Design and implementation of a standards based eCommerce solution in the business-to-business area

Thesis of Andreas Junghans
Department of Computer Science
Karlsruhe University of Applied Sciences, Germany

Written at ISB GmbH, Karlstrasse 52-54, 76133 Karlsruhe
03/01/2000 to 08/31/2000

Supervisor at Karlsruhe University of Applied Sciences:	Prof. Klaus Gremminger
Co-supervisor:	Prof. Dr. Lothar Gmeiner
Supervisor at ISB GmbH:	Dipl.Inform. Ralph Krause

Declaration

I have written this thesis independently, solely based on the literature and tools mentioned in the chapters and the appendix. This document - in the current or a similar form - has not and will not be submitted to any other institution apart from the Karlsruhe University of Applied Sciences to receive an academical grade.

I would like to thank Ralph Krause and Gerlinde Wiest of ISB GmbH for their support over the last six month, as well as Peter Heil and Frank Nielsen of Heidelberger Druckmaschinen AG for providing information and suggestions for the requirements analysis. Acknowledgements also go to Christian Schenk for his MiKTEX distribution and the Apache group for their XML tools. Finally, I would like to thank Professor Klaus Gremminger of the Karlsruhe University of Applied Sciences for his supervision of this thesis.

Karlsruhe, 3rd of September, 2000

(Andreas Junghans)

Contents

1 Introduction **1**
1.1 Document conventions . . . 1
1.2 Subject of this thesis . . . 1
1.3 Rough schedule . . . 2

2 Requirements analysis **3**
2.1 Tools used . . . 3
2.2 Domain requirements . . . 3
2.2.1 Clarification of terms . . . 3
2.2.2 Specific requirements on ”Business to Business” applications . . . 4
2.2.3 The pilot application . . . 5
2.2.4 Use cases . . . 10
2.2.5 Roles and access rights . . . 28
2.2.6 Domain class model . . . 31
2.3 IT specific requirements . . . 31
2.3.1 Server platforms . . . 31
2.3.2 System layers . . . 35
2.3.3 Business logic . . . 35
2.3.4 Data storage . . . 36
2.3.5 Data exchange . . . 36
2.3.6 Security . . . 37
2.3.7 Presentation . . . 39
2.3.8 Specific requirements for the pilot application . . . 40
2.3.9 Examination of Intershop enfinity . . . 41
2.3.10 Glossary of terms . . . 41

3 Design of an eCommerce framework **45**
3.1 Goals . . . 45
3.2 Tools used . . . 45
3.3 Overview . . . 45
3.3.1 Request/response pattern and schemas . . . 45
3.3.2 Framework approach . . . 46
3.3.3 Clustering . . . 46
3.3.4 Transactions . . . 47
3.4 Business process encapsulation . . . 47
3.4.1 Overview . . . 47
3.4.2 Contexts . . . 47
3.4.3 Process graphs . . . 48
3.4.4 Component design . . . 50
3.4.5 Limitations . . . 50
3.5 Data exchange components . . . 50

3.5.1 Overview . . . 50
3.5.2 Interface of the framework . . . 53
3.5.3 Structure of requests/responses . . . 53
3.5.4 Request processing . . . 53
3.5.5 Session context persistence . . . 55
3.5.6 Automation . . . 55
3.6 Data storage components . . . 58
3.6.1 Support by Enterprise JavaBeans . . . 58
3.6.2 Using XML schema to describe persistent datatypes . . . 58
3.6.3 The query component . . . 66
3.6.4 Limitations . . . 72
3.7 Security components . . . 73
3.7.1 Overview . . . 73
3.7.2 Authentication, privacy, integrity . . . 73
3.7.3 Access rights . . . 73
3.7.4 User management . . . 75
3.7.5 User management interface . . . 75
3.8 Domain components . . . 76
3.8.1 Overview . . . 76
3.8.2 Internationalization . . . 76
3.8.3 The catalog . . . 77
3.8.4 The shopping cart . . . 85
3.8.5 Orders . . . 87
3.9 Presentation components . . . 90
3.10 Glossary of terms . . . 91

4 Iterative development of the most important components **93**
4.1 Tools used . . . 93
4.2 Overview . . . 94
4.3 Developing a framework with Enterprise JavaBeans . . . 94
4.4 Data storage components . . . 95

5 Summary and assessment **96**

A XML schema for the framework (excerpt) **98**

B DTD for process graphs **100**

C Example XSLT stylesheets **102**
C.1 Stylesheet for generating the remote interface of an entity bean . . . 102
C.2 Stylesheet for transforming catalog pages to HTML . . . 105

D Bibliography **107**

List of Figures

2.1 Current business process at HDM Kiel . . . 7
2.2 Intended business process at HDM Kiel . . . 8
2.3 Actor hierarchy . . . 11
2.4 Use case overview . . . 11
2.5 Product catalog . . . 13
2.6 Start licenses . . . 14
2.7 Start licenses options . . . 14
2.8 Upgrade licenses . . . 15
2.9 Option licenses . . . 15
2.10 Products in the cart . . . 17
2.11 End customer information . . . 17
2.12 Configuration overview . . . 18
2.13 Additional order information . . . 19
2.14 Receive ordered products . . . 20
2.15 List of previous orders . . . 21
2.16 End customer management . . . 22
2.17 Privileged license key options . . . 24
2.18 User data . . . 26
2.19 Group membership . . . 27
2.20 User rights . . . 27
2.21 First domain class model (part 1) . . . 32
2.22 First domain class model (part 2) . . . 33
2.23 System layers . . . 35
2.24 Physical distribution of the pilot application . . . 41

3.1 Class diagram for sections 3.4 and 3.5 . . . 48
3.2 Sample process graph . . . 51
3.3 Java code created from figure 3.2 . . . 52
3.4 Sequence diagram for processing a request . . . 56
3.5 Example for a success message . . . 57
3.6 Example for an error message . . . 57
3.7 Example for a request bundle . . . 57
3.8 Example for a response bundle . . . 57
3.9 Sample schema fragment (framework) . . . 59
3.10 Sample schema fragment (framework customization) . . . 60
3.11 Generated request types . . . 60
3.12 Sample client request . . . 61
3.13 Sample response containing binary data . . . 63
3.14 Hierarchy of generated classes . . . 65
3.15 Sample query (SQL) . . . 67
3.16 Sample query (spiced with XML) . . . 68

3.17 Sequence diagram for executing a query . . . 71
3.18 Class diagram for security related classes . . . 73
3.19 Product query with access rights check . . . 75
3.20 Making a user member of a user group . . . 76
3.21 Revoking an access right from a user . . . 76
3.22 Class diagram for domain components . . . 77
3.23 Example for a multi-dimensional catalog hierarchy . . . 78
3.24 Example for a catalog page . . . 81
3.25 Request for catalog pages from the hierarchy . . . 82
3.26 Query by example for products . . . 83
3.27 Response containing catalog pages . . . 83
3.28 Request for a catalog hierarchy . . . 83
3.29 Response to figure 3.28 . . . 84
3.30 Request for putting an item into the shopping cart . . . 86
3.31 Request for displaying the shopping cart's contents . . . 86
3.32 Response containing the cart's contents . . . 86
3.33 Request for ordering a shopping cart . . . 88
3.34 Request for searching confirmed orders . . . 88
3.35 Response containing the contents of one or more orders . . . 88
3.36 Process graph for ordering a shopping cart . . . 89
3.37 Example for a catalog page . . . 90
3.38 Example for an auto-generated form . . . 90
3.39 HTML source of figure 3.37 . . . 91

List of Tables

3.1 Pre-initialized session variables . . . 57
3.2 Pre-initialized request variables . . . 57
3.3 Datatype conversions . . . 66
3.4 Fields of the `Dimension` bean. . . . 79
3.5 Fields of the `Value` bean. . . . 79
3.6 Fields of the `XMLPage` bean. . . . 81
3.7 Fields of the `PageProduct` bean. . . . 82
3.8 Fields of the `ShoppingCart` bean. . . . 85
3.9 Fields of the `ShoppingCartItem` bean. . . . 85
3.10 Fields of the `Order` bean. . . . 87
3.11 Fields of the `OrderItem` bean. . . . 87

Chapter 1

Introduction

1.1 Document conventions

Throughout this thesis, the following conventions are used:

- Important terms are *italicized* when first used.
- Within a glossary entry, *italicized* terms indicate that these terms have an entry of their own.
- All code fragments (Java, XML, and other) are formatted in `typewriter style`.
- Java methods are shown with trailing parentheses. Example: `someMethod()`.
- References are made by sections numbers rather than names. Example: see section 1.1.
- Figure and table numbering is done on a per chapter basis.
- References to figures and tables are made without page numbers, except the figure or table in question is more than 3 pages away from the reference.
- The names of dataclasses (see section 3.6.2) begin with a lower case letter. The names of Java classes begin with an upper case letter.

This document has been typeset using LaTeX2e in the MiKTeX 1.20e distribution.

1.2 Subject of this thesis

The company ISB has chosen eCommerce as an upcoming core field of their business. For this reason, ISB concludes a cooperation contract with Intershop that makes ISB a ProfessionalSolutionPartner of Intershop. The contract allows ISB to sell Intershop products as part of eCommerce solutions for their customers, as well as to customize them to meet individual customer requirements.

Due to their complexity, the products of Intershop are mainly intended for business-to-business applications. Therefore they are not generally suited for the creation of online market places. ISB plans to realize a prototypical eCommerce system for a pilot customer in the context of a thesis. The system serves as a market place for products that can be looked up via a catalog with different views and ordered over the internet. If possible within the limited time of the thesis, online payment processing should also be integrated.

Two possible realizations should be inspected: The first one uses Intershop products to implement the market place, the second one is based on open internet standards. The goal of the second approach is to produce a generally usable eCommerce framework that serves as the basis for the pilot application.

Since the project is planned together with one of ISB's customers (Heidelberger Druckmaschinen AG, Heidelberg, Germany) one important focus lies on conducting the thesis according to ISB's project processing.

This includes determining the domain requirements for the pilot application and, based on that, designing a data model for the efficient processing of queries and orders.

Goal of the thesis is to obtain statements relating to the demarcation of the possible realization variants based on the conceptual work a prototypical realization.

Period of time:	03/01/2000 to 08/31/2000
Contacts:	Dipl.Ing. Gerlinde Wiest Dipl.Inform. Ralph Krause

1.3 Rough schedule

All in all, about 125 working days are available for this thesis. The minimum and maximum time spent on the different aspects is planned as follows:

Requirements analysis (Chapter 2):

- minimum: 25 days
- maximum: 35 days

Design (Chapter 3):

- minimum: 25 days
- maximum: 35 days

Framework implementation (Chapter 4):

- minimum: 25 days
- maximum: 35 days

Implementation of the pilot application:

- maximum: 25 days

Extraordinary efforts:

- minimum: 15 days

Given these figures, the minimum effort is 90 working days, which means 35 days are available for free distribution over the planned activities. The maximum effort sums up to 145 days, which means that the maximum must not be exhausted for every part.

These figures of course give only a rough idea - also, overtime is not yet taken into account ;-). Since the emphasis lies on the theoretical work in chapters 2 and 3 and the actual implementation of the pilot application is optional, the schedule can be corrected relatively easy.

Chapter 2

Requirements analysis

2.1 Tools used

My first intention was to employ CASE tools (Computer Aided Software Engineering) already in the analysis phase. However, not a single one I inspected was really suitable for this purpose since they all lack support for "proper" diagrams significantly. *Rational Rose* has problems with aligning items, *Together* is only available at ISB as a free edition that only supports class diagrams, and *GDPro*, which looked very good at first, crashes on a regular basis. Finally, I ended up with *Visio Professional*. Although "only" a drawing application, support for some UML diagrams is already built in, and missing shapes can easily be created or found on the web. Visio is stable, flexible, and produces high quality output.

2.2 Domain requirements

The thesis of this topic is rather IT specific. For this reason, it is impossible to avoid some IT technical parts in this section. However, most of the requirements mentioned here are independent from the actual IT environment used as the basis of the developed framework.

2.2.1 Clarification of terms

eCommerce Abbreviation of *Electronic Commerce*. There are countless definitions of this term ranging from any business related electronic communication (including telephone and fax) to selling over the internet. According to [Spe99], the term originated about 1995 when the world wide web started to revolutionize, for better or worse, the global internet.

The German government defines eCommerce as "every kind of business transaction where the participants initiate and conduct business or trade goods and services electronically" ([BMW99]). The *Fraunhofer Anwendungszentrum für Logistikorientierte Betriebswirtschaft (ALB)*[1] claims that eCommerce arose from *Electronic Data Interchange (EDI)* and "denotes the use of any electronic media to perform transactions" ([ALB]). However, they require computers to be involved somehow to qualify for eCommerce. A more complex definition is given by the consulting company KPMG (`www.kpmg.de`):

> *Electronic commerce is the use of certain information and communication technologies to closely integrate various value adding chains or inter-enterprise business processes and to manage business relationships.*[2]

For the purpose of this thesis, eCommerce is defined as: **All activities which make use of the internet or its underlying technologies for commercial transactions.** This comparably narrow view is closer to what people really see as eCommerce than the explanations of administrative institutions, business schools, and consulting companies.

