Salespoint 5

Technical Reference

Version 1

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

3rd April 2012

Technische Universität Dresden
Department of Computer Science
Institute of Software- and Multimedia-Technology
Software Technology Group

Address
Department 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

1 Preface **5**

2 Technical Background **7**
2.1 JPA - Java Persistence API 7
2.2 Joda Time 8
2.3 Spring 9
2.4 Software Architecture of a Salespoint 5 Application 9
2.5 General Design Aspects 10

3 Salespoint 5 Components **15**
3.1 Shop 15
3.2 User 17
3.3 Calendar 18
3.4 Quantity and Money 20
3.5 Product 24
3.6 Catalog 26
3.7 Inventory 27
3.8 Accountancy 28
3.9 Order 30

4 Collaboration **33**

Bibliography **34**

List of Figures

2.1 Overview of a Salespoint 5 application. 9
2.2 Layers of a Salespoint 5 application. 10
2.3 Example for a type hierarchy. 13

3.1 Package Overview 16
3.2 Shop - Class Overview 17
3.3 User - Class Overview 18
3.4 Calendar - Class Overview 19
3.5 Quantity - Class Overview 20
3.6 Money - Class Overview 24
3.7 Product - Class Overview 25
3.8 Catalog - Class Overview 27
3.9 Inventory - Class Overview 28
3.10 Accountancy - Class Overview 29
3.11 Payment - Class Overview 29
3.12 Order - Class Overview 31
3.13 Order - Lifecycle 32

List of Tables

1 Preface

1.1 Typographic Conventions

Two typographic conventions are employed throughout this document to highlight specific phrases. The following paragraphs describe when and why these highlightings are used:

`Mono-spaced Blue`

The mono-spaced, blue font is used to denote variable names, class names, type names, java keywords, java package names, and so forth.

Proportional Italic

Proper names and termini are printed in proportional, italic font.

1.2 Introduction

The Salespoint Framework is intended to minimize developing effort of 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 are 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 5 developers also adhere to that principle. Thus, Salespoint 5 itself uses a number of frameworks and APIs, which are introduced briefly. The software architecture of Salespoint 5 applications is also detailed.

2.1 JPA - Java Persistence API

One of the key features of Salespoint 5 is 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 Standard Edition or Enterprise Edition applications. Salespoint 5 uses JPA 2.0, developed under JSR 317 and finalised in Dec, 2009 [jpa].

Additionally 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 therefore 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.

Persistence entities may be related to each other by an inheritance hierarchy. A persistence entity may have a non-persistent superclass. Fields declared by a non-persistent superclass are not stored in the database if an inheriting entity is persisted. Three schemes exist to persist entites with an inheritance relationship: *single table*, *join table*, and *table per class*.

The *single table* strategy stores all instances of classes of an inheritance hierarchy in the same table. The table contains columns for every attribute a persistence entity in the hierarchy declares. The different types are distinguished by a type discriminator column. The discriminator value for each persistence entity in an inheritance hierarchy is generated automatically or can be supplied by the user.

The *join table* strategy uses a table for the root persistence entity of the inheritance hierarchy. Additionally, a table is addded for each persistence entity in the inheritance hierarchy. In the supplementary tables, a foreign key is used to reference a row in the table of the parent persistence entity. Each table contains only columns for fields declared by a specific persistence entity in the inheritance hierarchy, but neither for the entities children nor parents. To reconstruct an object from the database, the different tables have to be joined using this foreign key, thus the name of this strategy.

The *table per class* strategy creates a table for each persistence entity containing all fields of the class, including inherited fields.

The inheritance strategy of an inheritance hierarchy has to be declared at the root persistence entity. The inheritance stratety may not be changes for a sub-hierarchy, because JPA 2.0 does not require this feature. JPA 2.0 only requires the *single table* and *join table* strategies to be implemented. Salespoint 5 uses the *single table* strategy exclusively.

The query language JPQL, which is similar to SQL, 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. 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* [jpa, ecla] to facilitate type safe querying.

Multiple implementations of JPA 2.0 exist, for example TopLink [top] and EclipseLink [eclb]. The open source persistence and ORM framework Hibernate [hib] also supports JPA 2.0. Salespoint 5 uses 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 2.0 implementation.[1]

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 5 incorporates 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*. Instant is explained in the Joda Time documentation as follows:

> 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.

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

A *Duration* in Joda Time represents a duration in time, exact to the milisecond. Durations can be thought of as length of an Interval. A Duration does not have a start and an end, but is rather the difference $end - start$.

A *Period* also represents a duration in time, but in a more abstract way. A Period may be a month, which may have 28, 29, 30 or 31 days. The absolute length in miliseconds of those periods differ.

[1] This document will be updated, as soon as another JPA provider is tested with Salespoint 5.

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 implemented 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. Furthermore, JSP tags to check if a user is logged in and if a user has a certain capability are included in the framework.

2.4 Software Architecture of a Salespoint 5 Application

Software often need to be adaptive, flexible, and extendable. Using a suitable architecture pattern, such as the Model-View-Controller pattern, helps to meet these non-functional requirements. Figure 2.1 gives an overview of how a Salespoint 5 application is modelled. Salespoint 5, as domain framework, takes the place of the model in the MVC pattern. The model can be extended by sub-classing Salespoint 5 classes or by introducing entirely new classes. Salespoint 5 model classes and sub-classes thereof are transparently stored in a database. If new classes are added to the model and their state is also required to be persistent, the developer also has to facilitate persisting those objects using the JPA-API.

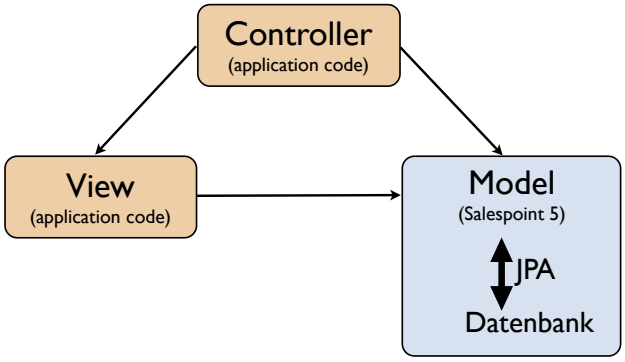

Figure 2.1: MVC-pattern of a Salespoint 5 application divided into application specific code (red) and framework code (blue).

The controller and view are application-specific and have to be implemented by the user. Although Salespoint 5 does not require a specific framework or API like

Swing [swi] or SWT [swt]. However, because Salespoint 5 is intended to be used in conjunction with the Spring MVC [spr] framework for SWP at TU Dresden, Salespoint 5 contains supplementary code, easing the development of Spring applications. This supplementary code consists of Spring MVC `PropertyEditor`s, and custom, Salespoint 5 specific JSP-tags.

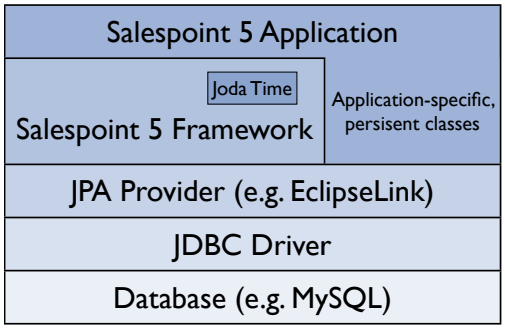

Figure 2.2: Layers of a Salespoint 5 application.

A layered view of the software architecture of a Salespoint 5 application is shown in Figure 2.2. The bottom layer corresponds to a DBMS, chosen by the developer. As stated in Section 2.1, JPA works with every DBMS for which a JDBC driver is available. The JPA provider, which is also chosen by the developer, interfaces with the JDBC driver and Salespoint 5. Salespoint 5 in turn uses a class library, namely Joda Time [jod], to deal with dates and times.

