framework, including, but not limited to naming of methods and behaviour of managers.

Figure 3.1 depicts the package structure of Salespoint 5. Salespoint 5 components are grouped into packages according to their respective functionality. Key concepts of Salespoint 5 are illustrated in the following paragraphs.

3.1 Shop

`Shop` is a central class in Salespoint 5; it holds references to all manager interfaces and a reference to the `Time` interface. There are six manager interfaces in Salespoint 5: Accountancy, Calendar, Catalog, Inventory, OrderManager and UserManager. Other classes use the `Shop` to access the manager interfaces, for example `Order.completeOrder()` uses `Shop.INSTANCE.getInventory()` for product removal. `PersistentCalendar` uses `Shop.INSTANCE.getTime()` for time based operations. There is also a convenience method to minimize boilerplate code

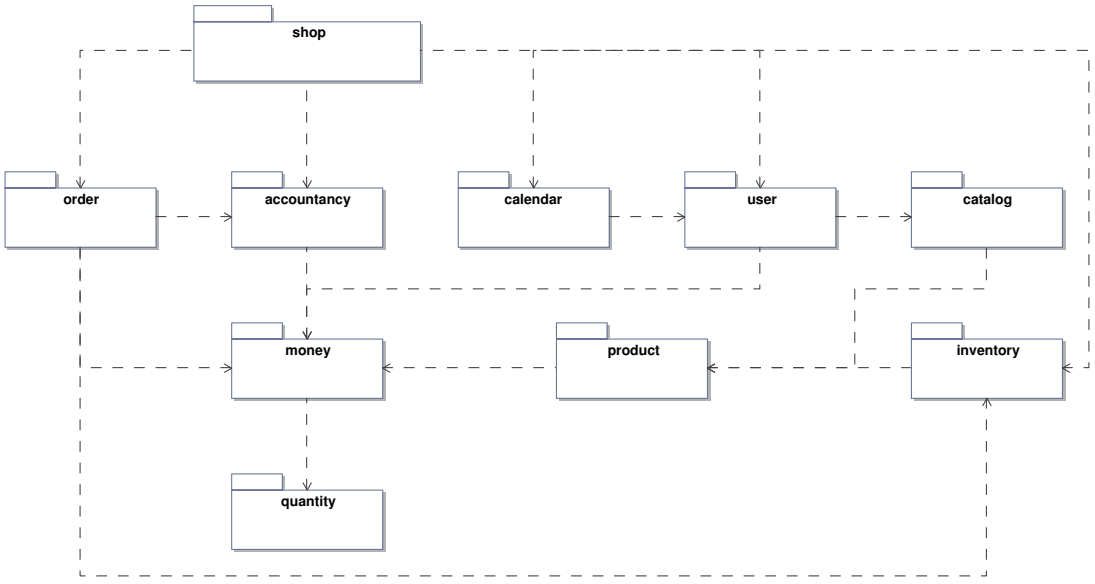

Figure 3.1: Package Overview

`Shop.initializeShop()`; it is used for setting all managers of `Shop` to Salespoints persistent class implementations and the time to `DefaultTime`. `Shop` is implemented as singleton.

3.2 User

To manage system accounts, Salespoint 5 has a notion of a user in the form of the `User` interface. Users are managed by the `UserManager`, which is also an interface. The implementing classes handling the persistence aspects are `PersistentUser` and `PersistentUserManager`, respectively. Every user is uniquely identified by a `UserIdentifier`, which also serves as primary key attribute for the database in the peristent implementation. The UML model is depicted in Figure 3.3.

Although `Shop` (Section 3.1) holds a reference to a `UserManager`, `PersistentUserManager` does not need to be a singleton instance and is indeed not. Moreover, a peculiarity of `PersistentUserManager` is, that a new instance can be created whenever one is needed. The reason for this behaviour is, that data is not stored inside the `PersistentUserManager` object itself, as it may be done with a collection-based implementation, but `PersistenUserManager` is merely a transparent interface to the JPA.

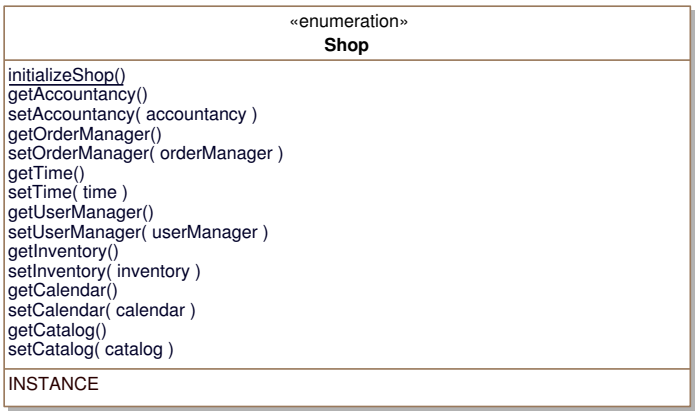

Figure 3.2: Shop - Class Overview

UserCapabilities

Capabilities in conjunction with a `HasCapabilityTag` (Section 2.3) can be used to change the appearance of a View, depending on a users status. For example, a View for a user having an "administrator" capability may display different content, for example delete buttons, than for a user not having that capability. Thus, capabilities allow for flexibility and assist in code reuse, when designing the View.

Login

To reduce code repetition, Salespoint 5 contains code to automate user log in. Using a JSP template, a special login form is generated, which is handled by an interceptor. [1] The interceptor verifies the user password and associates the current session with the user. The session is required, because multiple users can be logged in at the same time.

To modify the content of the View, depending on wether a user is logged in or not, the `LoggedInTag` can be used.

[1] An interceptor is a Spring concept.