[1] application center for logistics oriented business management

[2] translated from German by the author

Business-to-business (B2B) eCommerce between companies, rather than between a company and consumers, is referred to as *B2B eCommerce*.

Other terms

B2C — Business-to-consumer. Refers to commercial transactions between a company and private consumers.

CRM — Customer relationship management. All performances concerning the relationships between a company and its customers. In a broad sense, eCommerce is part of CRM.

eBusiness — The term eBusiness was presumably invented by IBM's marketing department. Some see it as just another buzzword with the same meaning as eCommerce (for example [Sie]). Another wide spread view is that it refers to business-to-business eCommerce. In this thesis, the term eBusiness is not used.

Electronic shopping cart — The electronic or virtual shopping cart (just *cart* for short) serves the same purpose as a real life shopping cart or basket. The customer can place products into the cart he or she wants to buy. However, there is no obligation to buy the cart's content until the customer decides to do so. If he or she looses interest, they simply leave the shop and the cart gets discarded after a while.

eMall; internet mall — A collection of eShops that are accessible through a common address.

ERP — Enterprise Resource Planning. This encompasses all activities in a company with the aim to manage and plan its resources with the focus on the products sold by the company, the products bought from suppliers, financial transactions, and human resources. Today, ERP systems like SAP R/3 are the main tool to manage large companies.

eShop; internet shop — A virtual shop which allows goods, information, or services to be purchased via the internet. A catalog allows the customer to browse through or search for offered products that he or she then can place into a virtual shopping cart, order them online, and pay for them online or offline.

eProcurement — Procurement via electronic means, mostly the internet. eProcurement aims at the improvement of procurement by closely linking the according IT processes of a company and its suppliers. This includes letting the suppliers of a company access parts of their ERP data.

Portal — A web site that allows personalized access to various information and other resources. Goal of every portal is to bind customers so that their first entry point when looking for information or products is the portal. Many portal sites are operated by several partners that use them to advertise their products. The term is often misused to praise arbitrary web sites.

SCM — Supply chain management. This term refers to the management of suppliers and theirs products and is basically the same as eProcurement.

2.2.2 Specific requirements on ”Business to Business” applications

There are several requirements that are not or less important in B2C applications than in the B2B area. Many software companies (including Intershop, see 2.3.9) put the B2B label on their software mainly for marketing reasons without meeting any specific B2B requirements. The following paragraphs present the most important B2B specifics.

Personalization To handle inter company business processes efficiently, the users of a B2B system should be presented with a personalizable user interface or at least entry page. This allows for a fast navigation as well as a better integration of the business partner's processes. While personalization is also a characteristic element in many B2C eShops, it is probably more important in the B2B area.

Access rights In a business environment, access rights must be more fine grained than the ones for end customers. It may be desirable for suppliers to see stocks of their partners, but not for other products than their own. For this reason, a complex (but still manageable) access rights system must be included in a true B2B application.

Automated transactions There is often a need for non-interactive business transactions. In this thesis they are called *automated transactions*; another term often used is "silent commerce". Principally, this is the same as the "old" batch jobs, so there is nothing special about it. However, many modern eCommerce applications either don't provide non-interactive access or offer only a very low level interface to the underlying database. The framework developed in this thesis should provide an interface for automated transactions that resembles the possibilities of an interactive user. Most importantly, a common data exchange format must be found that can be processed and generated by a variety of programs. XML is very well suited for this purpose (see 2.3.5).

Privacy While privacy is desirable in every eCommerce environment, it is a must when B2B transactions are involved. More and more secret services specialize on economic espionage to gain advantages for the companies in their country. Therefore orders and other transactions that might be of interest to competitors must be encrypted.

Authenticity When ordering books over the internet, there is not much that can go wrong when the consumer's identity is faked (from the seller's point of view - the buyer naturally sees this differently). Basically, the shop doesn't care who orders something. In a B2B scenario however, there is a strong need to authenticate clients of an eCommerce application. Since the usual way of payment in this case is invoicing, a faked client identity can lead to significant financial losses when products are delivered (e.g. to a rented postal box) and never paid. Ways of secure authentication are discussed in section 2.3.6.

2.2.3 The pilot application

2.2.3.1 Introduction

Heidelberger Druckmaschinen AG (HDM), situated in Heidelberg, Germany, acts as a pilot customer who will use the framework developed in this thesis as the basis for a pilot application[3]. HDM manufactures products for the print industry, including printing machines, pre-press and post-press.

In the pre-press sector, many of their products contain or consist of software. To simplify the purchasing of optional features (options for short), HDM has developed a licensing system based on license keys. There is only a single data medium for each version of a product, containing all the possible features. To use a specific feature, however, one has to order a license key from HDM to unlock it. This is done by so called *Sales and Service Units (SSUs)* which are responsible for distributing HDM products to end customers. Only SSUs order from HDM, never end customers.

The latest development by HDM in this area are so called *unified license keys* which are intended to replace product specific licensing in order to simplify the whole licensing process. They consist of two parts:

1. The **license id** contains a *serial number* (e.g. dongle number or machine id) together with some additional information that is updated on every unlock process. It identifies a *license controller* that restricts

[3]HDM have not yet decided if they will actually use the framework.

the use of software options to the ones actually unlocked. The license controller can be part of a local software installation or used as a server application for network wide use.

2. The **coded license key** contains all the options currently unlocked (current key) or the options to be unlocked (new key), respectively. There is one coded license keys for permanent licenses while there can be an arbitrary number of keys for time-limited licenses. However, several time-limited licenses with the same expiration date can be coded into one key.

The advantage is that SSUs can order several options for a customer and receive only a single coded license key to unlock them.

Basically, there are three different kinds of licenses. A *start license* is needed to get a product running at all. It is actually a fixed combination of a number of options. *Upgrade licenses* are needed if a new version of a product is installed and enable the continued use of already paid options. Finally, *option licenses* allow for the unlocking of additional options of an already installed and running product. All mentioned types of licenses require the current license id and coded license key in order to generate a new coded license key (note that the customer only has to enter the coded license key while the license id is updated on its own). An exception to this is when a first key is generated for a specific license controller since there is no current key in this situation. Start and option licenses can be time-limited.

Two special types of licenses are *demo* and *service licenses*. A demo license enables the temporary use of a complete software installation without a license key. Usually, the demo mode is active when the software is delivered to enable its use. The demo mode is deactivated when a license key is first entered or after a certain amount of time (usually 30 days). A service license can also be used to unlock the complete system for a limited time but, unlike a demo license, requires a license key.

2.2.3.2 Current situation

Figure 2.1 shows the current business process used to order Heidelberg software. Actually, every order involves two processes: First, HDM order management receives an ”ordinary order” for a software package (full version or upgrade) or for unlocking options of installed software. Order management enters the order into SAP which triggers invoicing. Second, a request for the corresponding license key to unlock the options ordered is made. HDM helpdesk (situated in Kiel, Germany) receives this request via fax or Lotus Notes form and checks if the specified options have indeed been ordered by the requesting SSU. If everything is correct, a license key is generated and sent to the SSU.

To enable use of full versions or upgrades before the time consuming license request process is finished, the software runs in a full featured but time-limited mode until the key is entered or, which is not supposed to happen, the time limit is over. Some products also come with a license key from the start, so the split order process described is only necessary for unlocking options.

The problem with the current process lies in the workload helpdesk has to handle. Additionally, many request forms are not filled out correctly so helpdesk has to check back with the SSUs. Furthermore it’s not clear to the SSUs why they have to place an order and then request a license key separately.

A solution to this is the *Heidelberg Internet Licensing System (HILS)*. HILS is a web application using applets to request license keys online. It is currently used to license two of HDM’s products, CP-2000 and CPC-32. While HILS already improves the process shown above, it has several drawbacks:

- HILS is proprietary for every product. There are currently two products licensed via HILS, each with its own user interface for requesting a license key. This makes it very difficult (and expensive!) to integrate new products. However, this is not the fault of the developers but rather caused by the diverging licensing strategies of different HDM products. These approaches are intended to be replaced by the unified license keys discussed above.

- HILS was developed by Linotype Library GmbH. HDM itself does not have control over the source code. Linotype Library calculates about 80,000 DM for the integration of *every* new product.

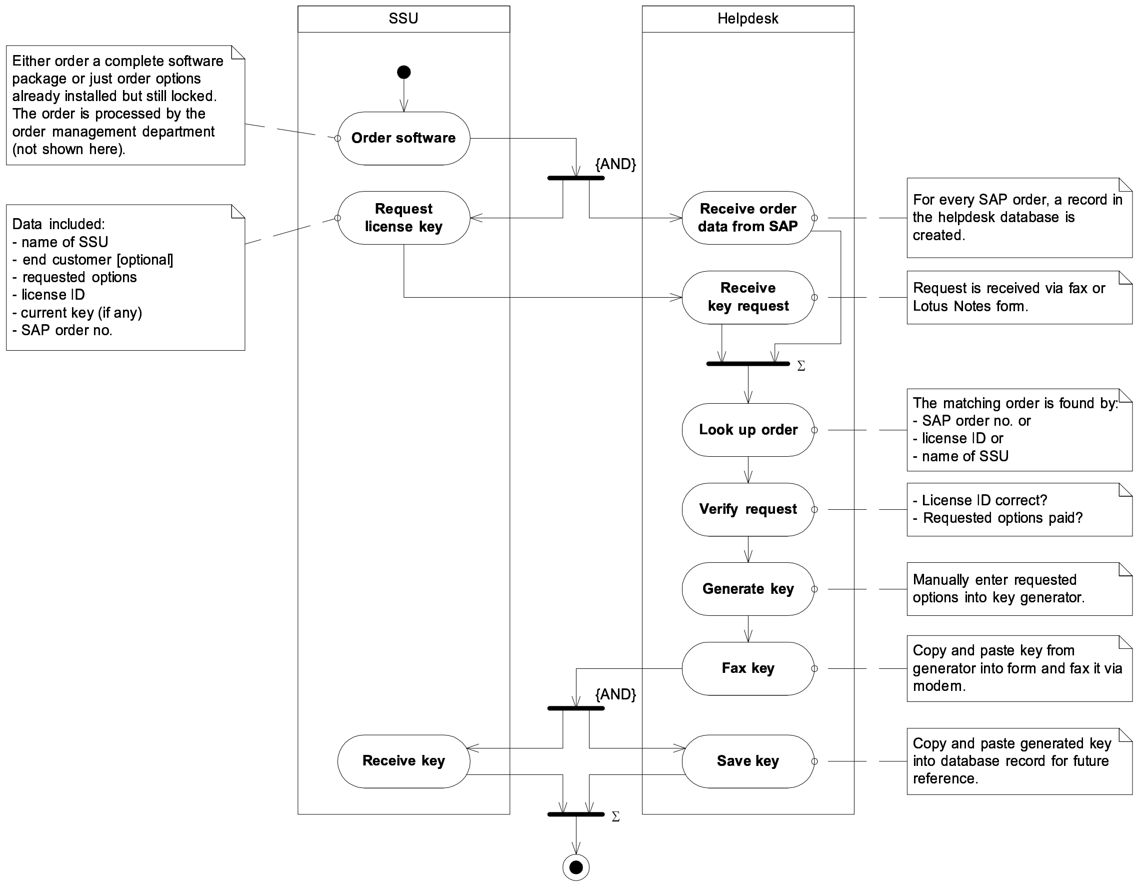


Figure 2.1: Current business process at HDM Kiel

- HILS uses Java Applets for the presentation of a rather simple user interface. It is not compatible with older browser versions (even if they support applets), not very stable, and takes a long time to load. A GUI based on HTML/JavaScript would probably be the better solution.
- HILS doesn't support automatic generation of orders in SAP (or other ERP systems). All orders have to be manually entered into SAP.
- HILS cannot be easily extended to support online orders of "physical" products (like software packages).

For these reasons, HILS is not suitable for future online licensing. This is the point where the pilot application presented in this thesis comes in.

2.2.3.3 Goals for the pilot application

Figure 2.2 shows the process that should be made possible by the pilot application. An SSU should be able to order license keys online via internet or extranet. After they have placed an order, they can directly download the license key or have it sent to an arbitrary eMail address or fax number. The online order causes the pilot application to place an order in the ERP system (SAP) which then sends an invoice to the SSU. Note that there is only a single order without the need for an additional request like shown in figure 2.1.

