Salespoint 2011

Technical Overview

Hannes Weisbach

1st September 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 2011 is not taylored to any specific application, but designed with a wide area of applications in mind.

Models and design patterns employed in Salespoint 2011 are inspired by "Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML" by Jim Arlow [AN03]. An overview over the functionality of and new features in Salespoint 2011 is detailed in this document.

We would like to thank all Salespoint users who submitted their feedback and encourage future users of Salespoint 2011 to do the same.

2 Technical Background

In this chapter we want to give a short overview about the APIs and Frameworks used by Salespoint 2011.

2.1 Persistence

The persistence layer was one of the biggest problems in Salespoint 2010. We wanted a solution that acknowledges latitudes for persisting objects with relatively low effort.

Therefore and because of the huge community support, we decided to use the Java Persistence API (JPA). JPA is a Java Programming Language Framework managing relational data in applications. The API itself is defined in the javax.persistence package.

Some of the most popular vendors of JPA are Hibernate and EclipseLink. For developing Salespoint 2011 we used JPA 2.0 with EclipseLink without vendor specific functionality. So it should also be possible to use other JPA 2.0 vendors like Hibernate with this framework.

2.2 Joda-Time

Joda-Time is a Java date and time API. It provides a quality replacement for the Java date and time classes. We used this API because the Java Date and Calendar classes are badly designed. Furthermore Joda-Time is very easy to use and provide functionality to handle Java Date and Calendar.

2.3 Spring

Salespoint 2011 is mainly designed to develop Web- and Serverapplications. Therefore the framework provides basic MVC and tag functionality for login and capability management. To make the Web-Functionality easier to handle we used the Spring-Framework and included the core into Salespoint 2011.

3 Salespoint 2011 Core

This chapter overviews the core functionality of Salespoint 2011 and gives a short introduction.

3.1 Package Overview

The following diagram shows the most important packages of the Salespoint 2011 and their dependencies. These packages are detailed in the chapters below.

3.2 Shop

3.3 User

3.4 Calendar

3.5 Quantity

`Money` itself is a subclass of `Quantity` which inturn 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`).

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

3.5.1 `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.

3.5.2 `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.

3.5.3 `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 for 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.

3.6 `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`.

A `Money` object can be instatiated 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 is 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.

3.7 `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`).

3.8 Product

3.9 Catalog

3.10 Inventory

3.11 Accountancy

3.12 Order

Orders are a new feature in Salespoint 2011. The intention was to prepare the old databaskets for development of web applications and extend them to collaborate with our new inventory, providing the opportunity of individual pricing, final payment and basic log functionality.

Therefore we created the **OrderEntry** as main data structure to access these functionality. An OrderEntry represents one order and can be imagined as sheet of paper which basically consists of lines representing the ordered products (**OrderLines**) and lines that are not bound to a concrete product but which cause charge (**ChargeLines**). ChargeLines make it possible to add individual charge to orders. For example they can be used to define forwarding charges or other special charges produced by this order.

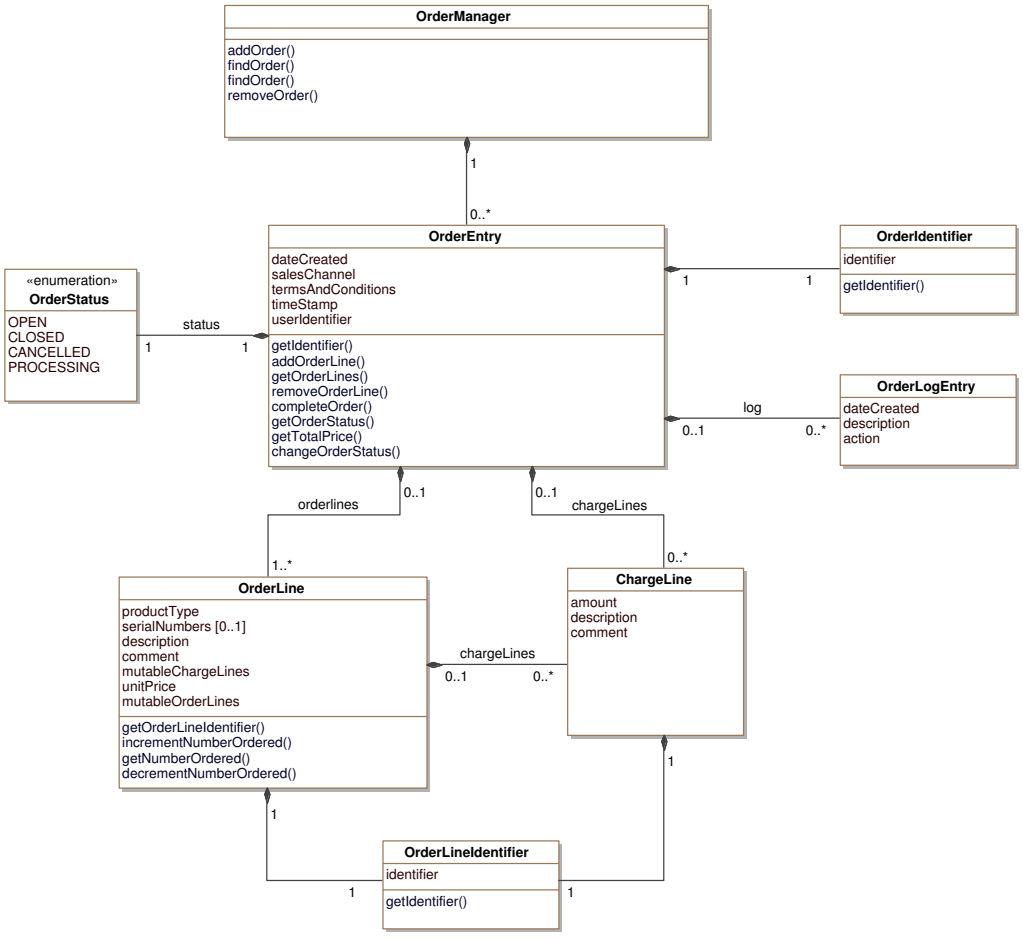

Figure 3.1: Order - Class Overview

Bibliography

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