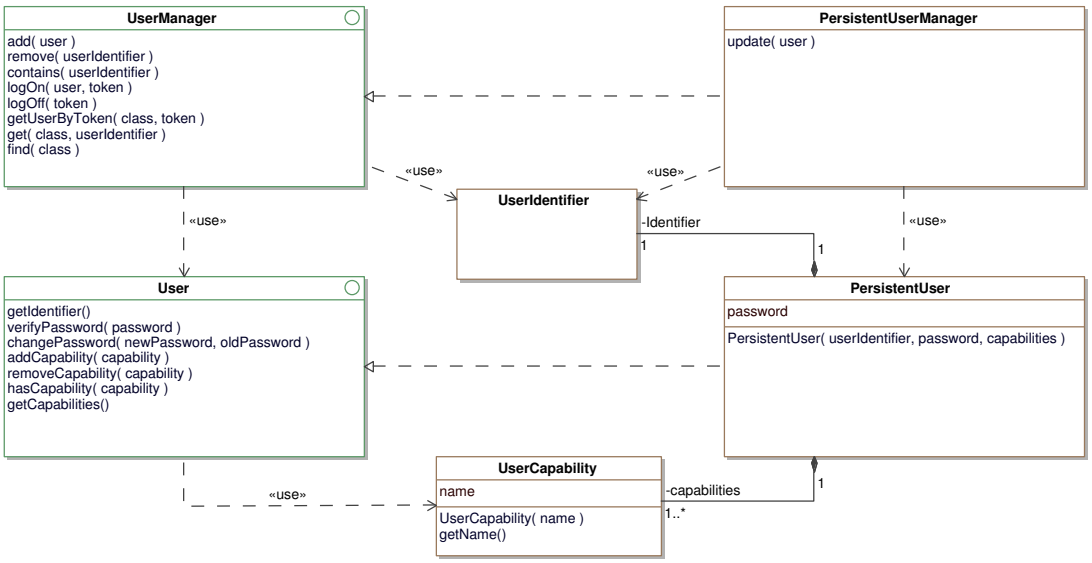

Figure 3.3: User - Class Overview

3.3 Calendar

The calendar is a new feature in Salespoint 5 to manage appointments. Figure 3.4 shows the UML model of the calendar.

`Calendar` is an interface that provides simple functionality to store calendar entries and retrieve `CalendarEntries`. `CalendarEntry` is an interface to set, store and access information about a single appointment. Both interfaces are implemented by `PersistentCalendar` and `PersitentCalendarEntry`, respectively. `PersistentCalendarEntry` is a persistence entity containing all information and `PersistentCalendar` manages the JPA access.

Every calendar entry is uniquely identified by a `CalendarEntryIdentifier`. This identifier also serves as primary key attribute when persisting an entry to the database. Additionally, a `PersistentCalendarEntry` must have an owner, a title, a start and an end date. The user who created the entry is known as the owner.

The access of other users than the owner to a calendar entry is restricted by capabilities. For each calendar entry, a user identifier and a set of capabilities are stored. Possible capabilities are:

- `READ` - indicates if a user can read this entry
- `WRITE` - indicates if a user can change this entry
- `DELETE` - indicates if a user can delete the entry

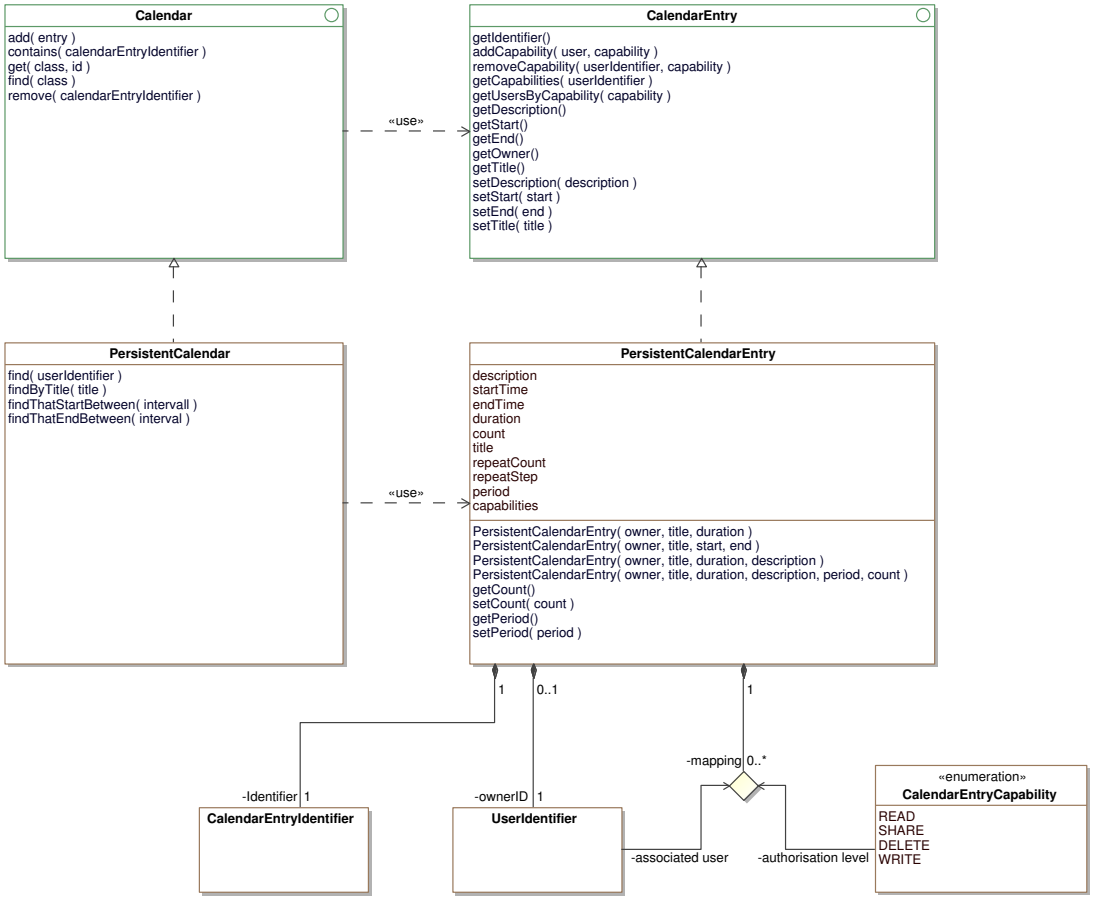

Figure 3.4: Calendar - Class Overview

- `SHARE` - indicates if a user can share the entry to other users

.