2.5 General Design Decisions and Aspects of Salespoint 5

This chapter summarises design decisions and aspects common to all Salespoint 5 subsection and details, why those decisions were made.

2.5.1 Notes on Interfaces ...[2]

Integrating a persistence layer into the Salespoint 5 Framework had a great impact on some design decisions made during the 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 JPA requirements influencing design decisions, Salespoint 5 strongly follows the *programming against interfaces* programming style. Although, creating an interface for almost every class violates the *KISS* (Keep it simpe, Stupid! Also, a hard-rock band.) 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. The classes itself are, however, not programmed against interfaces. Salespoint 5 usually just

[2] continued in the next sub-section

implement interfaces, but refer to other classes directly. The reason for this violation of the *programming against interfaces* paradigm is the generic typing, which would be required, if Salespoint 5 classes would refer to interfaces instead of concrete classes. For example, the `PersistentOrder` class would require three (cascaded) generic type parameters alone. As a consequence, Salespoint 5 classes refer to each other directly, instead of interface types.

2.5.2 ... implementing Classes, and Naming[3]

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). However, Java interfaces are used to separate behaviour and implementation. An interface defines only the behaviour of an object. Every persistence entity class is an implementation of a corresponding Java interface, to avoid JPA requirements to impact on design decision and, for example, influence the Salespoint 5 API. Each persistence entity has an aggregating class, which also implements an interface. The interface of an aggregating class specifies its API, but not implementation details. Aggregating classes and their respective interfaces are also called manager classes (interfaces), for example `UserManager` and `OrderManager`. Sometimes, class and interface names deviate from this naming scheme, for example `Calendar`, `Inventory` or `Accountancy`. The reason for this break in naming scheme is clarity: because a `Calendar` aggregates `CalendarEntrys`[4], its name according to the scheme would be `CalendarEntryManager`. Everybody knows what a `Calendar` is for, but not necessarily what a `CalendarEntryManager` does. Therefore, a more descriptive name was chosen. The specific manager implementations included Salespoint 5 facilitate storing objects to a database. However, an implementation based on Java collections rather than a database is entirely possible.

2.5.3 Why Salespoint 5 is so developer-friendly

Salespoint 5 is designed to be developer-friendly. A crucial part of its easy-to-use feeling is the consistency of interfaces, persistence entities and managers across the framework, including, but not limited to the naming of methods and behaviour of managers. All aggregating classes share a set of methods, namely `add`, `get`, `contains`, `remove` and `find`. The methods have the same semantics on every manager and have a similar method signature. Concise method names speed up development by reducing typing overhead. Instead of having an `addUser` method for the user manager or an `addOrder` method for the order manager, all managers have a method just named `add`. Consistency in the API is achieved by similar method signature. Consider the `get` method: it takes only one parameter which is an identifier. The precise type of the identifier is easily guessed: the order manager requires an `OrderIdentifier` and the `Accountancy` an

[3] started at the preceeding sub-section

[4] The correct plural form would be `CalendarEntries`, which is not a type name. The form "`CalendarEntrys`" is used to support full text search. Futhermore, "`CalendarEntrys`" should be read as "`CalendarEntry` objects" or "objects of the type `CalendarEntry`"

AccountancyEntryIdentifier. The name of the identifier type is derived from the name of class which is aggregated by the manager. This consistency allows a developer to use an unknown manager, when he is familiar with another manager.

Salespoint 5 does not contain checked exceptions, thus avoiding "the handcuffs they put on programmers" [HVE].

In Java 1.5 a new interface was introduced, the Iterable interface. The Iterable is implemented by classes, which aggregate objects and allow to iterate over those objects. The well known Collection interface is now a sub-interface of Iterable. An object implementing the Iterable interface is immutable in contrast to sub-interfaces of Collection, for example the List interface. Iterables are easy to handle, because compiler support to use the foreach construct is available. Also, there is no reason for a programmer to touch an iterator by himself because it is not idiomatic language use.

In Salespoint 5 Iterables are return on find() methods. Having an Iterable signals the developer, that he may not modify the object. Although an Iterable may be converted to a List or Set, changes to the resulting object, are not reflected in the original Iterable object. Thus, it has to be noted, that objects **cannot** be added by modifying a Salespoint 5 return value. The proper add() or addAll() methods have to be used.

2.5.4 Type-based queries in JPA

The get() and find() methods of aggregating classes in Salespoint 5 have a type parameter of Class<E>. The type E is a sub-class of type T, the generic type parameter of the interface. For example, the Catalog interface is generically typed to <T extends Product>, where Product is an interface itself. The complete type is thus Catalog<T extends Product>. PersistentCatalog implements the generic Catalog interface with Catalog<PersistentProduct>. Thus, PersistentCatalog aggregates instances of PersistentProduct or sub-classes thereof.

When Catalog is queried for Products using the get() and find() methods, a type parameter has to be supplied. This type parameter has two purposes: first, it ensures type-safety, and second it narrows the result set.

When a certain type, for example T extends PersistentProduct, is requested from an aggregating class, let's say the PersistentCatalog, the return type is T for the get() method and Iterable<T>[5] for the find() method. Thus, the return type depends on the request-type and is not a static type, for example PersistentProduct. The dependency of the return type on the requested type avoids type casts, which can fail at run-time and therefore increases type-safety. Because JPA is aware of persistence entities types, the result is always of the correct type, or empty, if no matching object could be found.

The result set determined by the requested type, because the type parameter does not request a specific type, but rather has instanceof behaviour. That means, a type parameter does not only match to objects of the same type, but also all objects which have the type of a sub-class of the requested type. It does not match super-classes of the

[5] More on Iterables in sub-section 2.5.3.

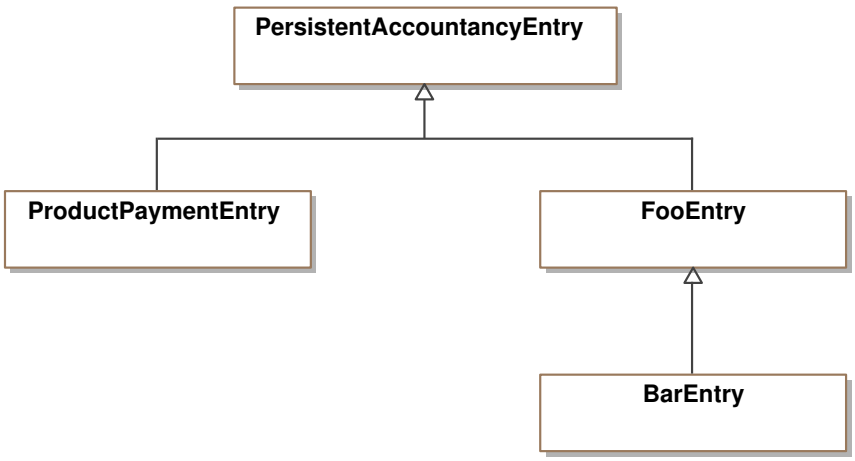

Figure 2.3: Exemplary class hierarchy of `PersistentAccountancyEntry`s.

requested type. The VideoShop-tutorial[6] is an example of how this functionality can be used. The `VideoCatalog` extends `PersistentCatalog`. By using the type parameters `Dvd.class` and `BluRay.class`, it requires only one line of code to find all products of a certain kind (type).

`PersistenceAccountancyEntry.class` matches all class types.

Consider Figure 2.3 for another example, explaining the `instanceof` semantics. `PersistentAccountancy` aggregates `PersistentAccountancyEntry`s and sub-classes thereof. One sub-class is already supplied by Salespoint 5: `ProductPaymentEntry`. `ProductPaymentEntry`s are automatically created, if an order is completed.