In the beginning, the pilot application is intended only for ordering license keys, no "physical" products. The goal of this first phase is that SSUs order actual software packages for a very low price covering the production costs for CDs, manuals, etc. When they actually sell a software to an end customer, they order a

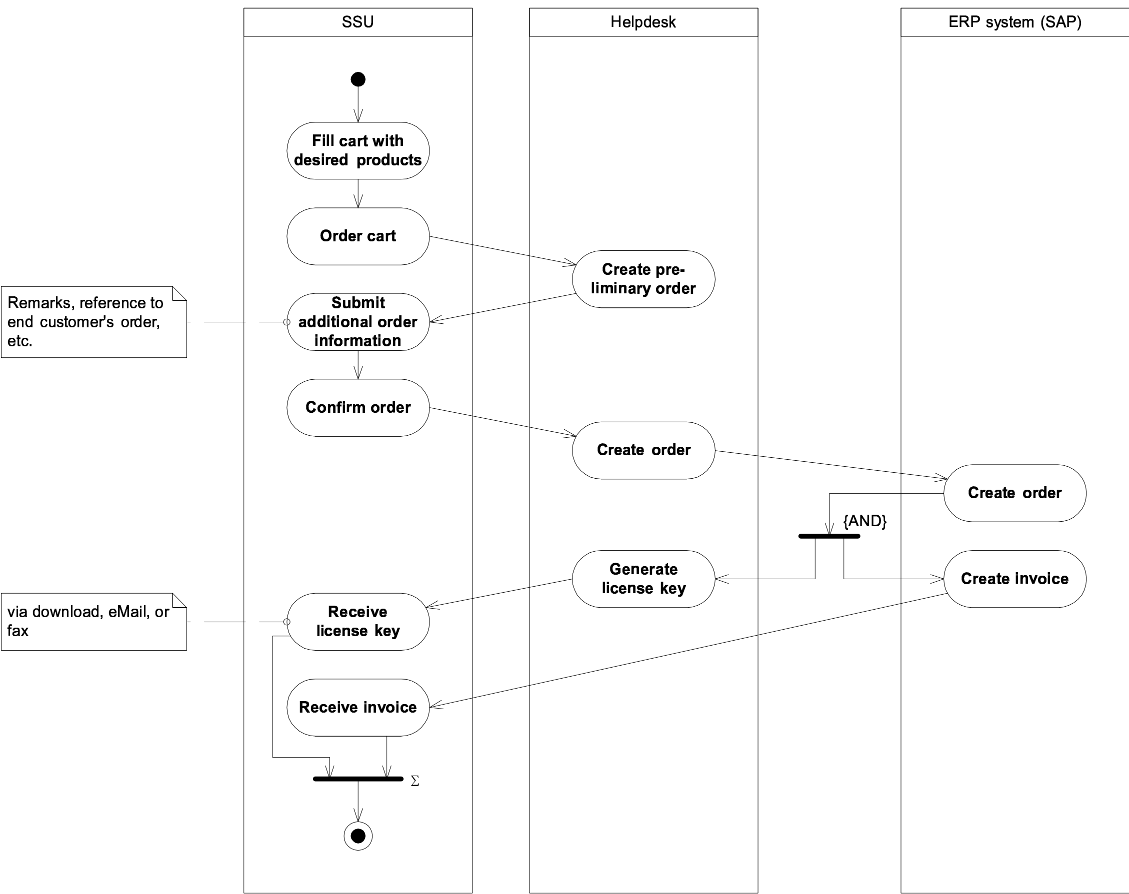


Figure 2.2: Intended business process at HDM Kiel

license key via the pilot application. In a later version, it should be possible to order also the software packages online. Another requirement for later versions is the possibility to transfer option licenses from one machine to another. This is useful when a certain feature is used on different workstations over time.

In addition to being used by SSUs, the pilot application can also serve as a means for helpdesk to provide SSUs with keys if they cannot or do not want to use the application themselves under certain circumstances. Besides helpdesk, there are some more "internal customers", such as R&D and the test department, that are also allowed to generate keys (with or without triggering invoices). This approach should replace the separate key generators used in every department. By using a central server, HDM has more control over the key generation and distribution. To support manageability, the pilot application must allow the definition of user groups and the assignment of appropriate access rights to users and groups.

2.2.3.4 Relationship to this thesis

The pilot application serves as an example for using the developed framework. Many of HDM's requirements fit well into the context of this thesis. The most important ones are:

- The customers using the system are sales agencies (SSUs) that again sell HDM's products to end customers. They are somewhat privileged regarding access to HDM's IT infrastructure, but are still independent companies that are not on the same level as internal HDM departments. Together with the following items, this makes the application "B2B".

- The pilot application not only involves electronic ordering but also *electronic shipment* of products, license keys in this case. From a more general point of view, this means trading with information which is a field of increasing importance in the information society.
- Users of the system are assigned different access rights. For example, a product manager can ”order” (without invoicing) a dozen keys for his product that can be used by the developers and testers. They might be permitted to access these keys but not to generate new ones on their own. Also, some SSUs may be granted access to beta versions while others are not.
- Online orders should be propagated to an ERP system (SAP in this case).
- The pilot application must be very reliable (24/7). Imagine a crash in the middle of the (European) night while sales agencies in the US or Asia try to get license keys they urgently need. This would not mean losing a tremendous amount of money, but the application might well be extended to products other than license keys in the future. Maybe at this point downtime really means cost. Thus a reliable and scalable platform is needed for the application and the framework it is based upon. Application servers are very well suited for this purpose. They offer distribution over several hardware nodes and independence from hardware and OS platforms. Other aspects are explained in more detail in section 2.3.

The approach in this thesis is an abstract one. It should be possible to use the developed framework in a wide range of eCommerce applications. The specifics of the pilot application are detailed after the general part of each section. Where examples are used (screenshots and the like), most of them are taken from the pilot application.

2.2.3.5 Quantities

1. Orders: 30 to 70 per day
2. SSUs: about 10 in the beginning, up to about 100 later that will use the system
3. End customers per SSU: 1000 to 2000 for big SSUs

2.2.3.6 Glossary of terms

Coded license key — A hexadecimal code that is used to identify or unlock the options that are usable on a software installation. For unlocking options, the coded license key must be made known to a license controller.

Current (coded license) key — Latest coded license key used for a software installation (managed by a license controller which is identified using a license id).

Demo license — A license that enables use of a complete software installation for a limited amount of time. No license key is needed. Every software installation starts with a demo license until expiration or a license key is entered.

License controller — Part of a local application or used as a separate server software to supervise the use of licensed options.

License id — Uniquely identifies a license controller.

Option — Software feature that can be unlocked using a license key.

Option license — A license that enables the use of a specific option on a software installation. Several option licenses can be combined in a single coded license key. Option licenses can be permanent or time-limited.

Service license A software installation can be put into the so called service mode by using a service license key. For a limited amount of time, all options are unlocked.

SSU Sales and Service Unit - a regional sales center.

Start license A license that enables the permanent use of a software installation with some of its options. One start and several additional option licenses can be combined in a single coded license key. Start licenses can be permanent or time-limited.

Upgrade license A license that enables the use of an upgraded software installation with all options that were in use before the upgrade.

2.2.4 Use cases

Goal of this thesis is to design a framework for eCommerce solutions. The following use cases present only the functionality that should be provided directly by the framework (without any programmatic additions or modifications) and the modifications and extensions necessary for the pilot application. They are not intended as a prescriptive or complete description of *all* systems that can be built using the framework.

2.2.4.1 Actors

All actors that play a role in any of the following use cases are listed below. Their inheritance relationships can be seen in figure 2.3.

Administrator A user with the right to access all data in the system and to manage all users and user groups.

Customer A user who places orders.

ERP system An *Enterprise Resource Planning* system like SAP R/3. This system is used to retrieve product information and/or place orders.

External customer A customer who, in general, is being charged for his or her orders.

Internal customer A customer who, in general, is not being charged for his or her orders.

Manager A customer with additional privileges (like managing certain other users and groups of users).

Prospective customer An entity (person or program) that is not yet a user of the system. A prospective customer may access parts of the system anonymously.

User Every entity (person or program) with the possibility to login to the system.

Managers are basically users with extended access rights. There is no sharp delimitation between them and other customers. Some managers may be granted rights that others don't have. In general, the manager actor is used where there are actions involved that should be forbidden for ordinary customers.

The actors listed with an individual use case are the ones that typically interact with the system in the way described there. However, other actors may also be able to act in this way. For example, an administrator can obviously search the product catalog, but this is not what he or she usually does.

Heidelberg specific The term *customer* always refers to a SSU or an HDM department. End customers are always referred to using the term *end customer*.

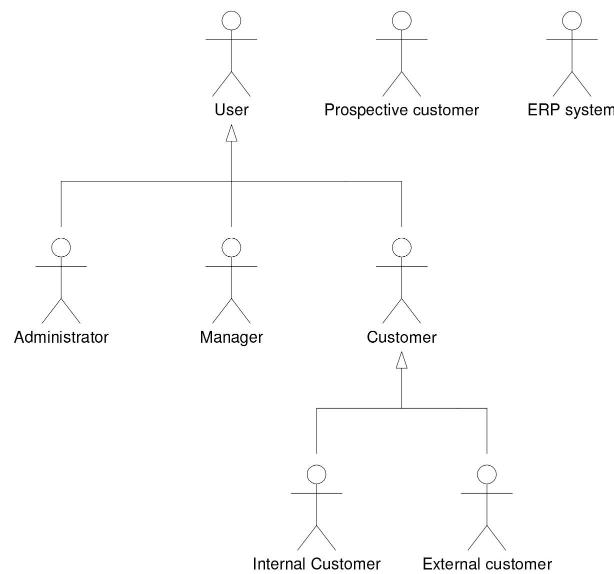


Figure 2.3: Actor hierarchy

2.2.4.2 Overview

The following sections present two kinds of use cases. The first one (numbered 1.x) is from the user's point of view, the second one (numbered 2.x) is from the administrator's point of view. Figure 2.4 shows the usual course of activities when using the system and the corresponding use cases. Administrative use cases are not included in the figure since they are only loosely related.

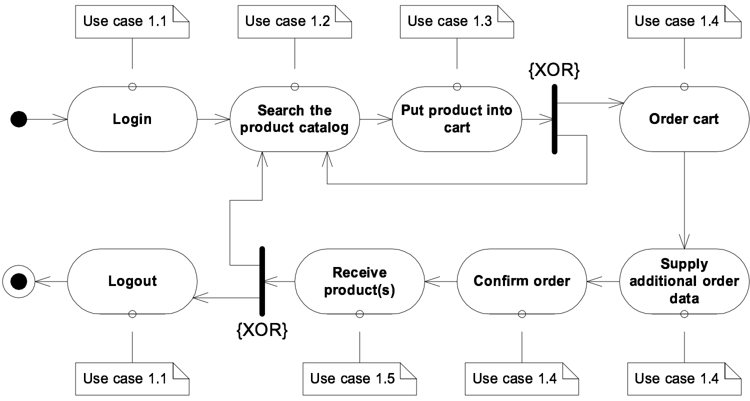


Figure 2.4: Use case overview

2.2.4.3 Use case 1.1: Login to and logout from the system

Actors User

Description Before a user can access the system, he or she has to authenticate him or herself. Note that a user can access the system as a prospective customer before actually logging in, but then he or she is not acting as a user.

Possible ways of secure authentication are discussed in section 2.3.6. If a PIN/TAN approach is used, the user may have to authenticate him or herself again before sensitive transactions (like placing an order). Logging out is optional. If a user doesn't logout, this is done automatically after a timeout period.

Heidelberg specific HILS (see section 2.2.3.2) uses a one time password concept based on hardware authenticators (see 2.3.6). A PIN enables the use of a specific authenticator. A one time password is then generated by the authenticator and afterwards used to login to the system. There are no additional challenges after a successful login. Since this works well and is already introduced at the German SSU, this concept may also be used by the pilot application. An alternative is to use digital certificates.

2.2.4.4 Use case 1.2: Search the product catalog

Actors Customer, Prospective Customer, ERP system

Description The customer can use a multi-dimensional hierarchy to search for products he would like to order. There are several *dimensions* describing a certain property of a product, e.g. ”Software/Hardware” or ”Area of use”. Every dimension has a set of possible *values*, e.g. ”Software, Hardware” or ”Graphics, Audio, ...”. For example, the customer can find graphics software by clicking through one of these trees:

1. All products - by Software/Hardware - Software - by Area of use - Graphics
2. All products - by Area of use - Graphics - by Software/Hardware - Software

For convenience, the dimension entries like ”by Area of use” can be left out so only the possible values are displayed. Restrictions may prevent the display of certain dimensions above or below others in the hierarchy. Note: A ”normal” catalog hierarchy is a special case of the described multi-dimensional hierarchy. The customer can also search the catalog by product properties (like names, prices, etc.) and keywords.