Because Salespoint 5 only implements the Model, the developer has to check and enforce capabilities in the controller.

Besides the minimum information a calendar entry can also have a description, which may contain more information. Periodic appointments are also supported, by specifing the number of repetitions and a time span between two appointments.

There are some conditions for temporal attributes of an calendar entry:

- the start must not be after the end
- the time between two repetitions of an appointment need to be longer than the appointment, so appointments do not overlap

3.4 Quantity

`Quantity` is used to represent amounts of anything. Three attributes allow `Quantity` to specify everything: a numerical value (`BigDecimal`), a (measurement) unit or metric (`Metric`), and a type specifying the rounding of the numerical type (`RoudingStrategy`), as can be seen in Figure 3.5.

`Quantity` objects are immutable and the class implements the `Comparable` interface.

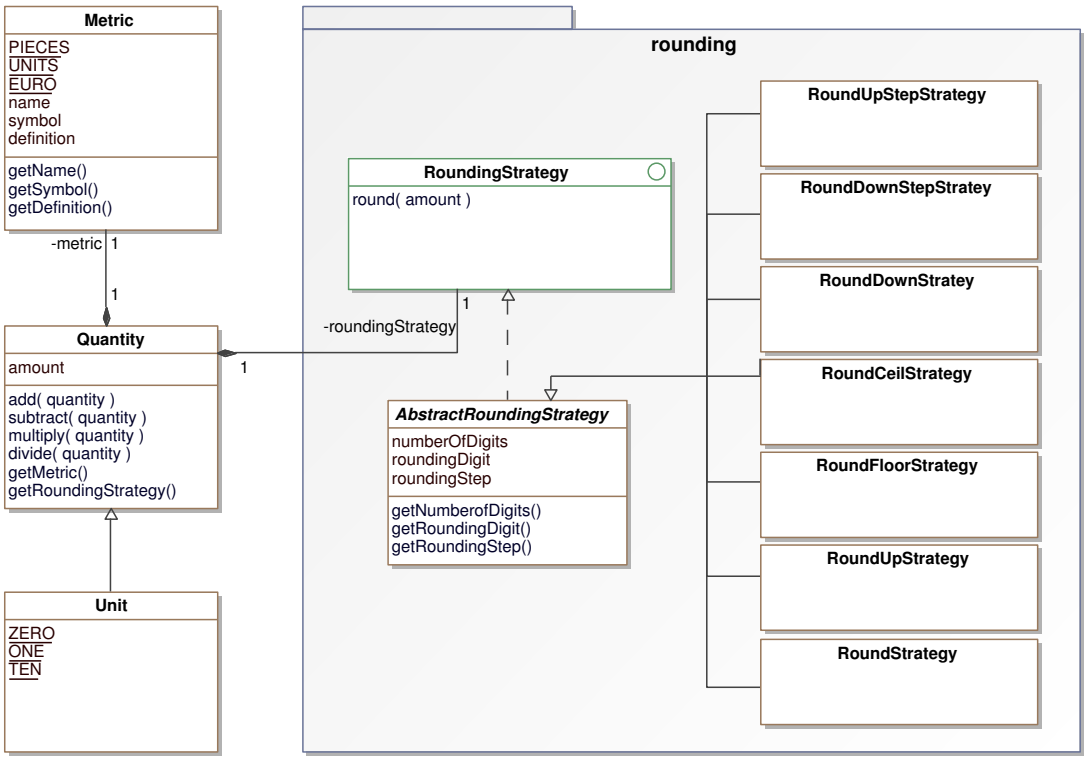

Figure 3.5: Quantity - Class Overview

3.4.1 `BigDecimal` - Representing numerical values

`BigDecimal` was chosen over `float` or `double` because of its arbitraty precision. Moreover, objects of `BigDecimal` are immutable and the `BigDecimal` class provides operations for including, but not limited to: arithmetic, rounding, and comparison.

3.4.2 `Metric` - What is represented

The composite type `Metric` contains all information pertaining to the unit or metric of the represented object. Examples for units or metrics are: m (meter), s (second), pcs (pieces). Thus, a metric can be described by a symbol (m) and a name (meter).

Furthermore, an object of type `Metric` has a description field, to explain the meaning of the metric in detail.

Convenience instances exist for euros, pieces and units.

3.4.3 `RoundingStrategy` - How to handle half a person

When handling quantities of unkown metric, standard rounding rules cannot always be employed. The case of natural persons is just one example, when rounding rules have to be restricted to yield a useful result. You can round in four general directions: away from zero, towards zero, towards positive infinity, and towards negative infinity.

Additionally, you can specify the digits after the decimal delimiter. Monetary values in € or $US are often just represented with two digits after the decimal delimiter. Other values, such as kilo grams may be required to be specified to four digits after the decimal delimiter or even further. In case of (natural) persons, the digits after the decimal delimiter is usually zero, except you are working in statistics (1.45 children per couple) or you are a serial killer dismembering your victims.

The third parameter for rounding is the rounding digit, i.e. the number specifying when you round up or down. Usually, this number is five. In case of persons, it is one: if you have $n.0\,persons$, you round down, otherwise up. If you are calculation a capacity for persons, you will have to round down, this can be done by specifying the correct rouding direction.

Sometimes, it is necessary to round a number to a nearest "step", i.e. if you sell something in packs of 50, and someone punches in 40, you will have to round up to 50. So your rounding step is 50. Another example is material, which is sold by the meter or yard. You have to round the amount specified by your customer accordingly. Of course, a rouding step can be smaller than 1, i.e. 0.25.

Two convenience rounding strategies exist so far: `RoudingStrategy.MONETARY` rouding with four digits after the decimal delimiter and rouding towards zero, and `RoudingStrategy.ROUND_ONE` with zero digits after the decimal delimiter and also rouding towards zero.