If `ProductPaymentEntry.class` is supplied as type parameter, only objects of type `ProductPaymentEntry` would be returned, if any. Using `FooEntry.class` as type parameter would return objects of type `FooEntry` and objects of type `BarEntry`, if any.

[6] `http://www.st.inf.tu-dresden.de/SalesPoint/v5.0/wiki/index.php/VideoShop`

3 Salespoint 5 Components

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, whose structure closely follows the package structure of Salespoint 5 `core` package.

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 and interfaces aggregating (persistent) objects 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[1] `Shop.initializeShop()`; it is used for setting all managers of `Shop` to Salespoints persistent class implementations and the time to `DefaultTime`. This behaviour is also known as *convention over configuration*, which means reasonable default values are supplied, eliminating the need to explicitly specify those values in most cases. `Shop` is implemented as a singleton.

Although `Shop` holds references to all managers and aggregating classes (see Section 3), persistent implementations do not need to be singletons and are infact not. Moreover, a peculiarity of all managers and aggregating classes 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 manager object itself, as it may be done with a collection-based implementation, but the manager class is merely a transparent interface to the JPA and database.

[1] It was mentioned to us, that boilerplate code may also be called infrastructure code. We disagree, because we consider configuration files, (startup-) scripts, deployment scripts, etc infrastructure code. Wikipedia defines boilerplate code as follows:

> In computer programming, boilerplate is the term used to describe sections of code that have to be included in many places with little or no alteration. It is more often used when referring to languages which are considered verbose, i.e. the programmer must write a lot of code to do minimal jobs.

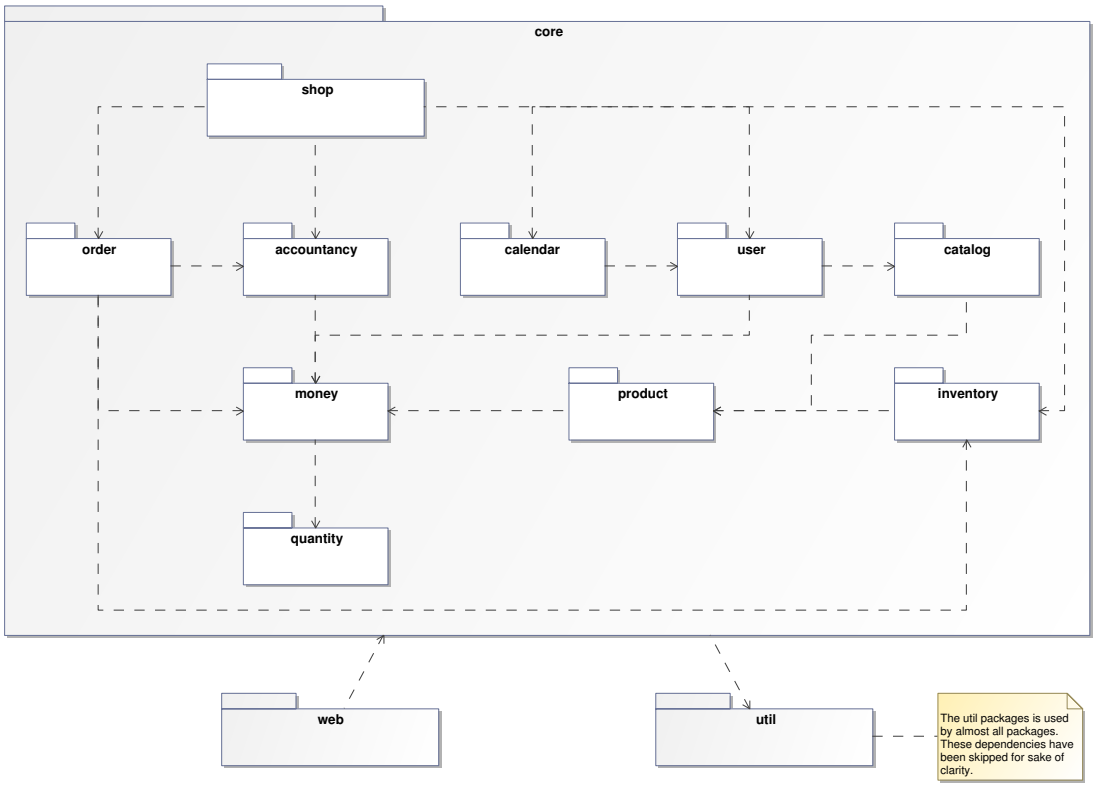

Figure 3.1: Package Overview

Figure 3.2: Shop - Class Overview

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`, who 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.

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.[2] The interceptor verifies the user password and associates the current session with the user using `login` and `logoff`. The session is required, because multiple users can be logged on at the same time.

[2] An interceptor is a Spring concept.

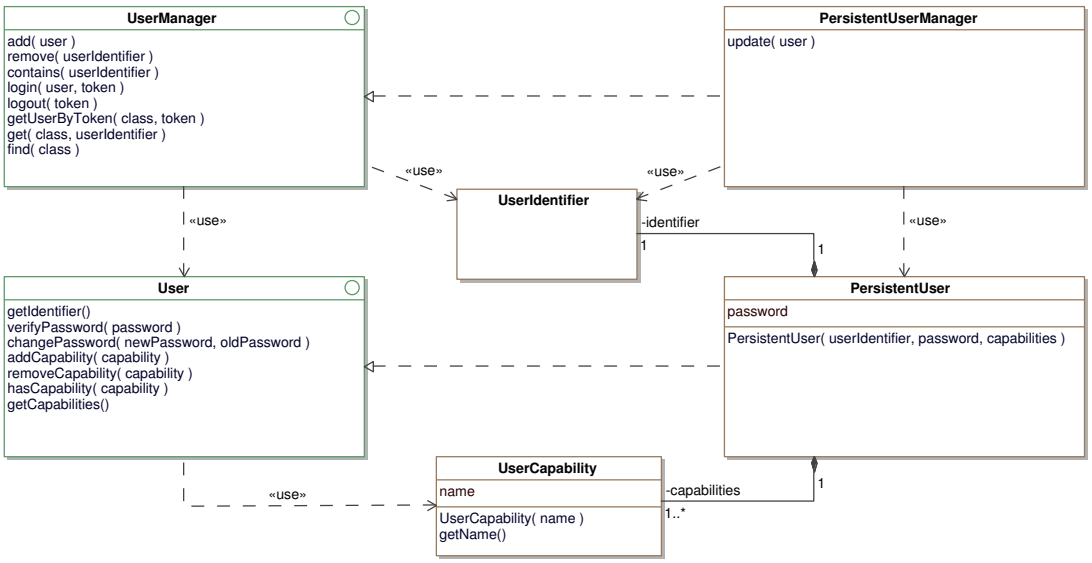

Figure 3.3: User - Class Overview

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

3.3 Calendar

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

`Calendar` is an interface that provides simple functionality to store and retrieve `CalendarEntry`s. `CalendarEntry` is an interface to set, store and access information of a single appointment/event. Both interfaces are implemented by `PersistentCalendar` and `PersistentCalendarEntry`, 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 and identified by `ownerID` of the type `UserIdentifier`.