After selecting an entry in the catalog hierarchy or searching for products, a list of matching *catalog entries* is displayed (figure 2.5). There are two flavors of entries:

1. A *short catalog entry* is used in lists together with other short entries, e.g. as a result of a successful search.
2. A *long catalog entry* is displayed as a single page and usually contains more information than a short entry. It’s the right place to offer customization options to the user or let him or her put several products into his or her cart at once.

This categorization does not describe the precise length or layout of the catalog entries. For example, there can be several kinds of short entries, depending on the actual application. Every entry (long or short) is associated with one or more products (e.g. software on CD and license key for the same software). Every entry can contain links to a number of other entries. Usually, short entries will point to long ones. Both kinds of entries can contain links that allow either direct downloading of a product (free of charge) or putting it into the cart. What information about a product is displayed depends on the actual application. Normally, at least price and availability are shown.

Access to the catalog can be restricted. For example, only authenticated users may be allowed to view certain catalog information or to put products into their cart. The latter restriction is useful to allow prospective customers to see what they could buy if they were registered with the system.

The ERP system is the source of product numbers, prices, names, and, where appropriate, descriptions. It’s also possible to maintain a separate product database if no ERP system is available or for some reason cannot or should not be used.

Heidelberg specific There is no need for a multi-dimensional catalog in the pilot application. The search function can be omitted too since there is only a small number of well known products. The catalog has no connection to an ERP system but uses a separate database. It doesn’t contain prices, and all products are always available. Most descriptions are auto-generated and simply consist of the product’s name.

A start license is represented by a long catalog entry listing the possible basic configurations (figure 2.6). From this catalog entry, the customer can open an overview page showing the differences between the possible

start licenses in detail (figure 2.7). Upgrade licenses for a product are also displayed using a long catalog entry (figure 2.8). They simply allows the selection of one upgrade license out of a list of choices. Finally, every product version has a number of option licenses that can be ordered (figure 2.9). These are represented by a single long catalog entry page that allows to put an arbitrary number of every option into the shopping cart. Alternatively, a request key[4] can be entered to put a number of options into the cart. Note that licenses can only be combined as long as a single coded license key can be generated out of them (see Use case 1.3).

Technically, all available options are individual products belonging to a "product family" (e.g. Prinergy) and version (e.g. 2.0).

There are no prospective customers in the pilot application. Every customer has to login before he or she is allowed to access the catalog.

Figure 2.5: Product catalog

[4] A request key is basically a license key, but not containing installed or purchased options, but rather an "option wish list" generated by the licensed software itself.

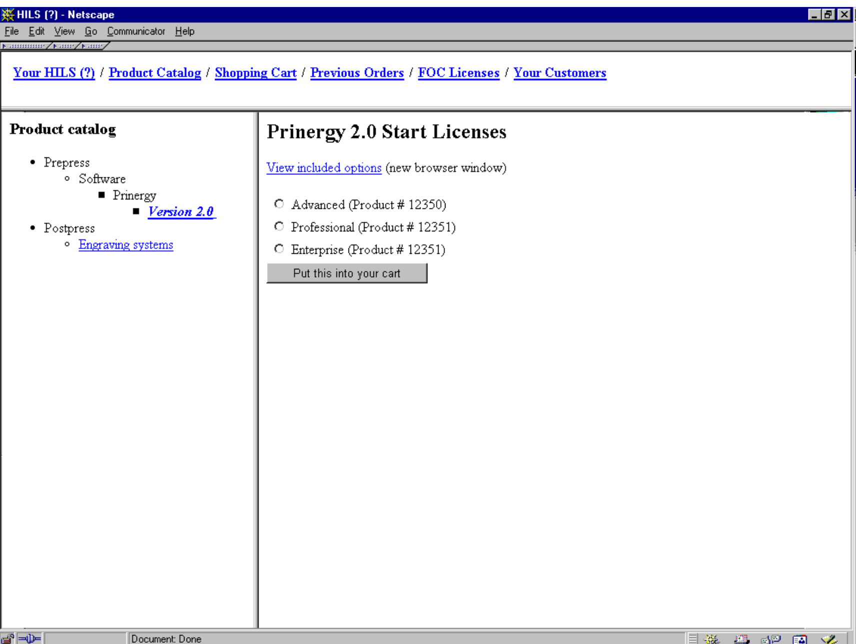
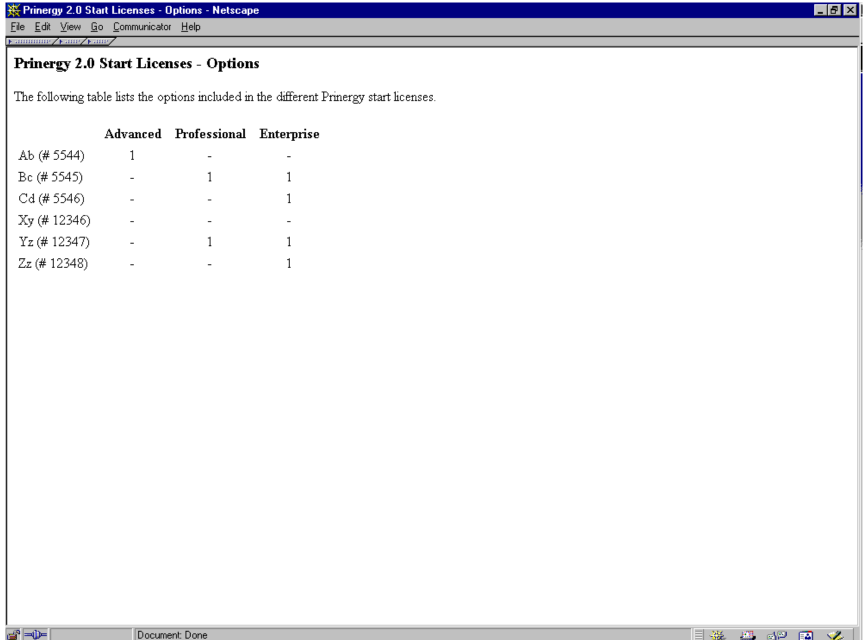


Figure 2.6: Start licenses



	Advanced	Professional	Enterprise
Ab (# 5544)	1	-	-
Bc (# 5545)	-	1	1
Cd (# 5546)	-	-	1
Xy (# 12346)	-	-	-
Yz (# 12347)	-	1	1
Zz (# 12348)	-	-	1

Figure 2.7: Start licenses options

Figure 2.8: Upgrade licenses

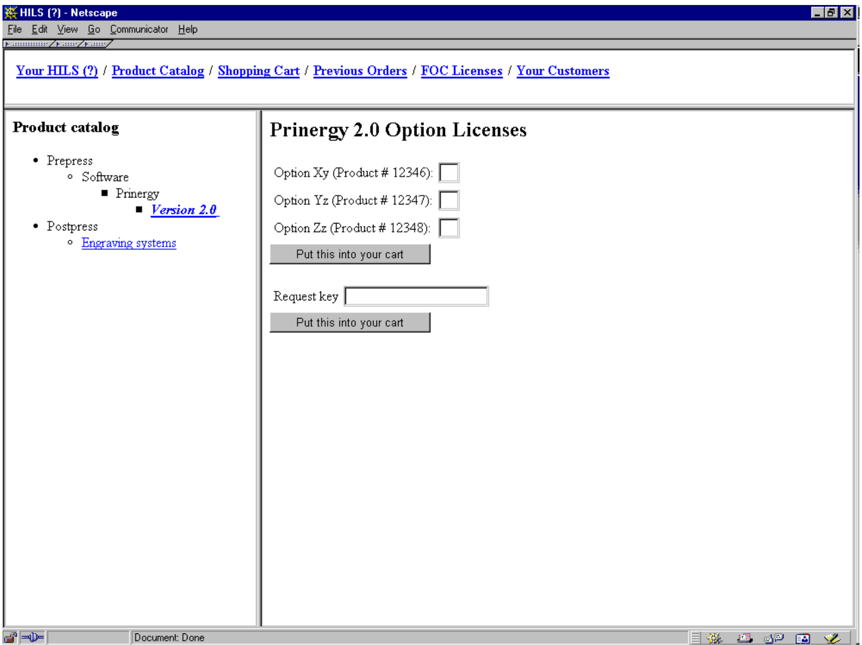


Figure 2.9: Option licenses

2.2.4.5 Use case 1.3: Using the shopping cart

Actors Customer, Prospective customer

Description Every time the (prospective) customer puts a product into the shopping cart, its contents is displayed in a table (figure 2.10). If a product is customizable, the choices made can be seen in this table. The customer can change the quantity of any position (if sensible) or delete it from his or her cart. No actual order or reservation is made at this stage.

When the customer is satisfied, he or she places an order for the cart's contents. This is the latest possible time a prospective customer must login the system. When a cart is not used for a certain amount of time (e.g. 30 minutes), it is automatically removed.

Heidelberg specific Using the coded license keys described in section 2.2.3, several positions in the shopping cart lead to a single key being generated. To simplify ordering the cart and receiving the key in the pilot application, one cart can contain only items that *together* lead to *one* key. If the customer wants to order keys for two different products (or two keys for the same product but different end customers), he or she has to subsequently fill and order two carts. This may be inconvenient but is far less confusing than putting everything in one cart and then specify what positions should be grouped together to get single keys. As a consequence of this concept, one cart is not only limited to one product and its options, but also to a single kind of key (permanent, time-limited, service).

Before a cart can be ordered, the customer is asked to enter the end customer he or she wants to place an order for (figure 2.11), including the current license id and key. This data must be provided as a prerequisite for generating a new key that includes the ordered options. Non-privileged customers are only allowed to order permanent licenses. Time-limited and service licenses can only be generated by managers and administrators (see Use case 2.3).

As a side effect, the current key allows for an overview over the end customer's configuration. On a separate page, the current configuration (read from the current key) is shown together with the options added by the products currently in the cart (figure 2.12). There is one column for the current configuration and one for every product in the cart, except for single options which are summarized in the last column. Note that a quantity of two or more start licenses (e.g. ”Prinergy Advanced”) or other option packages leads to only one column on this page.

Users with sufficient privileges can directly generate license keys from the shopping cart, thus effectively using the system as a key generator front-end.

There are no prospective customers in the pilot application. Every customer has to login before he or she is allowed to use the shopping cart.