Rounding is implemented as strategy pattern, but an abstract class (`AbstractRoundingStrategy`) is introduced in the pattern (Figure 3.5). `AbstractRoundingStrategy` contains common methods such as `equals()` and `hashcode` and getters, thus reducing code repetition.

3.4.4 `Money` - A usecase for `Quantity`

Objects of class `Money` are used to represent amounts of currency within Salespoint. The following paragraphs detail the intended use, internal modelling and implementation of `Money`. The UML model is given in Figure 3.6.

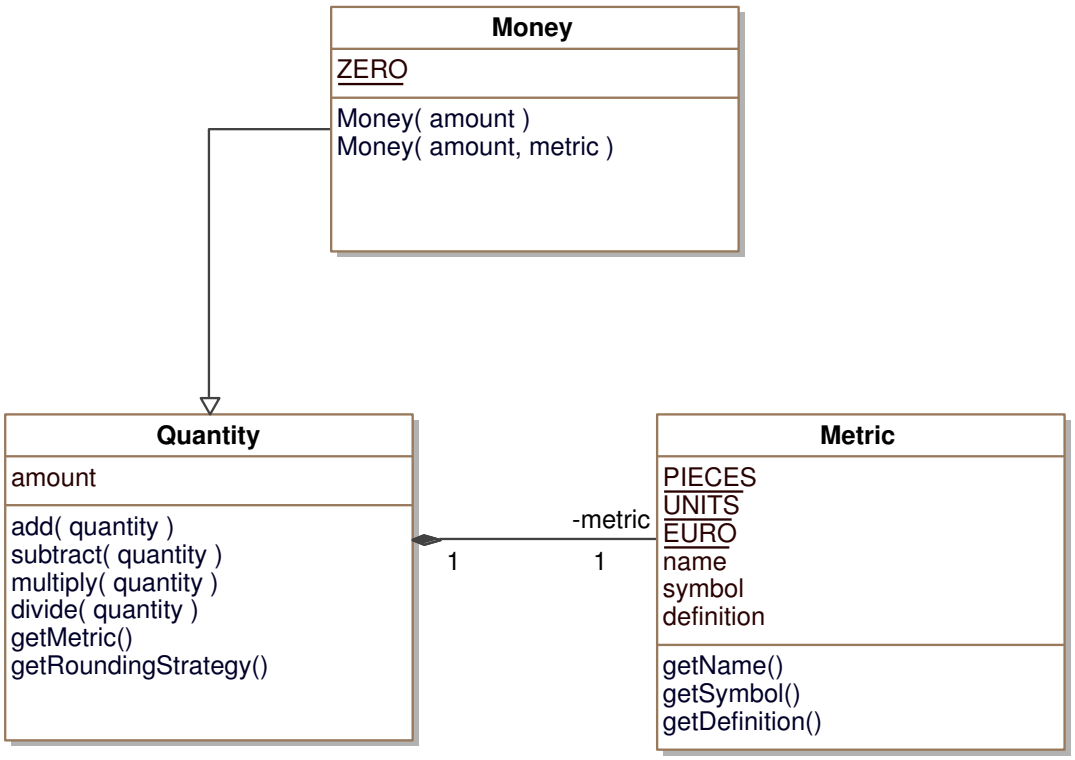

Figure 3.6: Money - Class Overview

A `Money` object can be instantiated by just passing the numerical value as constructor parameter. In this case, the metric `Metric.EURO` is used, as well as `RoundingStrategy.MONETARY` for the rounding strategy attribute.

For other currencies, a `Metric` parameter can be passed to the constructor along with a numerical paramter. However, conversion between currencies is not supported, as it was not deemed necessary.

The rounding strategy cannot be overridden. Internally, `Money` objects calculate with and are rounded to four digits after the decimal delimiter to minimize the rounding error. The `toString()` method, however, limits the output to the expected two digits after the decimal delimiter and appends the symbol of the associated `Metric`.

Two convenience instances exist: `Money.ZERO`, representing €0,00, and `Money.OVER9000`, representing an amount greater than €9000,00.

3.4.5 `Unit` - Representing persons or other integral items

To represent integral items conveniently, the objects of class `Unit` can be used. The rouding strategy is fixed for all instances to `RoundingStrategy.ROUND_ONE` and `Metric.PIECES` is used as metric. Convenience instances for amounts of zero, one and ten unit(s) exist (`Unit.ZERO`, `Unit.ONE`, and `Unit.TEN`).

3.5 Product

Products are represented by ProductTypes.