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

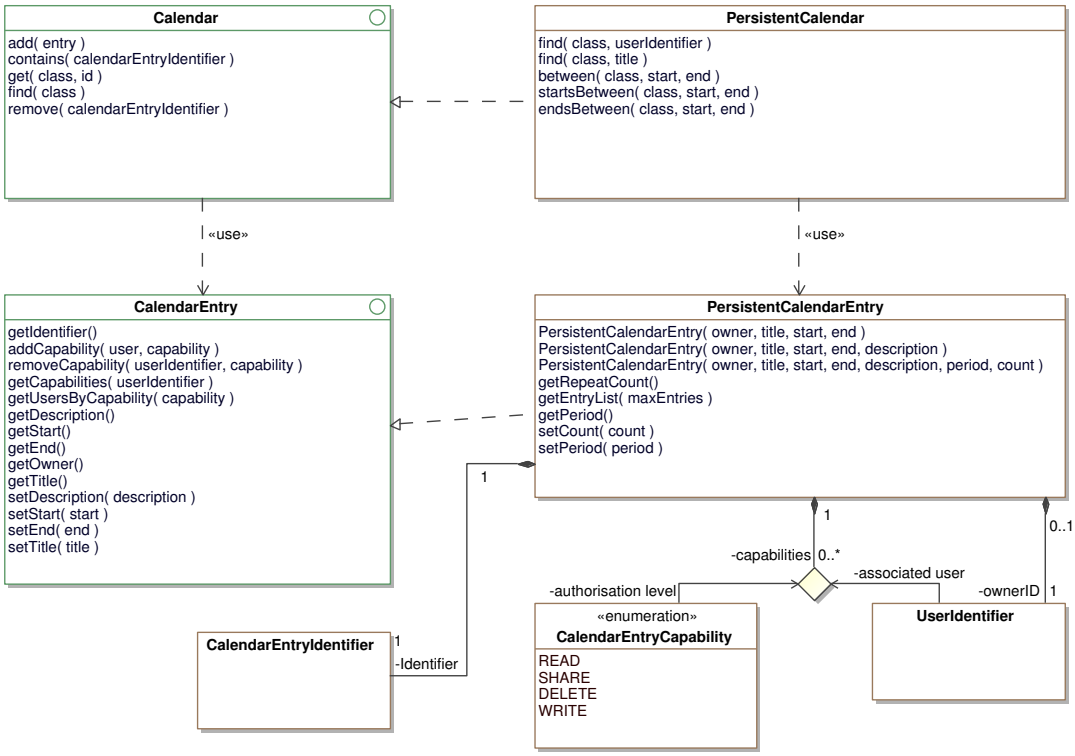

Figure 3.4: Calendar - Class Overview

- `WRITE` - indicates if a user can change this entry
- `DELETE` - indicates if a user can delete this entry
- `SHARE` - indicates if a user can share this entry with other users

.

Because Salespoint 5 only implements the model of the MVC pattern (see Section 2.3), the developer has to check and enforce capabilities in the controller.

Besides the minimum information of owner, title, start and end a calendar entry can also have a description, which may contain more information. Periodic appointments are also supported, by specifing the number of repetitions (`count` in Figure 3.4) and a time span between two appointments (`period` in Figure 3.4).

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 duration of the appointment, so that appointments do not overlap

3.4 Quantity and Money

`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 (`RoundingStrategy`), as can be seen in Figure 3.5.

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

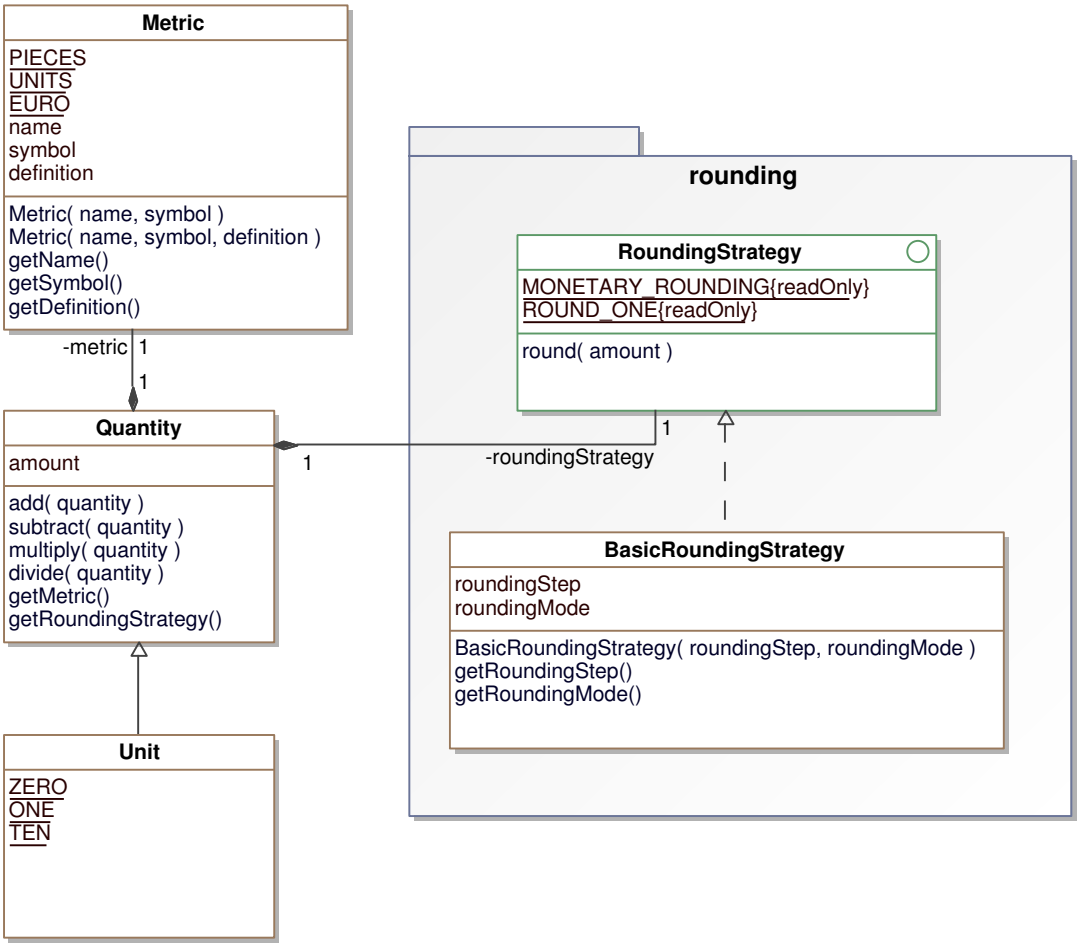

Figure 3.5: Quantity - Class Overview

3.4.1 `BigDecimal` - Representing numerical values

`BigDecimal` was chosen as datatype for the `amount` attribute (see Figure 3.5) 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). For example consider the unit of length "meter": represented by an object of the class Metric the symbol would be set to "m" and the name to "meter". Furthermore, an object of type Metric has a description field, to explain the meaning of the metric in detail. For the example of a meter a possible description could be "The meter is the length of the path travelled by light in vacuum during a time interval

Convenience instances exist for euros, pieces and units, namely EURO, PIECES, and UNIT (see Figure 3.5).

3.4.3 Rounding

Rounding is an arithmetic operation where a numberical value is replaced by another, simpler representation. Rounding is often results in a number which is easier to handle than the original value, for example 3.1415 instead of π, or 1.1414 instead of $\sqrt{2}$. Also, rounding may be employed to indicate the accuracy of a computed or measured value. For example a value is calculated to be 3.4563 but it is known the result is only accurate to one hundredth because of measurement errors, so the result may be rounded to 3.46. Rounding may also be used to limit the number of significant digits, for example in statistics. The german population is rounded to full thousands by the Federal Statistical Office and the statistical Offices of the Länder[3].

Different types of rounding exist. A number can be rounded to a specific precision, to an (integer) multiple of a step or increment, or to the nearest available value ot of a set of preferred values. Thus, rounding is implemented as strategy pattern, allowing a consistent interface to be used with different algorithms. So far, the only strategy implemented is the BasicRoundingStrategy which supports rounding to a specific precision or rounding to a multiple of a pre-determined multiple of a step.