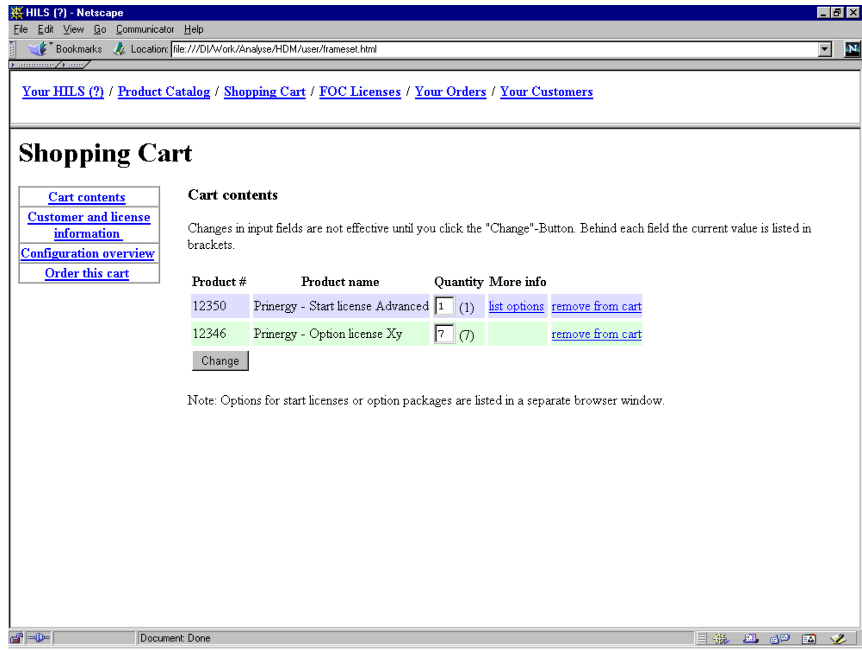


Figure 2.10: Products in the cart

Figure 2.11: End customer information

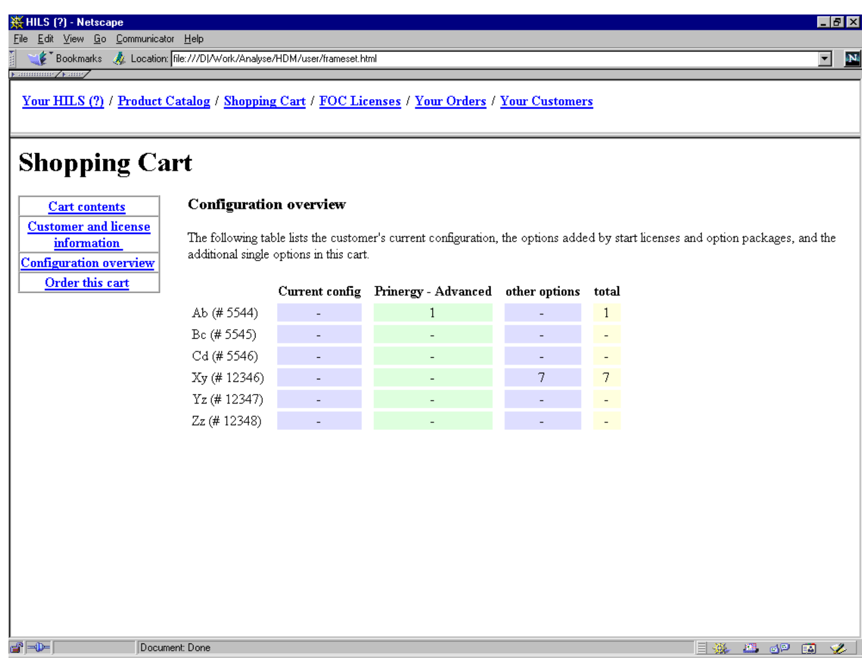


Figure 2.12: Configuration overview

2.2.4.6 Use case 1.4: Placing an order

Actors Customer, ERP system

Description When a customer decides to order the shopping cart's contents, a preliminary order is generated. Before the actual order can be placed, the customer must supply additional data, like payment information, that is needed to process the order. After that, he or she is able to confirm the order. The order can still be canceled at any time before a confirmation is made. Unlike shopping carts, preliminary orders are not deleted after some time but stay in the system until confirmed or canceled, thus making it possible to look for additional data and enter it at a later time. The system can also be configured in a way that a certain customer can only place a preliminary order that must be confirmed by another user (see Use case 2.4).

After the customer has placed an order, it is forwarded to the ERP system to trigger invoicing and, when physical products are involved, delivery. Under which circumstances confirmed orders can be canceled depends on the actual application.

Heidelberg specific There is no need for payment information at all in the pilot application. All SSUs allowed to access the pilot application are charged by invoice. The address for invoicing is maintained together with the SSU's other data in the SAP system and does not have to be entered with every order.

The end customer's data can still be changed after a preliminary order is placed. However, this is not recommended to avoid ordering products that don't fit with the end customers current configuration (see Use case 1.3).

Every order has additional data associated with it that can be used to relate it to an end customer's order or to enter remarks (figure 2.13). Also, every product in a specific version can have additional fields that may be or have to be filled out by the customer (remember that every cart and therefore every order is associated with one product and it's options). To keep the user interface simple and consistent, such product specific fields should only be used if necessary. Some fields may be editable even after the order has been placed, while some may be not. In any case, changes to them are logged so the original values can be viewed any time.

A confirmed order cannot be canceled without involvement of HDM. If such a cancellation is requested, someone at HDM has to change the order status manually.

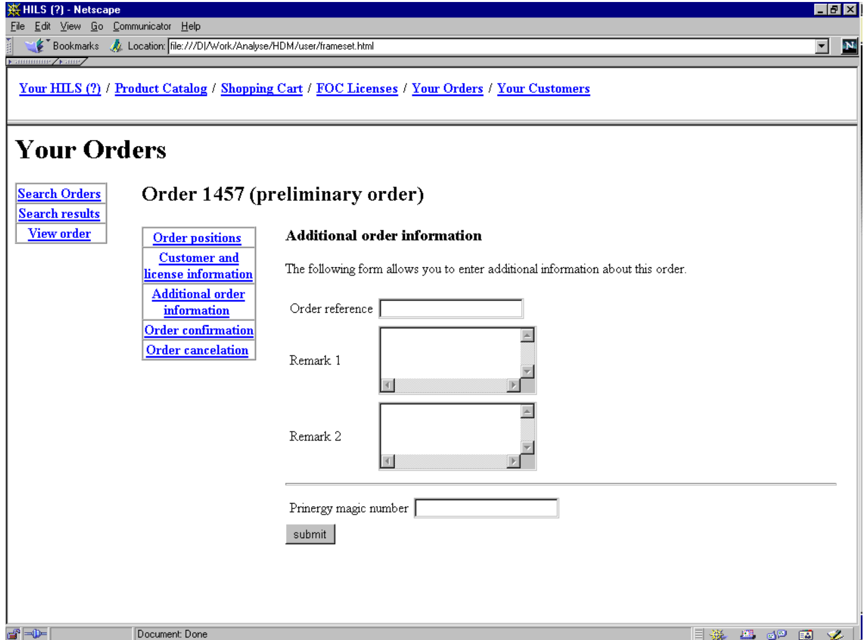


Figure 2.13: Additional order information

2.2.4.7 Use case 1.5: Delivering the products

Actors Customer, ERP system

Description If the customer ordered products that consist of digital information (like license keys), he or she can directly receive them after placing the order or after the money charged has been transferred. The customer can choose which way he or she wants to receive the products (figure 2.14). The delivery of "physical" products is triggered by the ERP system.

Heidelberg specific Every order leads to the generation of a single coded license key. This key can be received as plain text or as a license certificate including the key and end customer data. The key can be downloaded or received via eMail or fax.

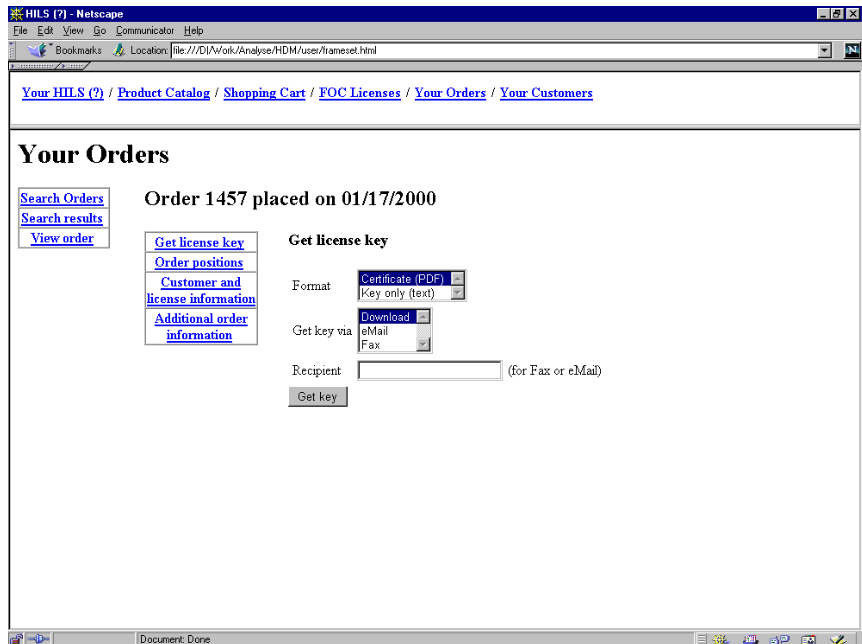


Figure 2.14: Receive ordered products

2.2.4.8 Use case 1.6: Viewing previous orders

Actors Customer, Manager, Administrator

Description A customer can view his or her previously placed orders and preliminary orders. A list shows all orders and the most important data (like order date) associated with them (figure 2.15). Selecting an entry from this list opens the same view as if the order had just been placed, with the possibility to download digital information products and view all order data. The customer can also search his or her orders and download order data for further processing.

Managers can view/search/download orders of all the users/groups they manage. Administrators can view/search/download all orders of all users. Managers and administrator can also upload orders, provided they have sufficient access rights (see section 2.2.5).

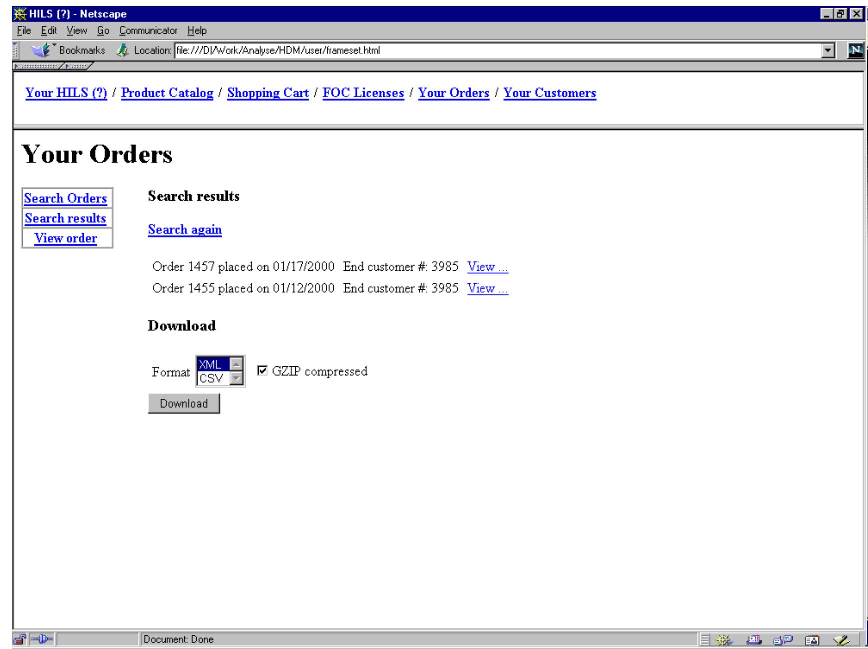


Figure 2.15: List of previous orders

2.2.4.9 Use case 1.7: Managing end customers

Actors Customer, Manager, Administrator

Heidelberg specific A customer can search, create, edit, and delete end customers (figure 2.16). The data entered can then be used to specify the end customer when ordering products (see Use case 1.3). Managers and administrators can also manage end customers of other users, provided they have sufficient access rights (see section 2.2.5).

End customer data can also be uploaded and downloaded for several end customers at once (see section 2.3.5).

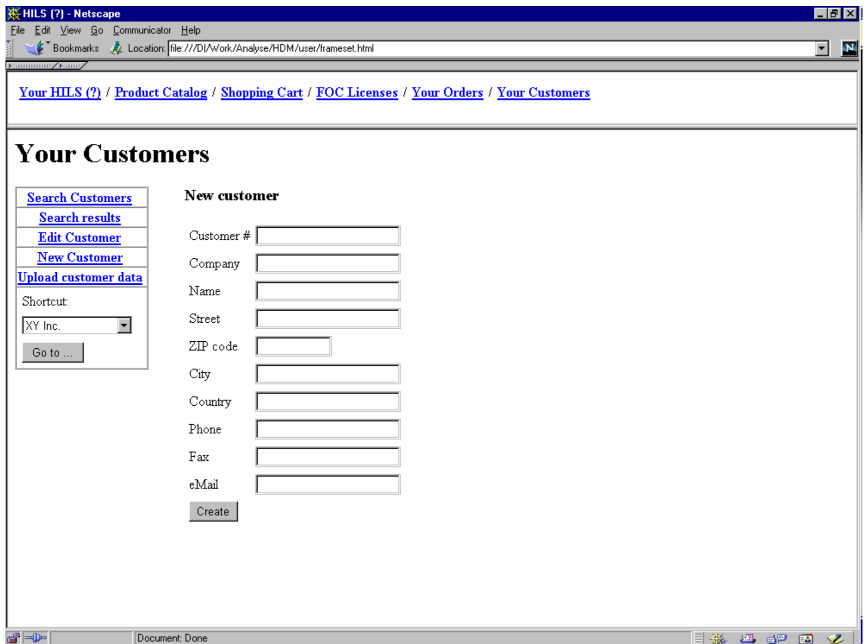


Figure 2.16: End customer management

2.2.4.10 Use case 1.8: Getting a certificate with new end customer data

Actors Customer

Heidelberg specific If a product is sold or given away for free from one end customer to another, it is possible to generate a new license certificate with the new end customer data on it.

2.2.4.11 Use case 2.1: Editing the catalog

Actors Manager, Administrator

Description The only sensible way of managing medium size to large product catalogs is via external tools (e.g. Content Management Systems). The catalog pages should be formulated in some XML dialect and transferred to the system via its automation interface. The categorization of products with respect to the catalog hierarchy should be part of the XML pages and processed automatically when the systems receives them.

In cases of small systems and for maintenance reasons, there must be a simple user interface that allows for editing catalog data and the creation of a uni-dimensional, simple catalog hierarchy and the placement of products into it.

Heidelberg specific The catalog pages are auto-generated from product description files (with the option to edit them afterwards). A simple catalog hierarchy is used in the beginning that may later be extended towards multiple catalog dimensions.