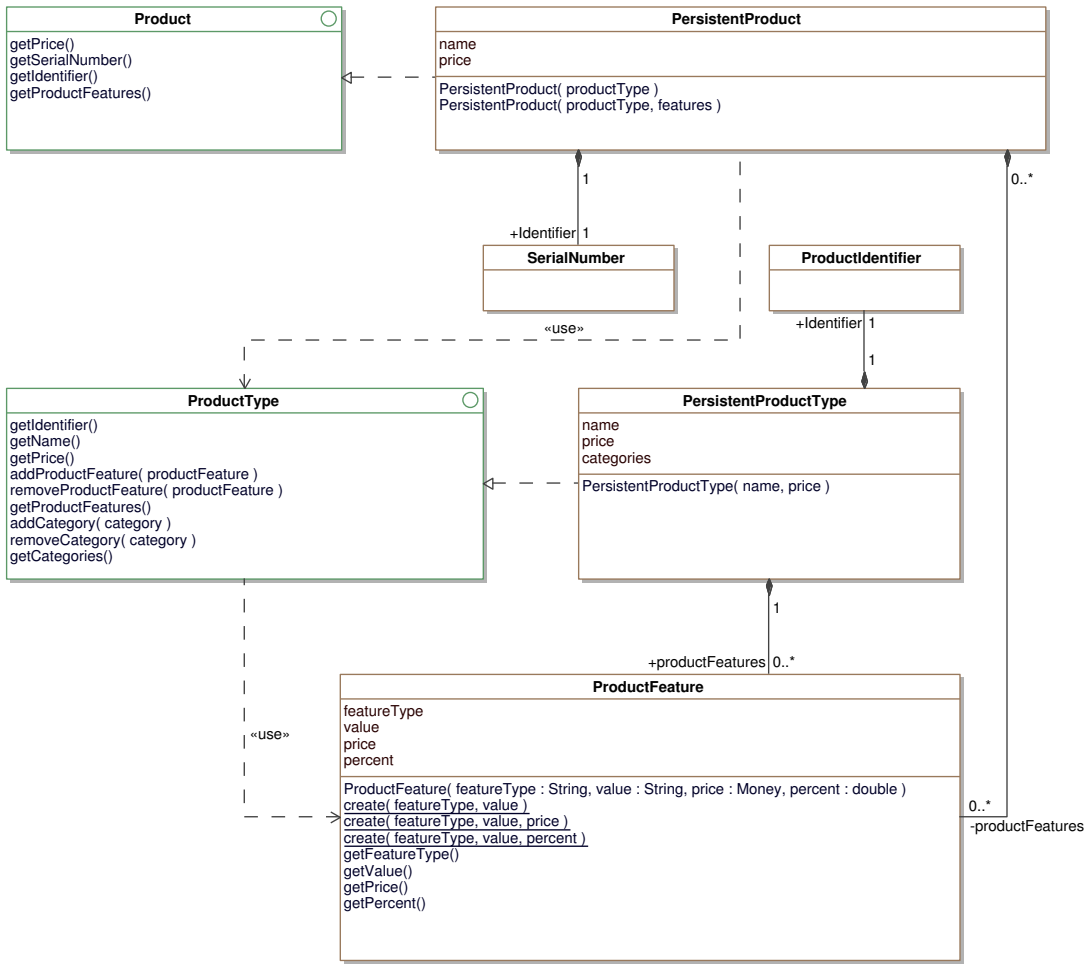

Figure 3.7: Product - Class Overview

3.5.1 `ProductType` - Representing specified products

A `ProductType` is an interface which represents a specified product. The `PersistentProductType`-class is an implementation of this interface to worked with it. An `PersistentProductType` contains `ProductFeatures` to specify your product. You can add any `ProductFeatures` to this set or remove them from it. An example: Our specified `ProductType` is a table. About to sell any variations of this table, you need any `ProductFeatures` like its color or the material of it. Also you can add categories

to your ProductType, in this case you would choose the category: furniture or living room.

3.5.2 ProductFeature - Creating Features

Every ProductFeature has a featureType, a value and a price, which define it. Any ProductFeatures can offer the same featureType. For example the features red or green offer the featureType color. Mahogany, oak or beech offer the featureType wood. The value defines the number of this ProductFeature in its ProductType. If you want to create a table, which offered four table-legs, then your table has a ProductFeature table-leg with a value 4.

3.5.3 Product - Representing ProductTypes

A Product is an interface which represents one specified ProductType. The PersistentProduct-class is an implementation of this interface to worked with it. If you create a PersistentProduct, it gets a SerialNumber ,the ProductIdentifier of its PersistentProductType and the price will be calculated. SerialNumber and ProductIdentifier are unique SalespointIdentifier to identify ProductInstances and ProductTypes in the database. There are subclasses of ProductType and Product, with them the functionality will be extended. The subclasses of ProductType are ServiceType and MeasuredProductType, the subclasses of Product are Service and MeasuredProduct.

3.5.4 MeasuredProductType - Creating MeasuredProducts

The interface MeasuredProductType is implemented by PersistentMeasuredProductType. With this class you can realize products, which are not sold as predefined unit, but rather as measures of something. For example flooring may be sold by square foot or fresh products by kilogram. A PersistentMeasuredProductType has a name, a price and a quantity on hand. The quantity on hand, is the quantity of this PersistentMeasuredProductType, which is available to be sold. You can also add or reduce a specified quantity of the PersistentMeasuredProductType to it. The price of the PersistentMeasuredProductType is not the price for an unit of that, but the price of the quantity on hand of that. The unit price will be calculated automatically and you can get it with the method getUnitPrice().

3.5.5 MeasuredProduct - Representing MeasuredProductTypes

The interface MeasuredProductType is implemented by PersistentMeasuredProductType, which represents a specified quantity of the PersistentMeasuredProductType. If it is created, the quantity of that, will be reduce from the quantity on hand of the PersistentMeasuredProductType, because this quantity is used and no more available for other PersistentMeasuredProducts. If this quantity is greater than the quantity on hand, an exception will be thrown.

3.6 Catalog

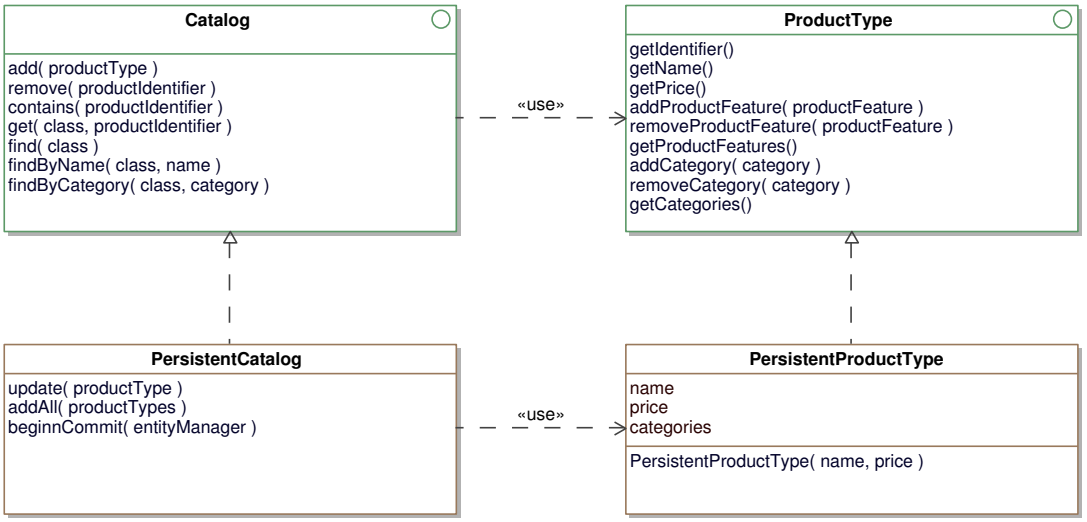

Figure 3.8: Catalog - Class Overview

3.6.1 `Catalog` - Organizing and presenting `ProductTypes`

The `Catalog` interface was designed to manage `ProductTypes` and `ProductFeatures`. It provides functionality for adding and removing `ProductTypes` as well as finding them based on their name and category. `ProductTypes` and `ProductFeatures` are more closely described in chapter 3.5.