Rounding to precision zero is also known as rounding to integer. When doing so, five general strategies can be employed:

- **up** - result is the next integer, away from zero
- **down** (also known as "truncation") - result is the next integer, towards zero
- **ceil** - result is the next integer, towards positive infinity
- **floor** - result is the next integer, towards negative infinity
- **half** - result is the nearest integer

All strategies except the last yield unambiguous results. If a number ends in $x.5$ and is rounded to the nearest integer two results exist: x and $x + 1$, since both have the

[3] http://www.statistik-portal.de/Statistik-Portal/en/en_zs01_bund.asp

same absolute difference to $x.5$. To resolve the ambiguity, tie breaking rules have to be employed. The first four strategies to round integers (up, down, ceil, floor) can also be used for tie-breaking. Additionally, more sophisticated methods may be employed, for example:

- **even** - result is the nearest even integer
- **odd** - result is the nearest odd integer
- **stochastic** - result randomly chosen to be x or $x + 1$
- **alternate** - using strategies up and down intermittently

A technique called dithering, which is related to stochastic rounding tie-breaking rule, is used when instead of the accuracy of single rounding operation is less important than the distribution of rounding results across a set of values. Dithering is often used in quantizing image or audio data. For example, if a continous stream of values of approximately 0.88 should be rounded to zero digits after the radix delimiter, that is to integer, would result in contiuous 1 or 0 result, depending on the strategy. Using dithering, a number is rounded up (or down) with the probability of the fraction. In our example, this would result in a stream containing 88% ones and 12% zeroes, randomly distributed. A similar technique, which reduces the average error, is called "error diffusion".

As mentioned earlier, only one strategy is implemented as of yet, namely `BasicRoundingStrategy`. Instantiating an object of this strategy, it is possible to specify either a rounding step or a precision in digits after the radix delimiter. Internally, the following algorithm is used to round numbers:

$$x = round(q/m) * m$$

where q is the number to be rounded, m is the increment or rounding step, x is the result, and $round$ is a rounding strategy to round the quotient q/m to an integer value.

When the number of digits after the radix delimiter are specified, the appropriate rounding step is calculated as follows:

$$step = \frac{1}{10^{digits}} = 10^{-digits}$$

For example, rounding a number to four digits after the radix delimiter is equal to rounding the number to the next (integer) multiple of 0.0001.

Rounding to a nearest step or increment, can be used if something is sold in fixed quantities. For example, if an item is sold 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 rounding step can be smaller than 1, i.e. 0.25.

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

rounding towards zero.

Rounding strategies which are not yet implemented, but may be interesting to Salespoint 5 users and developers are scaled rounding, rounding to nearest value of a predetermined set and a sum preserving rounding strategy. Implementing these strategies may require extending the `RoundingStrategy` interface.

Scaled rounding is used to round values on a logarithmic scale. Near zero a high precision is required, for example using three digits after the radix delimiter. But farther away precision requirements drop, so only one digit after the radix delimiter may suffice. Thus, the precision to which is rounded depends on the magnitude of the value to be rounded.

Rounding to nearest preferred value is used, when a calculated value is rounded to the nearest standard value. Examples of preferred values are the E series, the Renard series, powers of two, metric paper sizes and pen sizes, tuning systems in music, or film speed, aperture sizes and shutter speeds in photography. E series are most commonly used in electronics industry for resistor, capacitor, and inductor values.[4] In computer science, powers of two are often used as preferred values for sizes. Paper is sized so that neighboring dimensions have a ratio of $\sqrt{2}$.[5] This also applies for pen sizes, so that a pen size is available for a scaled drawing.

Sum preserving rounding is necessary if, for example for every item on an invoice the tax is explicitly listed. The tax for each item has to be rounded, introducing a rounding error. When the taxes for each item are summed up, the sum has to match the tax calculated for the summed prices of all items. This is seldomly the case, so the rounding of the individual tax has to be fitted. The method of least squares may be used to achieve fitting. Another algorithm which may be used is minimizing the round-off error.

3.4.4 `Money` - A usecase for `Quantity`

An object of the class `Money` is used to represent an amount of currency. The following paragraphs detail the intended use, internal modelling and implementation of `Money`. The UML model is given in Figure 3.6.

A `Money` object can be instantiated by 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 use four digits after the decimal delimiter for arithmetic operations 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.

[4] IEC 60063
[5] ISO 216

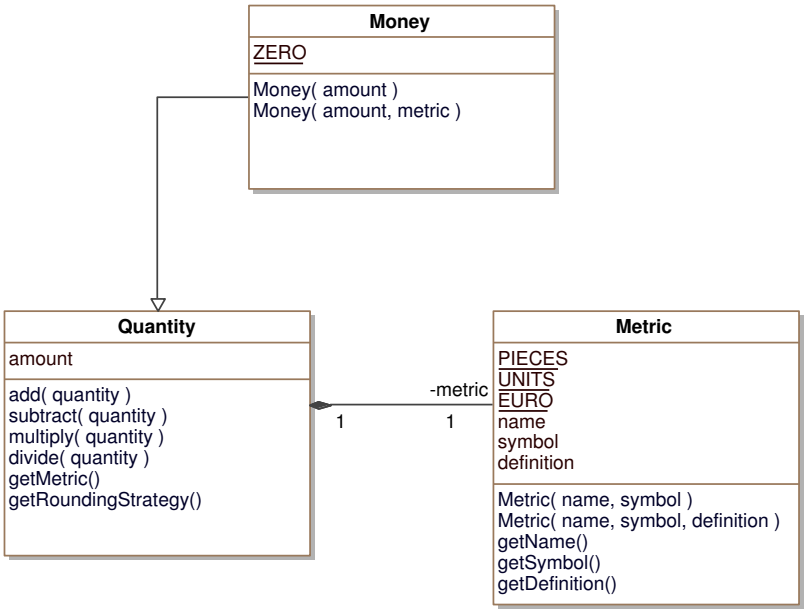

Figure 3.6: Money - Class Overview

3.4.5 `Unit` - Representing persons or other integral[6] items

To represent integral items conveniently, the objects of class `Unit` can be used. The rounding strategy is fixed for all instances to `RoundingStrategy.ROUND_ONE` (Figure 3.5) and `Metric.PIECES` (Figure 3.6) is used as metric. Convenience instances for amounts of zero, one and ten unit(s) exist (`Unit.ZERO`, `Unit.ONE`, and `Unit.TEN`; see Figure 3.5).

3.5 Product

Salespoint 5 is intended as framework for point-of-sale applications. The items for sale are called "products" and represented by instances of classes who implement the `Product` interface. A general overview of the Salespoint 5 products subsystem is given in Figure 3.7. To represent different kinds of products, `PersistentProduct` can be subclassed; see Section 2.5.4 for more information. `PersistentProduct`s are aggregated by `PersistentCatalog` (see Section 3.6).

`Product`s are supposed to be an abstraction, like an item on display or a picture in a catalog. `ProductInstance`s are used to represent the actual item you get, when you a buy a product. `Product`s are identified using a `ProductIndentifier`, whereas `ProductInstance`s are identified by a `SerialNumber`. `ProductInstance`s can be thought of as identifiable instances of a certain product, which are identical apart from their `SerialNumber`.