2.2.4.12 Use case 2.2: Placing orders for other users

Actors Manager, Administrator

Description It is sometimes necessary to place an order for a user that he or she cannot place him or herself. An example is a service delivery that is made free of charge but involves data the user should be able to collect in the same way as with ordinary orders. If a user has the right to place orders of this kind, an additional field allows him or her to select whom they want to place an order for. They can also select which user should be charged (if any), provided they are granted the right to do so.

Heidelberg specific In the pilot application, a department manager (e.g. of the R&D department) can generate keys free of charge that members of the department then can download. If these members don't have the right to order keys themselves, generation and distribution of keys can be efficiently managed.

If the manager or administrator has the right to do so, he or she can defer entry of all or some of the license information (like the current key) to the customer they place an order for (figure 2.17). When they first want to download the license key, they have to complete the license information.

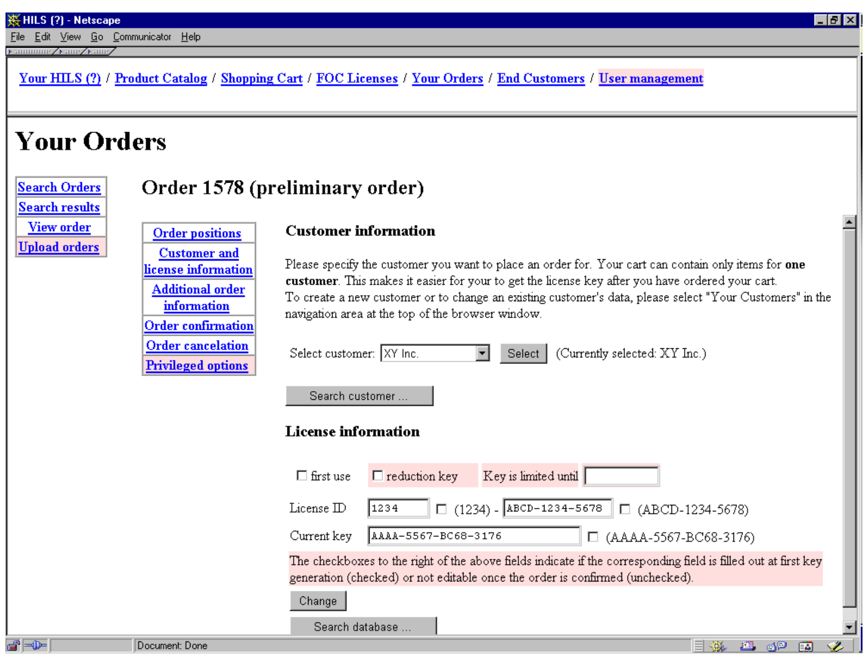


Figure 2.17: Privileged license key options

2.2.4.13 Use case 2.3: Ordering time limited or otherwise special products

Actors Manager, Administrator

Description Privileged users can fix a time limit for software products to allow for demo versions and the like (figure 2.17). Care should be taken that this feature is not abused!

Heidelberg specific If a user has the according access right, he or she can create a *reduction key* that takes away options from a user's software installation. This is necessary to support transfer of options from one machine to another. Further support for option transfers is planned for later versions of the application.

2.2.4.14 Use case 2.4: View and/or edit orders of other users

Actors Manager, Administrator

Description A manager or administrator can view information about and retrieve digital products associated with orders placed by others (see Use case 1.6), provided he or she has sufficient access rights.

To support different order processing workflows on the customer's side, managers and administrators can edit preliminary orders of other users. For example, a user may be allowed to place an order but not to enter payment information or confirm the order. A Manager then has to confirm the order before it is actually in effect. He or she may in turn not be allowed to place an order.

2.2.4.15 Use case 2.5: User management

Actors Manager, Administrator

Description Managers and administrators are responsible for managing users and groups of users. There can exist several *types* of users to allow for different information stored with each one. For example, the type ”business partner” may include an address for invoicing in its data while the type ”department manager” may include the name of the department. Note that the user type doesn't determine the actual access rights.

A user group is an abstract concept to simplify administration. It simply consists of a name and associated access rights. A user can be assigned to an arbitrary number of groups. In places where a list of users and/or groups must be specified (e.g. when granting someone the right to view orders of other users), ”All users” can be used. Instead of adding every user individually to the list, this allows someone to see orders of all current and future users. To simplify handling of user groups, their organization is not hierarchical, i.e. a group cannot be member of another group. While a group hierarchy might be desirable in certain cases, it's usually not necessary and may lead to confusion.

User management consists of the following activities:

- Create a new user. A user name must be specified, an expiry date for the account is optional. After selecting the desired type of user (e.g. business partner), data specific to this type (e.g. address for invoicing) has to be entered. After all necessary data is provided, the user can finally be created, and access rights, groups belonged to, etc. can be specified (see next item).

- Edit a user. The data entered on user creation can be modified if necessary (figure 2.18). Also the groups a user belongs to can be changed (figure 2.19), as well as his or her access rights (figure 2.20; for further details see section 2.2.5). If no longer needed, a user can be deleted.

- Search for a user. The search can be restricted to user name, expiry date, type of user, groups belonged to, and access rights. When searching by access rights, all users with the specified rights are found, regardless whether they have these rights by themselves or via a group they belong to. If needed, this can be restricted to users that have these rights by themselves.

- Create a user group. Only a name has to be specified.
- Edit a user group. The name of the group can be changed (all users still remain members of this group), and the access rights of this group can be specified (see section 2.2.5). If no longer needed, a group can be deleted.
- Search for a user group. A group can be found by name and/or by its access rights.
- See a list of users currently logged in.

Access rights are managed *user centric* rather than *object centric*. To explain this, take the following example scenario: User X should be granted access to a new beta version of product Y. In the user centric approach, the rights of X are displayed on the screen. The manager selects ”Allowed products” and adds ”Y” to the appearing list. The object centric approach would be to select ”Y” from a list of products and then ”Add user” to allow a certain user access to it. Both approaches can be queried in the opposite direction. For example, all users with access to a certain product can be looked up in a user centric system, or all objects a user has access to can be searched in an object centric system. However, the user centric approach seems more convenient, especially when a new user is created - all his or her access rights can be specified right away. Its also less likely to overlook a granted access right that should in fact be denied.

User and group information may be maintained in a directory (e.g. LDAP based), independent from the eCommerce application. In this case, only the assignment of access rights would be handled in the application.

User or group information can be uploaded and downloaded for several users or groups at once (see section 2.3.5).

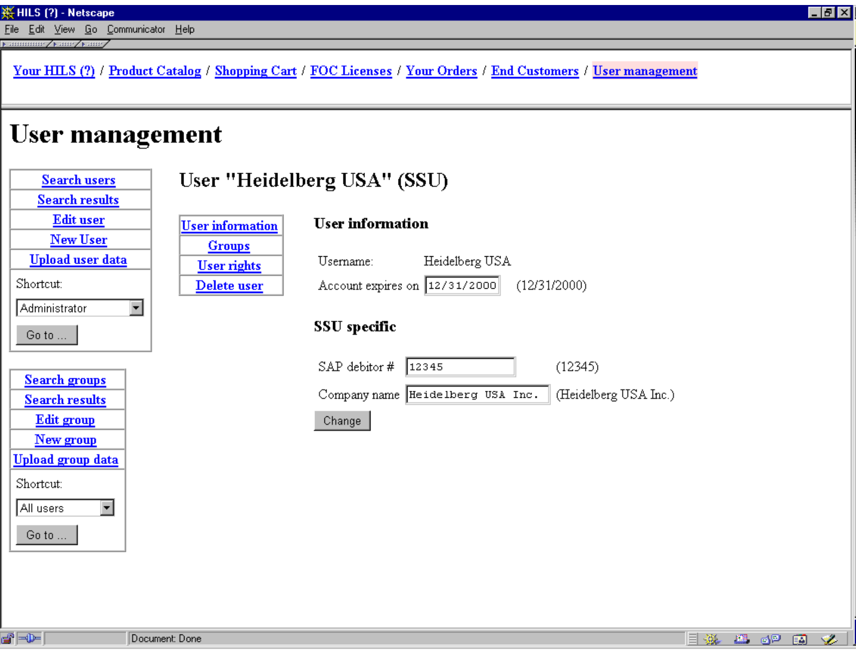


Figure 2.18: User data

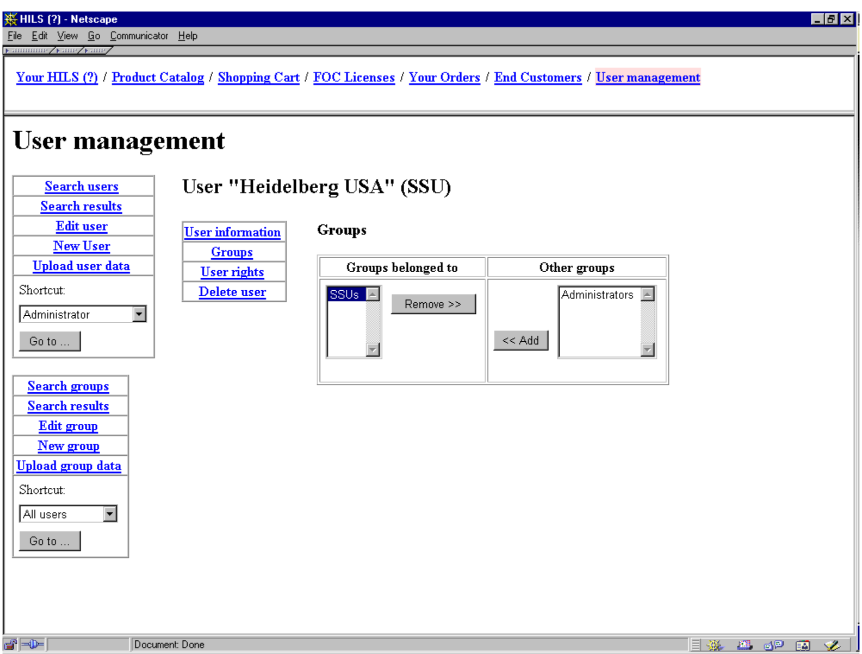


Figure 2.19: Group membership

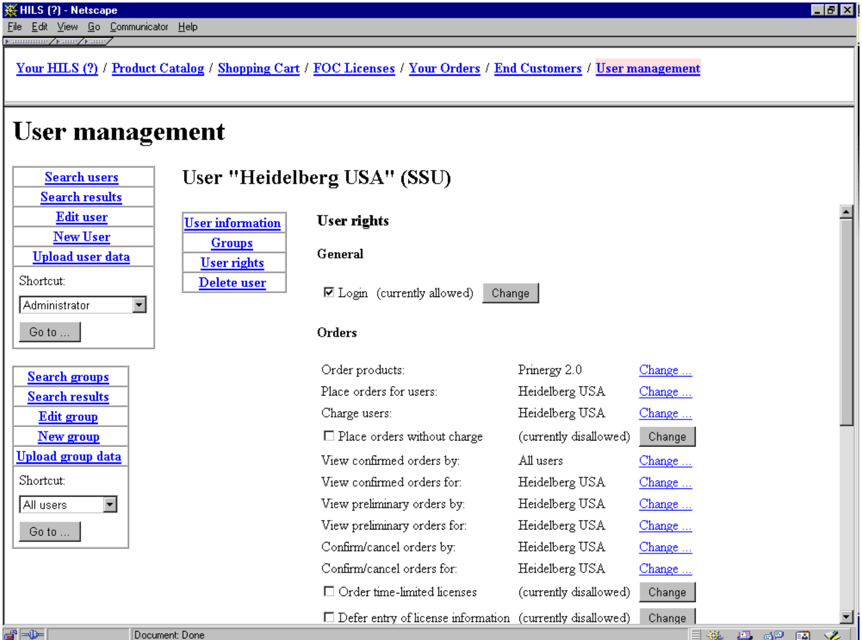


Figure 2.20: User rights

2.2.5 Roles and access rights

2.2.5.1 General access restriction concept

Many modern security concepts (like the one introduced by Java 2 Enterprise Edition, [Sunb]) use the notion of *roles* instead of users and user groups. The basic idea is to take the users and groups that are already established in an existing IT environment (e.g. held in a directory service) and map them to a set of roles. Access to functions or resources can be restricted to certain roles. This way, the user management doesn't have to be implemented many times and there is only a single record representing a user.

The problem with the role based approach is that this concept is not always appropriate. As an example, consider the right to view orders. Using a role based approach, you could say the functionality ”view my own orders” is accessible to roles user and admin, while ”view all orders” is restricted to role admin. The main problems with this are:

1. The same functionality (viewing orders) is split over several functional units (”view my own orders” and ”view all orders” in this example). Also, consider a request like ”view all orders above x EUR” (or other queries). If a user makes this request, only his or her orders must be returned; if an admin makes the request, all matching orders should be returned. This results in the new functional units ”view my orders above x” and ”view all orders above x”.