The `PersistentCatalog` is an implementation of this interface and also provides an `update()` method to update and merge existing `PersistentProductTypes` to `Catalog` and database.

The `find()` methods request the database in the form of `CriteriaQuerys` which will be processed by JPA and results are returned in the form of `Iterables`. The reason for this is to make returned objects immutable without making it difficult to iterate over these results.

3.7 Inventory

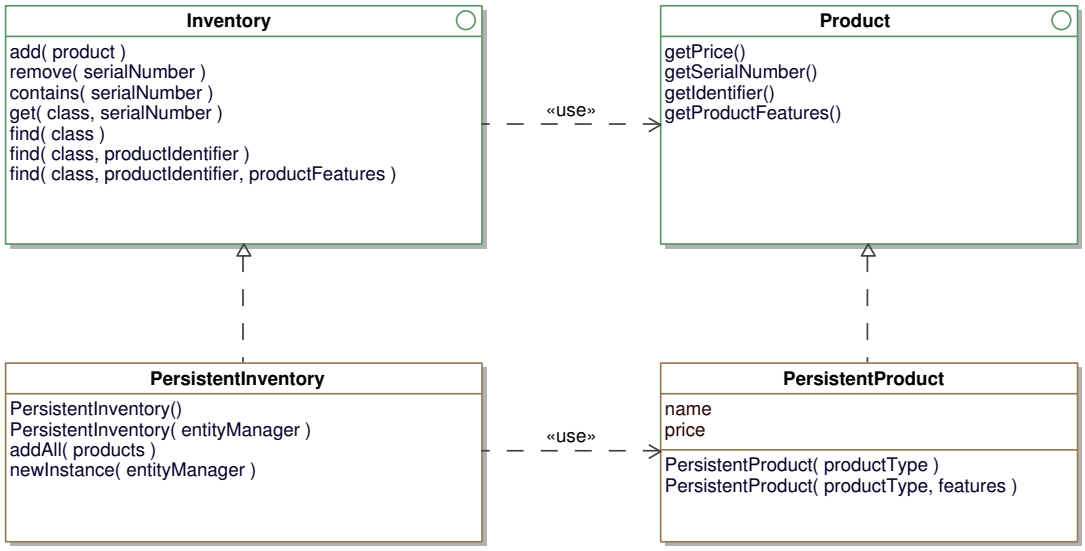

Figure 3.9: Inventory - Class Overview

`Inventory` is an interface and provides methods for adding and removing products as well as finding products based on the `ProductType` via the `ProductTypeIdentifier` and a set of `ProductFeature`s. `PersistentInventory` is an implementation of the `Inventory` interface. Every operation is delegated to the database, via `CriteriaQuery`s. Some methods require additional processing of query results with Java. `Inventory` aggregates `Product`s, `PersistentInventory` aggregates `PersistentProduct`s

3.8 Accountancy

The accountancy package contains functionality pertaining to book keeping. As in other packages, interfaces are used to separate between behaviour and implementation. `AccountancyEntry` is a representation of an accounting entry. `Accountancy` aggregates `AccountancyEntry`s. By implementing and subclassing `AccountancyEntry` the notion of different accounts, as known from double-entry bookkepping, can be realised. Every `AccountancyEntry` is uniquely identified by an `AccountancyEntryIdentifier`.

`PersistentAccountancyEntry` implements `AccountancyEntry` and serves as persistence entity, while `PersistentAccountancy` implements `Accountancy` and provides opaque access to the JPA layer. `AccountancyEntryIdentifier` is used as primary key attribute, when persisting entities to the database.

As can be seen in Figure 3.10, `PersistentAccountancyEntry` is sub classed to create a second "account" used to store payment information, namely `ProductPaymentEntry`. To access entries of only one type, thus belonging to one "account", `get()` and `find()` methods in `Accountancy` have a type parameter.

Payment information also includes a user identifier referencing the buyer, an order identifier referring to the `Order` which was payed, and a `PaymentMethod` describing the money transfer. The inheritance hierarchy of `PaymentMethod` is depicted in Figure 3.11.