[6] whole-number

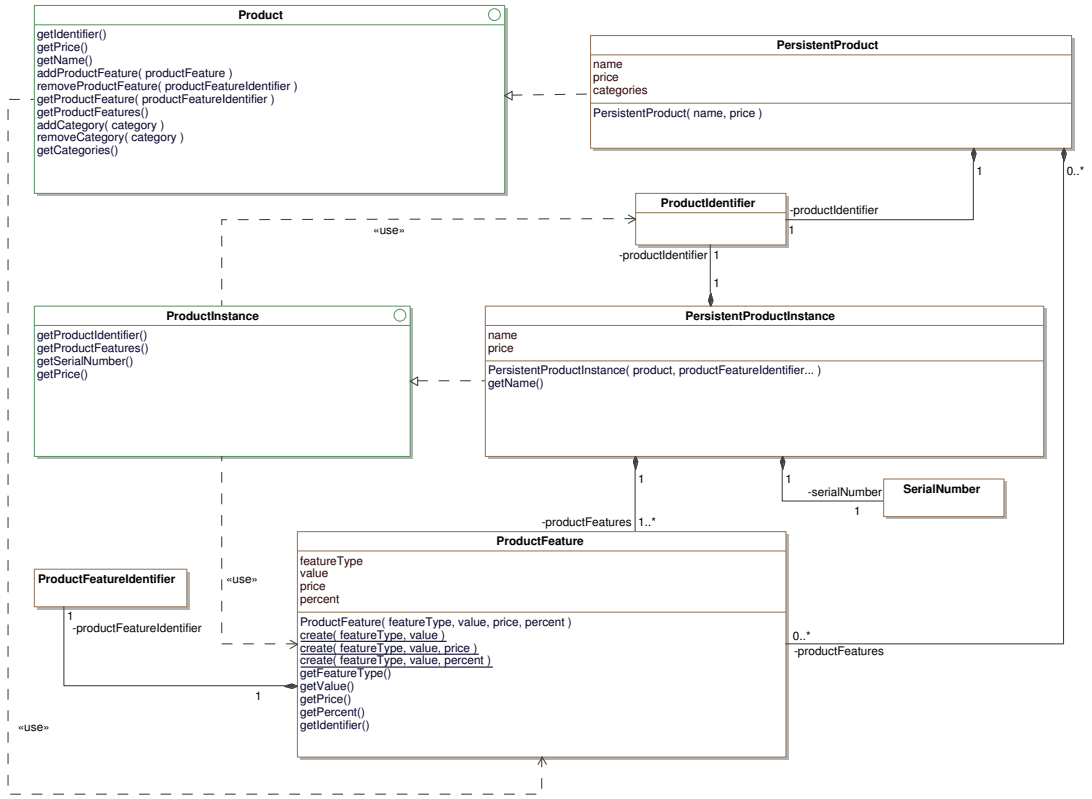

Figure 3.7: Product - Class Overview

To conviently handle products, which are essentially the same but differ in certain aspects, such as color or size Salespoint 5 has the concept of a ProductFeature. ProductFeatures are specified by a featureType, for example color or size, and a corresponding value, for example "black" or "blue" for the feature "color". Additionally a ProductFeature may reference a Money object, to describe an increase or decrease in price of a Product, if it has a certain ProductFeature. Alternatively, a change in price can be expressed as a percentage of the price of the Product.

An example: A class Shoe extends PersistentProduct and has a Set<ProductFeature> containg the values 36, 37, 38, 39, 40, 41, 42, 43, 44, 45 of the productType "size". The set of ProductFeatures declared in PersistentProduct defines, which ProductFeatures can be aggregated by the corresponding ProductInstance. An instance of Shoe represents a specific model a vendor might have. Additionally, a class ShoeInstance may sub-class PersistentProductInstance. An instance of ShoeInstance represents a specific pair of shoes. ProductInstance also aggregates ProductFeatures, but in contrast to Product exactly one ProductFeature is allowed for any featureType. In other words: a shoe has a size - exactly one size.

Not all items might by sold by number. Other units, such as litres, kilo grams, or meters are conceivable. To accomodate for the sell of such items, the MeasuredProduct interface was created. Implemented by PersistentMeasuredProduct, a MeasuredProduct is specified by a name, price and quantity available. When an amount from a MeasuredProduct is removed or added, the price attribute is automatically modified to represent the total monetary value of the MeasuredProduct. The getUnitPrice() method can be used to access the price of a single unit.

MeasuredProducts bought by customers are represented by classes implementing the MeasuredProductInstance interface. An instance of a MeasuredProductInstance stands for a certain amount of a product. When instanciating an object of a class implementing MeasuredProductInstance, the corresponding MeasuredProduct has to be known. Furthermore, the amount of the product represented by the new instance of MeasuredProductInstance is removed from the MeasuredProduct instance. If an instance is to be created, which would remove a higher quantity than is available in the MeasuredProduct, the instantiation fails with an exception.

3.6 Catalog

The Catalog interface was designed to manage Products and ProductFeatures in the system. It provides functionality to add, remove, and find Products. Products can be searched by their name or category. Products and ProductFeatures are more closely described in Section 3.5.

The PersistentCatalog is an implementation of the Catalog interface. Additionally PersistentCatalog provides an update()-method to update and merge existing PersistentProducts to the database.[7]

[7] update() to the interface. Misuse add() for updates? this seems inconsistent.

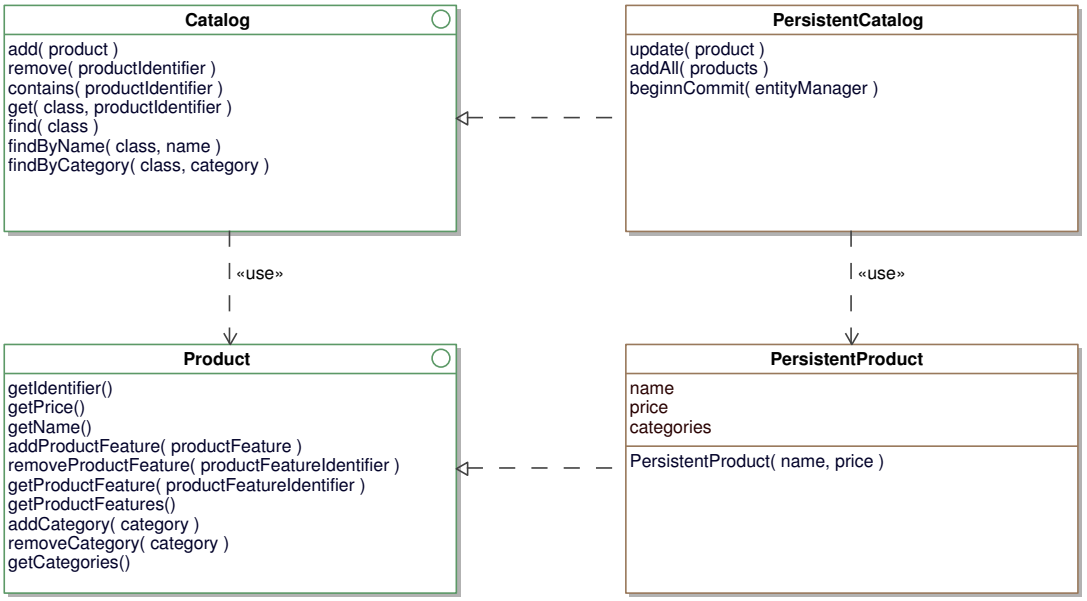

Figure 3.8: Catalog - Class Overview

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

An inventory is a place, where products are stored. In Salespoint 5, an abstract representation if the `Inventory` interface and its implementing class `PersistentInventory`. The interface and declares methods to add, remove and find products. Because an inventory contains specific product instances, `PersistentInventory` aggregates `PersistentProductInstances`.