2. It's not easily possible to create a new manager role that may look at orders of certain groups of users (one group per manager).

Both restrictions result from the ”boolean” schema of roles. Either a right is granted (the user is member of the corresponding role) or not. In the example, a better approach is to have a single operation ”view orders” and appropriate access rights per user or user group. The default right for all users would be ”view orders created by [id of the user him or herself]”. An admin would have to right to ”view orders created by [all users]”. A manager's right would be to ”view orders created by [id of the managed group]”. All query results are filtered according to the rights of the current user. The pattern for access checks is not ”does the current user belong to role [...]” but rather ”does the current user (directly or indirectly) have the right to do [...]”.

The conclusion is that roles can help in identifying necessary access rights but are usually not enough for systems with more than basic needs. For this reason, users and user groups are used here. Roles can easily be substituted by user groups, while these groups allow for greater flexibility.

The effective rights of a user are the combined rights of all the groups he or she belongs to and the individual user rights. This means that a user can have a certain right because one of his or her groups has it or because he or she has been granted this right individually.

There are two predefined users: *anonymous* is not really a user but stands for everyone accessing the system who is not logged in. This user can be assigned rights as any other one. The second predefined user is *admin*, which is an administrative account with all rights[5]. The admin account cannot be edited in any way. The *system administrator* who installed the application must be able to reset the admin account, so it is accessible after a password or private key is lost.

Three main roles/groups can be identified in an eCommerce system:

1. A *Customer* uses the system to purchase products.

2. A *Content manager* is responsible for the presentation of products to the customers.

3. An *Administrator* is responsible for supervising the system, including rights management.

The following items explain the most important access rights. In a B2B environment, there are a lot of rights that depend on the actual application. It is not possible to foresee all of these, so the following items include only standard access rights that are suitable for most applications. They are sorted by the roles

[5]There may be exceptions to this rule, for example private user data that the admin may delete but not view

mentioned above. This is, however, just a rough classification. Most of the presented rights are not just true/false but can be seen as a scale on which a mark determines the actual right. This means that giving a customer the rights listed under 2.2.5.2 to their full extent would certainly make him or her a kind of administrator rather than a customer.

2.2.5.2 Customer rights

1. Right to login (default: disallowed[6]). This can be used mainly for two purposes: First, users can be disallowed to use the system anymore. This is not only checked when logging in, but also every time they send a request to the system. Second, special users can be created to share orders (and consequently downloadable products) among other users without the need to place an order for each of them. The receivers of the products are granted read access to the special user's orders while a department manager is granted write access. The special user itself is not allowed to login for security reasons. It is simply used for sharing products.

2. Right to order certain products. Either a list of allowed products or "all products" can be specified here. "All products" only includes products currently known to the system. The default is no products at all.[7]

3. Right to place orders for (i.e. in the name of) certain users. A list of users and/or groups can be specified (default: only the user him or herself). This is useful for administrative purposes.

4. Right to charge certain users for orders. A list of users and/or groups can be specified (default: only the user him or herself).

5. Right to place orders without charge (limited to certain products; default: disallowed).

6. Right to place a certain number of orders (or orders with a certain value) per time period. Default: zero.

7. Right to view confirmed orders placed for (i.e. in the name of) certain users. A list of users and/or groups can be specified (default: only the user him or herself).[8]

8. The preceding item for preliminary orders. Default: view only preliminary orders for (i.e. made in the name of) the user him or herself. To actually view a certain preliminary order, also the next right must be granted for this order.

9. Right to view preliminary orders made *by* certain users. A list of users and/or groups can be specified (default: only the user him or herself). To actually view a certain preliminary order, also the previous right must be granted for this order.

10. Right to confirm or cancel orders *for* (i.e. made in the name of) certain users. A list of users and/or groups can be specified (default: only the user him or herself). To actually confirm or cancel a certain preliminary order, also the previous right must be granted for this order.

11. Right to confirm or cancel preliminary orders made *by* certain users. A list of users and/or groups can be specified (default: only the user him or herself). To actually confirm or cancel a certain preliminary order, also the next right must be granted for this order.

12. Right to order time-limited products (limited to certain products; default: disallowed).

[6] This is important to ensure that access rights can be specified after a user was created, but before he or she is able to login.

[7] Earlier versions of this document stated that access to products could be restricted using catalog dimension values. This has been removed for simplicity. The example given (beta versions of software products) can be realized easier by a dedicated access right. Earlier versions also stated that "all products" included future products. This has been changed for security and design reasons (see section 3.7.3).

[8] Earlier versions of this document included the right to view orders placed *by* certain users. This is not particularly useful and would complicate the design, which is why it was removed.

13. Right to upload orders. This can be used to simplify bulk order placement. The restrictions above still apply (e.g. limitation to certain products). Default: uploading orders is disallowed.

2.2.5.3 Content manager rights

1. Right to create new products (default: disallowed).
2. Right to edit certain products (default: no products).
3. Right to create/edit catalog information for certain products (default: no products).

2.2.5.4 Administrator rights

1. Right to view information about certain users. A list of users and/or groups can be specified (default: no one). The user information includes the type of the user (e.g. business partner), the user account's expiration date, the groups belonged to, and the user's access rights.
2. Right to edit information about certain users. A list of users and/or groups can be specified (default: no one). The user information includes the type of the user (e.g. business partner), the user account's expiration date, the groups belonged to (which can be further restricted, see next item), and the user's access rights. The access rights that can be granted are limited to the ones of the user granting them.
3. Right to put users in certain groups. A list of groups or "all groups" (which does *not*[9] include future groups) can be specified (default: no group).
4. Right to create new users of certain types. A list of user types can be specified (e.g. business partner, department manager, etc.). Default: no type (i.e. no new users can be created at all).
5. Right to view information about certain groups. A list of groups or "all groups" can be specified (default: no group). The group information includes the group's access rights.
6. Right to edit information about certain groups. A list of groups or "all groups" can be specified (default: no group). The group information includes the group's access rights. The access rights that can be granted are limited to the ones of the user granting them.
7. Right to create new groups. Default: creation of new groups is disallowed.
8. Right to upload user and group data. This can be used for bulk generation of users and groups. Default: upload of user and group data is disallowed.
9. Right to see a list of the users currently logged in (default: disabled).

2.2.5.5 Heidelberg specific

The following access rights are additionally needed for the pilot application:

1. Right to order beta versions of products (default: disallowed).
2. Right to defer entry of license information (default: disallowed). This can be used to place orders with incomplete license information. When the corresponding key is first requested, the missing license information must be added.
3. Right to view end customers of certain users. A list of users and/or groups can be specified (default: only the user him or herself).

[9] This has changed since earlier versions of this document. See corresponding footnote for product related rights.

4. Right to edit end customers of certain users. A list of users and/or groups can be specified (default: only the user him or herself).

5. Right to order products for end customers of certain users. A list of users and/or groups can be specified (default: only the user him or herself).

6. Right to order products without specifying an end customer (default: disallowed).

7. Right to bypass entry of additional order information. Either a list of allowed products or ”all products” can be specified (default: no products).

2.2.6 Domain class model

Figures 2.21 and 2.22 show a first domain class model derived from the preceding requirements analysis. It is intended primarily to show the most important domain classes, their relationships between each other, and attributes and operations with specific relevance. Operations like Order.confirm, Order.cancel, ShoppingCart.addItem, etc. are left out in order not to overload the model. These operations and other details like data types, parameters, etc. are depicted in chapter 3 and appendix A.

2.3 IT specific requirements

2.3.1 Server platforms

On the server side, there are several ways to run an eCommerce application. However, only few are really suitable for open, scalable, and reliable systems. Platform dependent technologies[10] (like Microsoft's COM+) should not be taken into consideration. They limit both portability and scalability and, if there is only one vendor, lead to a dangerous ”lock-in” that makes an ”escape” to other technologies very difficult. With that in mind and looking for a widely supported and at least partly open software platform, only Java and CORBA based solutions are left for closer examination in the next sections.

2.3.1.1 Java Servlets and Java Server Pages

By now, most of the common web servers (including Apache and Microsoft Internet Information Server[11]) support Java Servlets. Servlets are java classes that run in a defined environment within a web server and can answer HTTP (and, if necessary, other) requests by dynamically generating corresponding HTML pages or other documents. More on servlets can be found in [Sund].

Java Server Pages (JSPs) consist of markup (e.g. HTML) and embedded Java code and are compiled into servlets by the web server. JSPs facilitate using page templates rather than generating pages purely by program code. More on JSPs can by found in [Sune].

2.3.1.2 CORBA

The Common Object Request Broker Architecture (CORBA, [OMG]) can serve as a *middleware* to build distributed systems. CORBA includes services for transaction, persistence, and others that make it suitable for eCommerce applications. Implementations exist for almost every platform, and products of different vendors are able to interoperate (at least on a defined common basis). However, it tends to be difficult to use and manage. One example is the Persistent Object Service (POS) that has been replaced recently by the Persistent State Service (PSS) because it was too complex and not widely implemented by ORB vendors. Also, component technology has just become part of the CORBA specification with version 3 and is thus hardly

[10] In the sense of hardware and operating systems.

[11] with 3rd party extensions

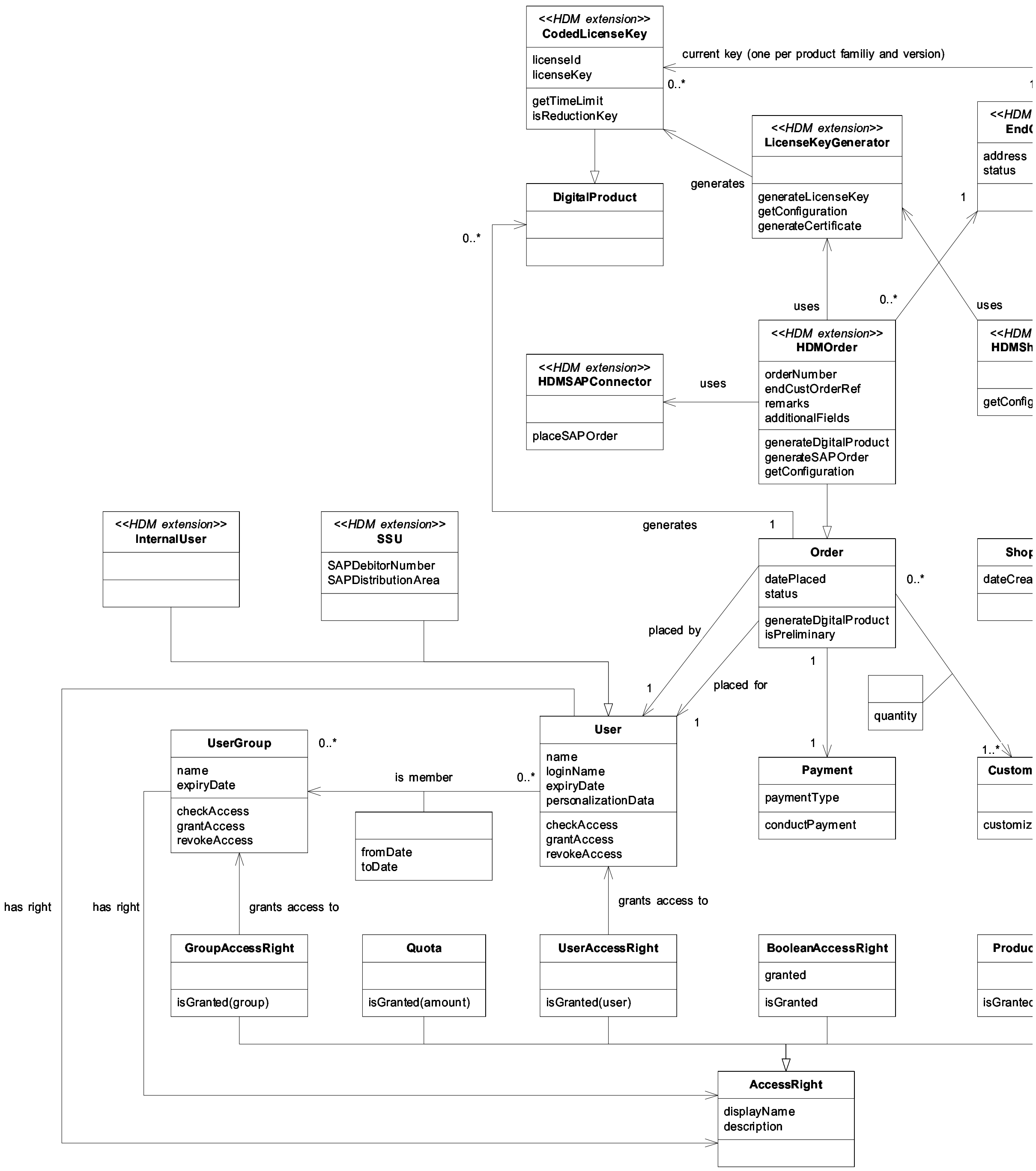


Figure 2.21: First domain class model (part 1)