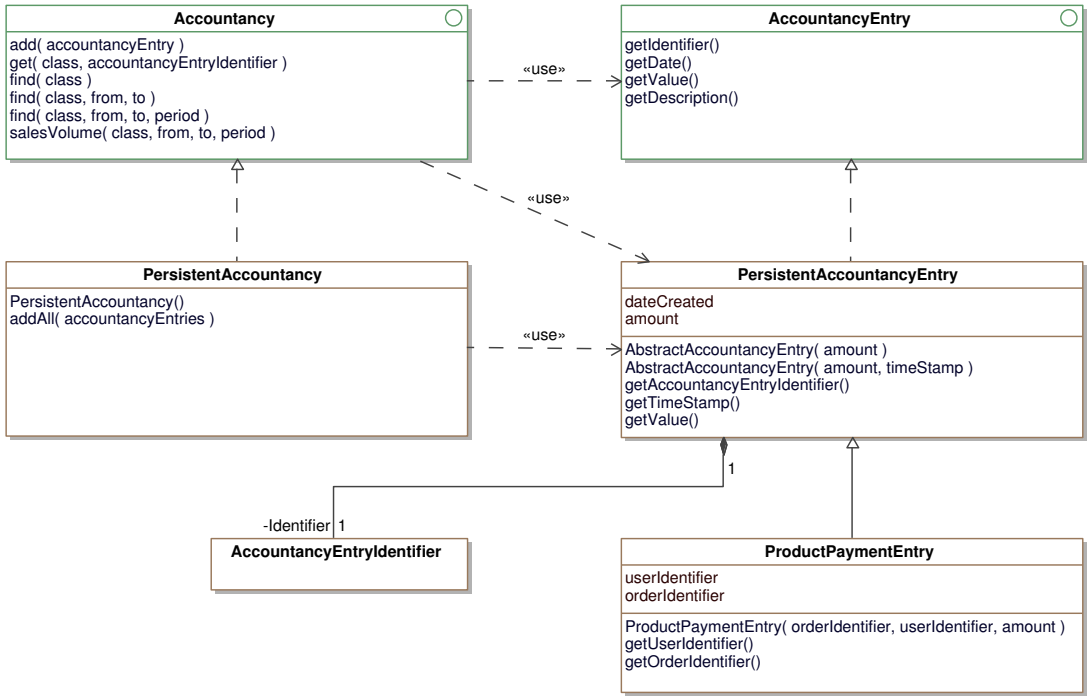

Figure 3.10: Accountancy - Class Overview

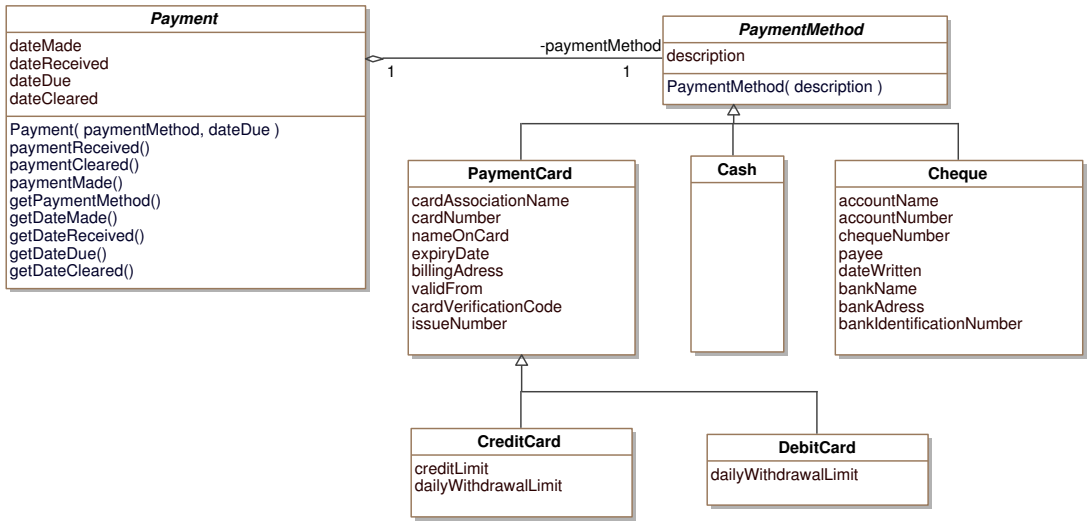

Figure 3.11: Payment - Class Overview

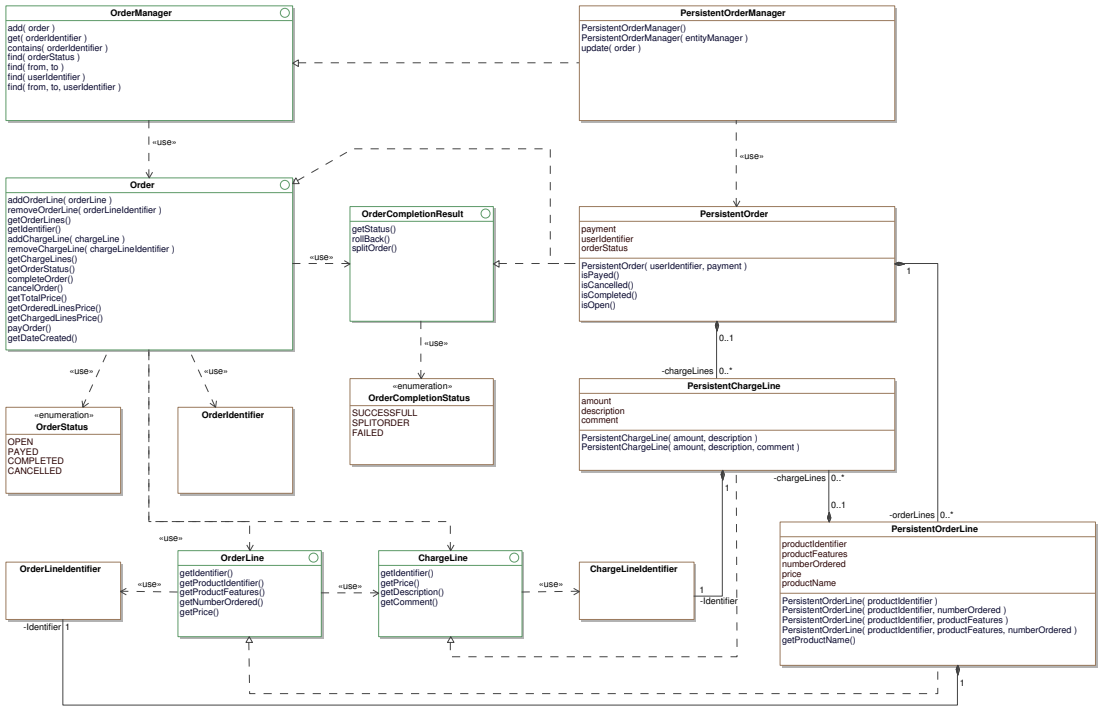

Figure 3.12: Order - Class Overview

3.9 Order

An `Order` can be imagined as sheet of paper which basically consists of lines representing the ordered products. An order can be uniquely identified by an `OrderIdentifier`.