`PersistentProductInstances` can be retrieved from `PersistentInventory` by specifying a `SerialNumber` or a `ProductIdentifier`. A `SerialNumber` is used to reference a specific `ProductInstance`. A `ProductIdentifier` identifies a `Product` uniquely, thus all `PersistentProductInstances` of the `PersistentProduct` specified by the supplied `ProductIdentifier` are returned. Additionally an `Iterable<ProductFeature>` can be supplied to the `find()`-method along with a `ProductIdentifier` to retrieve all instances of a product, where the `ProductFeatures` match exactly those specified. Matching a set of `ProductFeatures` against a `PersistentProductInstance` is hard to express in JPQL or Criteria Queries (see Section 2.1). Therefore, only the `ProductIdentifier` is used to build a Criteria Query, which is executed on the database. Selecting only those

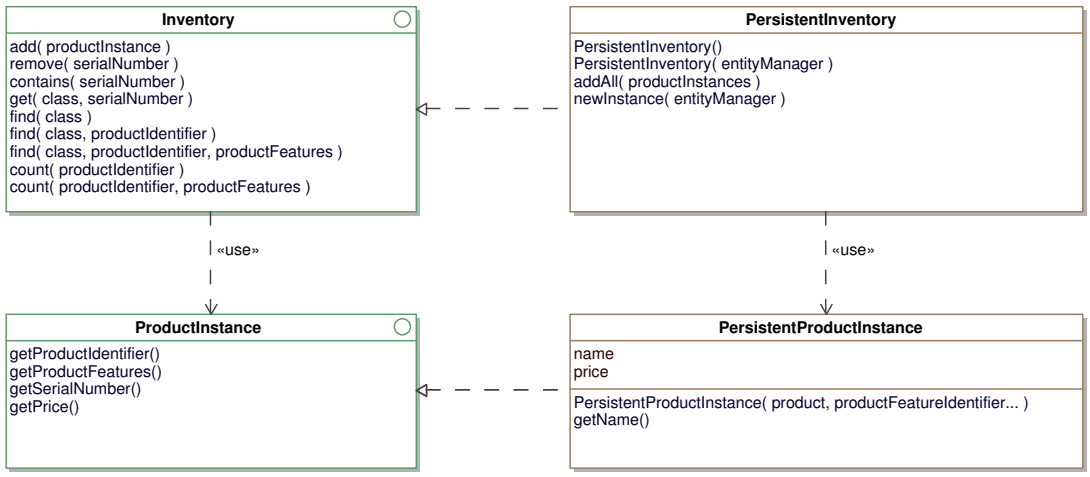

Figure 3.9: Inventory - Class Overview

`PersistentProductInstances` which match the specified `ProductFeature`s is done in Java code.

3.8 Accountancy

The accountancy package contains functionality supporting book keeping. `AccountancyEntry` is a representation of an accounting entry. `Accountancy` aggregates `AccountancyEntry`s. Every `AccountancyEntry` is uniquely identified by an `AccountancyEntryIdentifier`.

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

By implementing and sub-classing the `AccountancyEntry` interface, the notion of different accounts, as known from double-entry bookkepping, can be realised. As can be seen in Figure 3.10, `PersistentAccountancyEntry` is sub-classed to create a second "account" used to store payment information, namely `ProductPaymentEntry`.

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 attributes are named `userIdentifier`, `orderIdentifier`, and `paymentMethod` respectively. The inheritance hierarchy of `PaymentMethod` is depicted in Figure 3.11.

To create a new account, `AccountancyEntry` has to be sub-classed. Every (persisted) object of such a class belongs to the same account. Accessing per-account entries is

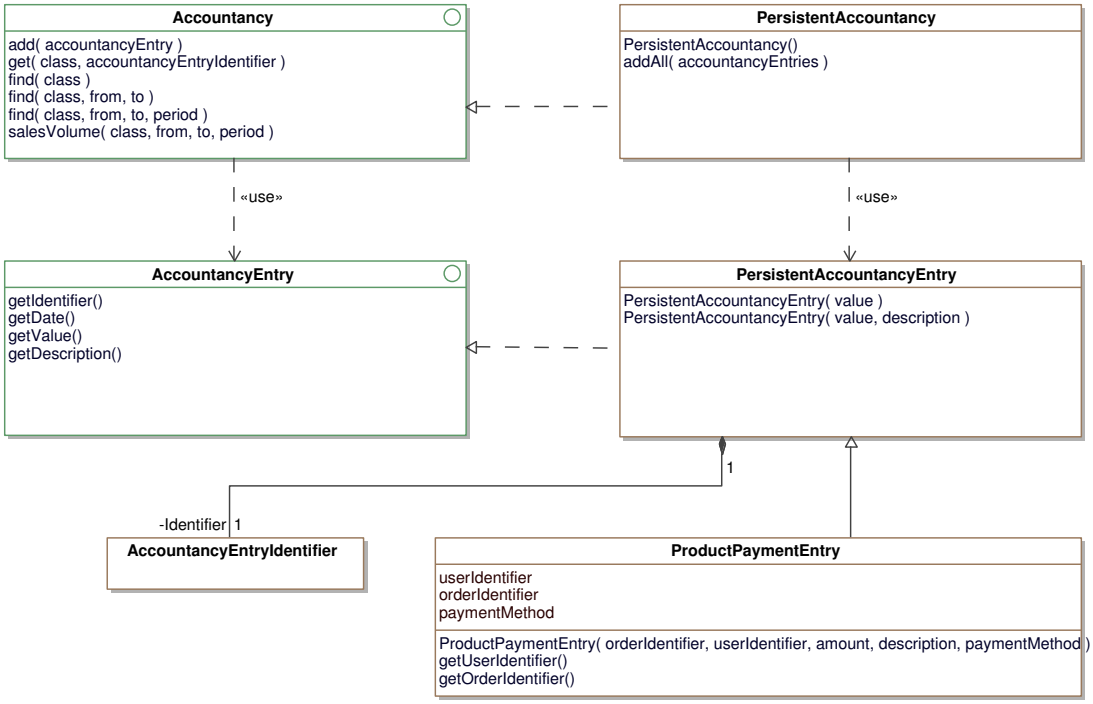

Figure 3.10: Accountancy - Class Overview

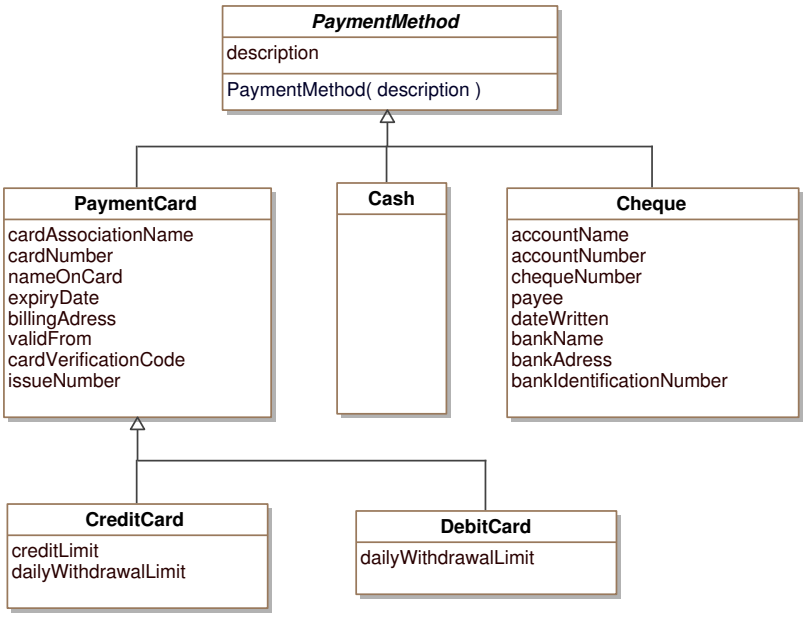

Figure 3.11: Payment - Class Overview

facilitated by specifiying the desired class type when calling `get()` or `find()` methods of `Accountancy` as explained in Section 2.5.4.

3.9 Order

An `Order` can be considered as a sheet of paper which basically consists of lines, each representing an ordered product. An order can be uniquely identified by an `OrderIdentifier`.

Every product of an order is stored in a separate `OrderLine`. An `OrderLine` in turn is uniquely identified by an `OrderLineIdentifier`. An `OrderLine` contains all information to identify a `ProductInstance` (see Section 3.5). A `ProductInstance` is identified by a `ProductIdentifier`, and an optional set of `ProductFeature`s.

A `ChargeLine` represents additional costs or discounts and can be applied to an `OrderLine` or an `Order`. For example, `ChargeLine`s 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 `OrderStatus`. The lifecycle state cannot be arbitrarily changed, but follows a fixed scheme and is represented as field `orderStatus` in the class `PersistentOrder`. State transistions are automatically carried out when certain methods are called on an `Order` object, for example `cancelOrder()`.

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()` method is called. `COMPLETED` is one of the final states and it is not possible to change the state of such orders.

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:

- `SUCCESSFUL`: 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.

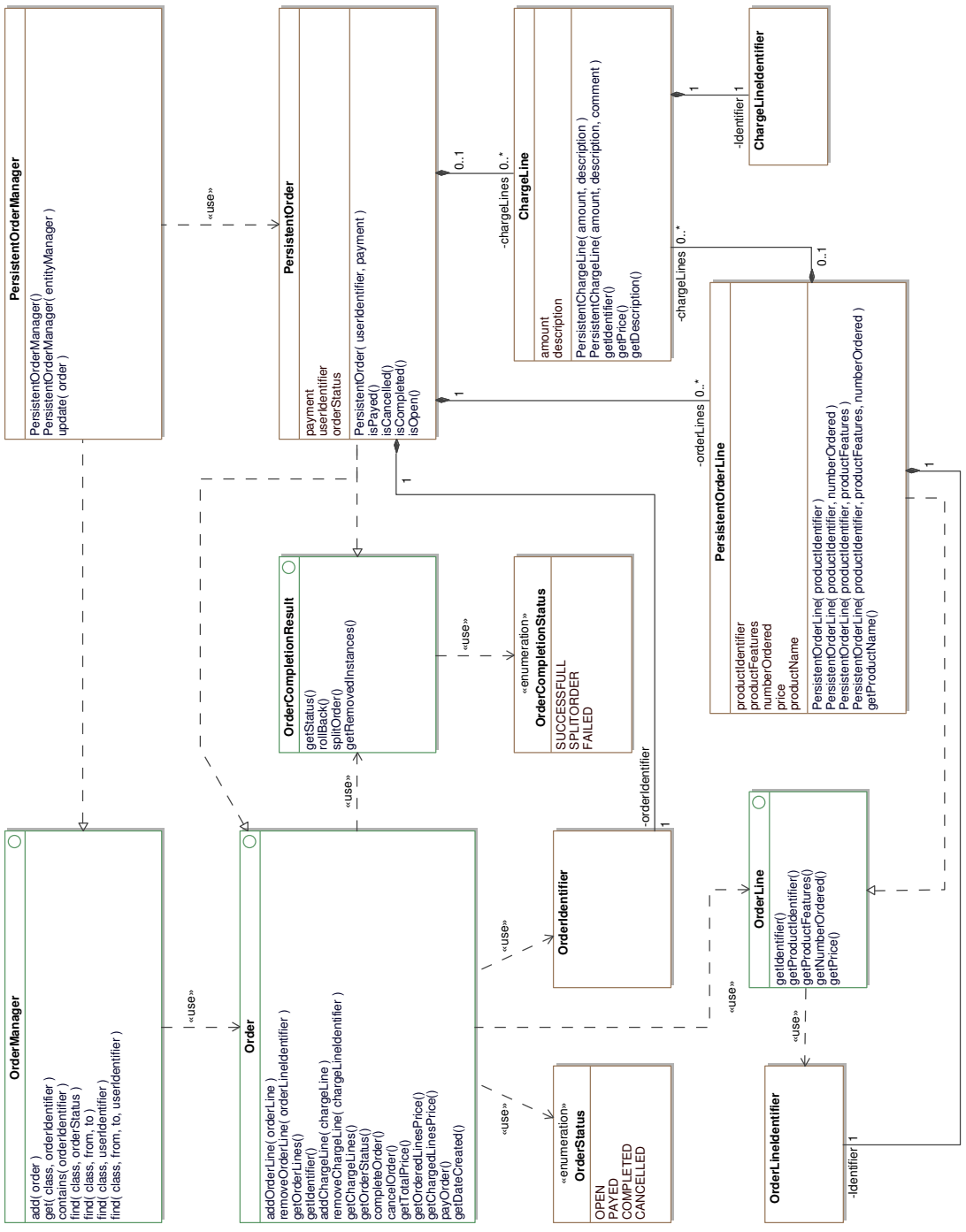

Figure 3.12: Order - Class Overview

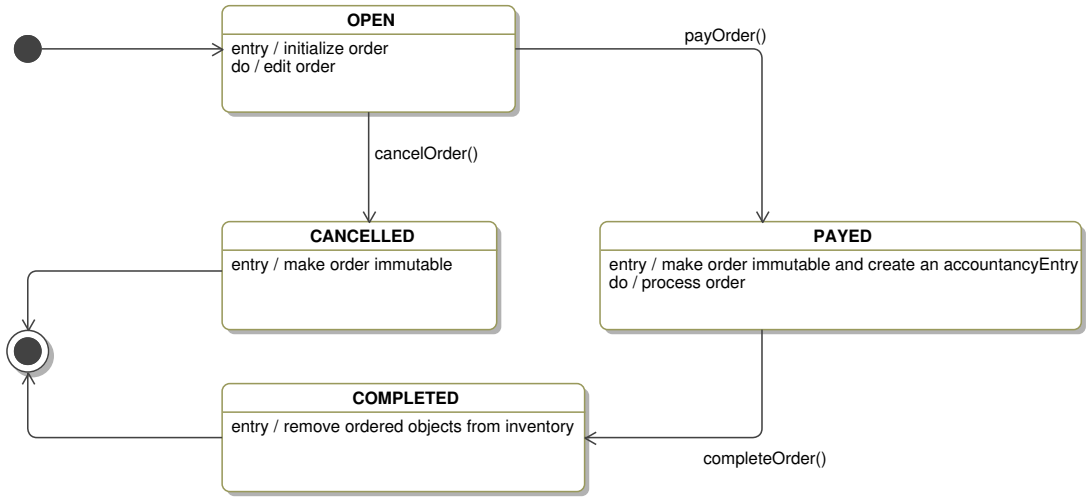

Figure 3.13: Order - Lifecycle

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 `Order`s. The implementations `PersistentOrderManager`, `PersistentOrder`, and `PersistentOrderLine` are used to persist, update, find and remove orders to/from the database. In `Order` aggregated objects, like `OrderLine`s and `ChargeLine`s 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.

[ecla] `http://www.eclipse.org/eclipselink/api/2.3/index.html`.

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

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

[HVE] Anders Hejlsberg, Bill Venners, and Bruce Eckel. The Trouble with Checked Exceptions. `http://www.artima.com/intv/handcuffs.html`.

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

[jpa] `http://jcp.org/aboutJava/communityprocess/final/jsr317/index.html`.

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

[swi] `http://java.sun.com/javase/technologies/desktop/`.

[swt] `http://www.eclipse.org/swt/`.

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