Every product of an order is stored in separate `OrderLine`. An `OrderLine` is uniquely identified by an `OrderLineIdentifier`. `OrderLine`s contain all information to identify product instances, like a `ProductType` and `ProductFeatures`.

`ChargeLine`s represent additional costs or discounts and can be applied to `OrderLine`s or `Order`s. For example, they can be used to handle special taxes or handling fees. A `ChargeLine` is uniquely identified by a `ChargeLineIdentifier`.

`Orders` are lifecycle-objects. The lifecycle covers four states which are defined by enumeration type `OrderEntryStatus`. Changing lifecylce states is restricted, which is automatically governed by the according methods, for example `cancelOrder()`. `COMPLETED` is one of the final states and it is not possible to change the state of such orders.

As you can see in Figure 3.13, a PersistentOrder can only be modified in state `OPEN`. `PAYED`, `CANCELLED` and `COMPLETED` `Orders` are immutable. Calling the `payOrder()` method changes the state and calls the accountancy to create a `ProductPaymentEntry` Ordered objects will only be removed from inventory when the `completeOrder()`

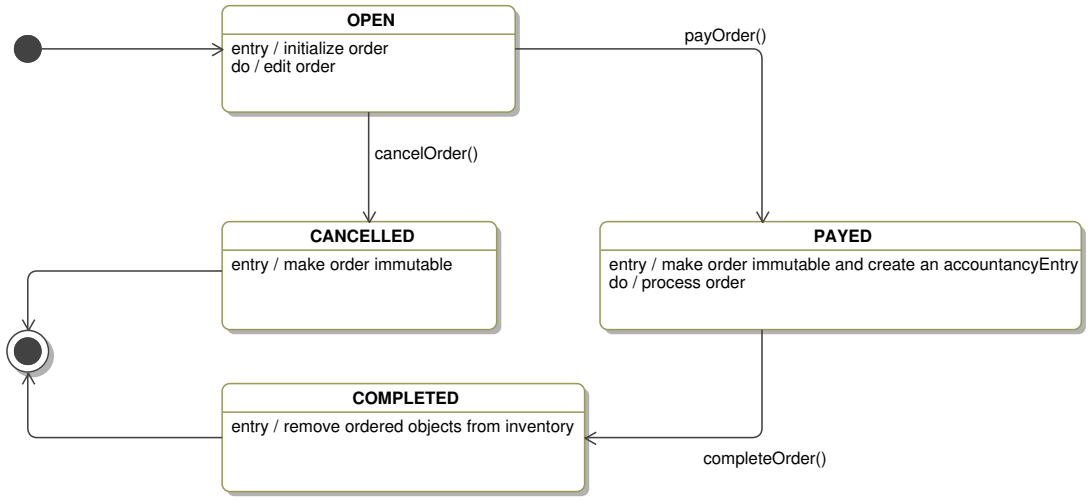

Figure 3.13: Order - Lifecycle

method is called.

Completing an order causes product instances to be removed from the inventory. Because product instances may not be present anymore in the inventory, or their number may not be suffice to fulfill an order, completing an order requires special attention. To handle these situations, the `OrderCompletionResult` interface was introduced. First of all, three `OrderCompletionStatus` are possible:

- `SUCCESSFULL`: The order was completed successfully, and all products were removed from the inventory.
- `SPLIT`: Some products could be found in the inventory and were removed.
- `FAILED`: An error from which recovery is impossible occured.

When completing an order results in the `SPLIT` status, the original order is splitted: all product that could be removed from the inventory are kept in the original order. The original order's state is changed to `COMPLETED`. All products which could not be removed from the inventory are transferred to a second order, the split order. The split order is set to `PAYED`. This scheme allows for the Controller to implement whatever logic necessary: placing a product on back order, splitting the order into multiple deliveries, or cancelling the order. It is paramount to understand, that `OrderCompletionResult` does not make a decision, but prepares for every decision, the business logic may come to.

The `OrderManager` aggregates `Orders`. The implementations `PersistentOrderManager`, `PersistentOrder`, and `PersistentOrderLine` are used to persist, update, find and remove orders to/from the database. In `Order` aggregated objects, like `OrderLines` and `ChargeLines` will also be persisted, updated or removed with the `Order` object.

4 Collaboration

Salespoint 5 is more than just a collection of classes and interfaces. It also provides processes between packages that are triggered automatically to facilitate working. This chapter draws attention to those dependencies between packages and describes how they collaborate.

Figure 3.1 illustrates the main dependencies between Salespoint 5 packages. As can be seen nearly all packages are interdependent.

The central class that connects all features in Salespoint 5 is the `Shop`. The most packages access this class to communicate with other packages. Therefore the `Shop` contains all interfaces which are global connected. This class should also be the first point for software engineers to request the individual parts of Salespoint 5.

Another package that collaborates with nearly all packages is the `Order` package. `OrderLines` using interfaces from `Product` package to identify product instances and calculate their prices. The `Catalog` package is used to check whether the catalog contains added products. `Orders` are also associated with the `UserManager`, to receive information about involved users.

Completed `Orders` will communicate with the `Inventory` (via `Shop` class) to remove considered product instances.

Before completion, `Orders` have to be payed. An `Order` which changed its status to `PAYED` will automatically access the accountancy and create the corresponding `AccountancyEntry` which represents that payment.

`Catalogs` and `Inventorys` also work closely together with all classes in `Product` package. There are a lot of other packages and classes that provide structures which are used in Salespoint 5 like the `Money` and `Quantity` packages. After all there are much more smaller collaborations in Salespoint 5, but the above described are the most important ones.

Bibliography

[AN03] Jim Arlow and Ila Neustadt. *Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML*. Addison-Wesley, 2003.

[ecl] `http://www.eclipse.org/eclipselink/`.

[hib] `http://www.hibernate.org/`.

[jod] `http://joda-time.sourceforge.net/`.

[spr] `http://www.springsource.org/`.

[top] `http://www.oracle.com/technetwork/middleware/toplink/overview/index.html`.