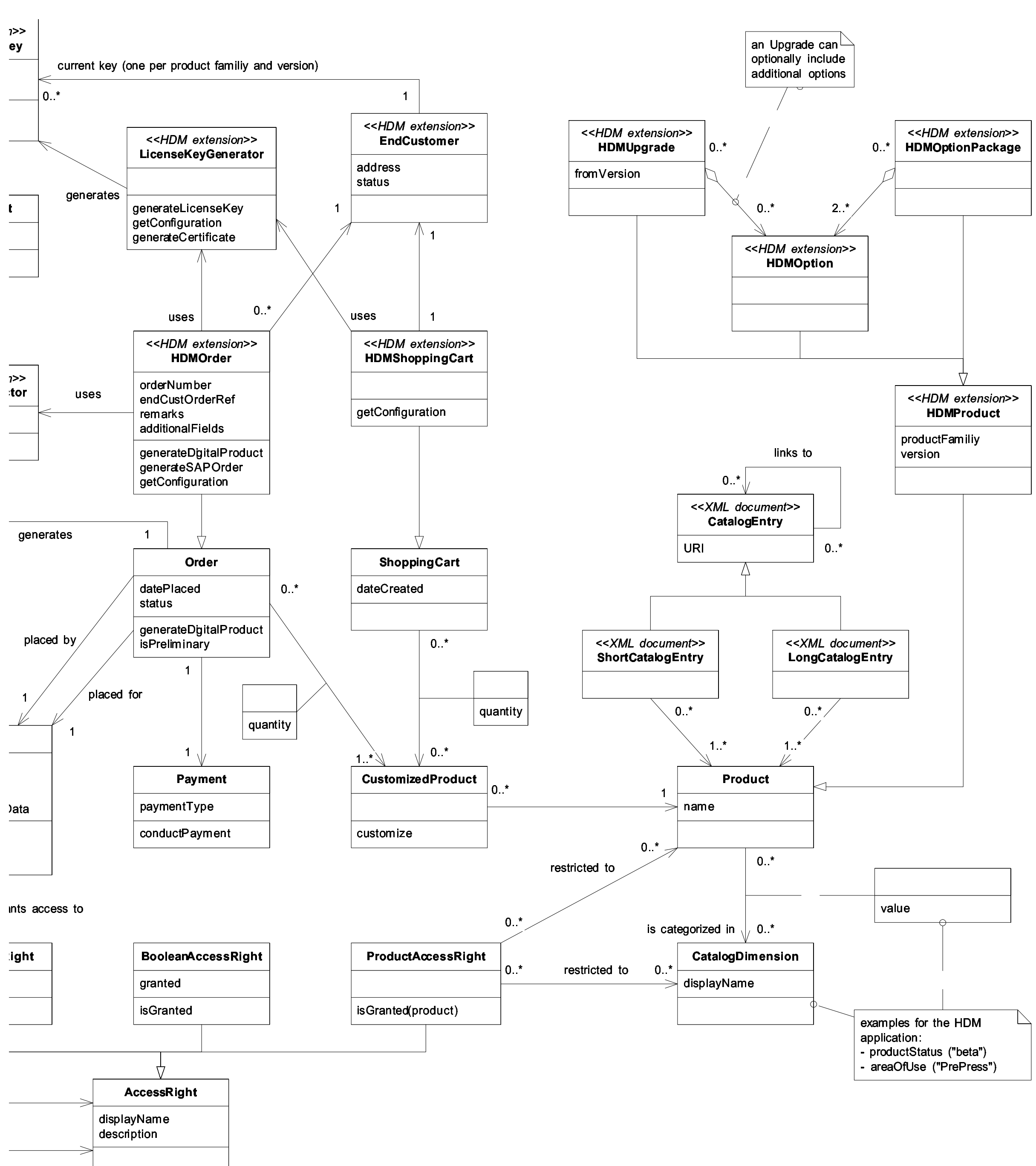


Figure 2.22: First domain class model (part 2)

implemented by any vendor yet. The component model parallels the Enterprise JavaBeans specification, with several additions that make it look a bit more like CORBA and harder to understand and use[12].

2.3.1.3 Enterprise JavaBeans

Enterprise JavaBeans (EJB) is a specification by Sun Microsystems ([Suna]) that describes a server side Java component model. The current version of the specification is 1.1. EJBs run in a defined environment called a *container* that offers services like transactions, persistence, naming, and others. Usually, an EJB container is part of an *Application Server* that may include a web server, a CORBA ORB, a transaction monitor, an interface to host or other ”legacy applications”, and more. Most EJB compliant application servers are CORBA based. Together with COM+, EJB is one of the most sophisticated distributed component technologies available. Several vendors offer implementations for a variety of platforms.

2.3.1.4 eCommerce frameworks

Instead of developing an entire eCommerce application on one's own, a complete framework can be used that provides the most commonly needed features. In this context, the term framework is used in two slightly different meanings:

1. An ”application template” that is (almost) fully functional and can be customized and extended to meet specific requirements. Example: BEA WebLogic Commerce Server (`http://www.bea.com/products/weblogic/commerce/`).

2. A more or less loose collection of components from a specific area, e.g. commerce or medicine. Example: IBM San Francisco (commerce framework with over 1000 components, `http://www.ibm.com/software/ad/sanfrancisco/`).

2.3.1.5 Conclusion

Servlets/JSPs are probably the best choice to handle incoming requests for dynamic web pages. Whether they also contain the application's logic or delegate requests to another layer depends on the application's complexity, number of requests, necessary scalability, and other factors. In general, eCommerce systems with a large number of requests or with the expectation to grow rapidly should be built using more advanced technologies like CORBA or EJB. These also offer services (like transactions, persistence, etc.) that would otherwise have to be built in manually, thus saving development time. CORBA could be a choice if its component model was easier to use and, most important, already implemented. These are the advantages of EJBs, together with their wide portability.

In this thesis, Servlets/JSPs are used as an input/output gateway to web browsers, while the core system is EJB based. Complete existing frameworks are not used here for several reasons:

- Many commercial Java frameworks are delivered without source code[13]. However, no creator of a framework can foresee all future problems or necessary extensions. Thus, it is essential to be able to modify the framework if necessary (though framework modifications should be made carefully and only with good reason).

- ISB doesn't want to depend on a vendor's license costs and conditions. By developing an own framework, the company remains independent.

- Most eCommerce frameworks don't provide satisfying XML support (e.g. BEA WebLogic Commerce Server). It would be possible to integrate XML capabilities into them, but this is not the same as supporting XML as the basic input/output interface of an eCommerce system (see 2.3.5).

[12]This is the personal opinion of the author.

[13]Especially Java frameworks don't need to be shipped with source code to work, while many C++ frameworks cannot be used without it.

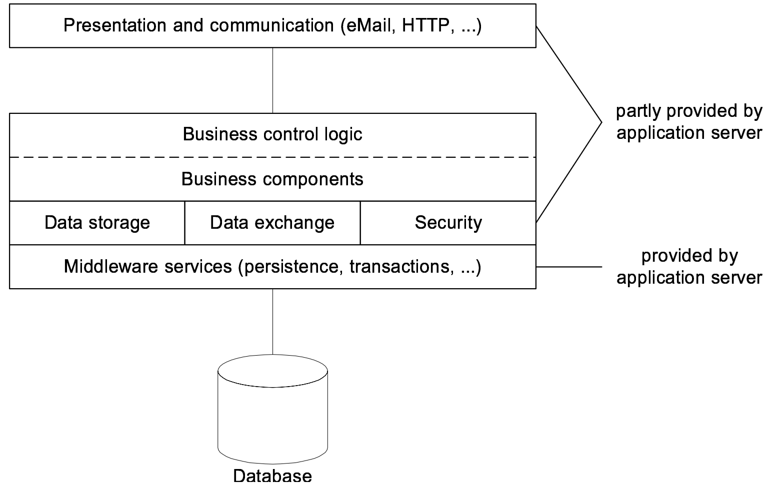


Figure 2.23: System layers

- A majority of existing frameworks is still based on ”conventional” rather than component technology. An example is IBM’s San Francisco that has a reputation of being comprehensive and sophisticated, but performing rather slow. IBM is still working on an EJB based version of the framework (http://www.ibm.com/software/ad/sanfrancisco/javabeans.html) which will at first only run on IBM’s own application server WebSphere.

2.3.2 System layers

eCommerce systems mostly follow the ”classical” three tiered architecture: presentation - application - database[14]. For the purpose of this thesis, the middle tier is further divided as shown in figure 2.23. Data storage, data exchange, and security facilities may be partly provided by the middleware services of the application server, but have most likely to be extended to be usable by the business logic. This is the same for presentation and communication components. The business layers are described in the following section, the others afterwards.

2.3.3 Business logic

The integration of customer specific business logic is one of the key points in building business frameworks. Out of the several possible approaches to this, two more interesting and commonly used are discussed in this section.

First of all, the business logic must be customizable on a programmatic basis. This means that the information flow across the business objects must be separated from them into *controller* components (see figure 2.23). The business logic can be customized by creating specialized controller and business components which can partly or completely adjust, extend, or override the framework’s default behaviour or, where there are no sensible defaults, implement interfaces that the framework provides. Intershop enfinity realizes a concept like this where the controller function is taken by Java classes called *pipelets*. Existing and new pipelets can be combined in various ways and exchange data with business objects and each other. Note that pipelets are Java classes running in the web server’s Servlet engine, while the controllers in this thesis should be based on Enterprise JavaBeans.

The second way is to provide a tool (usually graphical) that allows to manipulate the control flow without writing code. In Intershop enfinity, this is realized by graphically composing pipelets. A similar solution is

[14]Sometimes the web server is listed as a separate tier.

presented in chapter 3. However, a graphical interface will - mostly for time reasons - not be realized in this thesis.

2.3.4 Data storage

The only sensible way of data storage in an eCommerce system is using databases. Today, relational databases still have a much higher market share than object oriented ones. This is perfectly reasonable since objects are not really suitable for all purposes and make operations on large amounts of data harder. Thus, relational databases have to be supported by any eCommerce framework. There should also be a way to connect to existing databases rather than requiring an own database (like Intershop enfinity does). This means for example that it must be possible to load data from and save data to an SAP database. Another point is that it must not break the application if the switch to an object oriented database is made.

In general, the two most common approaches work as follows:

1. Generic load/store mechanism. This way, any components of the system that want to load or store something send a corresponding request to a storage access component. The problem with this approach is that it must work field by field to enable the use of relational databases. Also, relationships between objects cannot easily be mapped. Basically, the business components are responsible for translating object relationships to key or id relationships, respectively.

2. Code generation. This solution analyses the code of classes and, with help from the developer, generates code to load and store objects. Most tools allow the developer to link object attributes to fields in a relational database (called *object relational mapping*). While very flexible, the major disadvantage is that the code generation has to be repeated every time a classes fields are changed. This can be compensated by including the code generation into the build process.

The first method is used by the CORBA Persistent State Service. However, the most common approach is number 2 with several products on the market - e.g. JavaBlend ([Sunc]) and TopLink ([TOP]) - many of them with EJB support. It is also used by most EJB compliant application servers where it's part of the deployment process. To prevent a "vendor lock-in", only mapping tools that are not bound to a specific application server should be considered for use. One should not rely on special mapping capabilities that any EJB server/container offers to prevent porting problems when migrating to another product.

2.3.5 Data exchange

There are two possible ways an eCommerce system can be used: interactively or automated (i.e. batch mode). In the interactive case, the user enters data through HTML forms or similar means, while this is very inconvenient for automated use (although there is an approach for automated form processing called Electronic Commerce Markup Language, see [ECM]). On the other hand, it doesn't make much sense to work with two different interfaces on the application logic side. The best solution in the eyes of the author is to constuct one interface for both kinds of use based on XML. While automated requests (which take the form of XML documents conforming to a specific schema) are simply passed through from the presentation layer to the application logic, interactive requests are transformed to the XML format and then forwarded. This conversion is relatively simple and allows other means of input to be integrated easily. For example, a java application could be constructed as a front end with the application logic remaining unchanged. Another example are safety queries like "do you really want to cancel this order". They can be handled on the presentation layer without bothering the application logic. Furthermore, there is only a small interface to the application logic. Basically, only a single request handler component has to be exposed to the presentation layer which can simplify changes and extensions a lot and encapsulates the internal processing on the application layer. An XML based format also allows for easy integration of other XML exchange standards like BizTalk ([MS]) or XML-EDI ([EDI]). Ideally, these can converted to the "native" format by using a stylesheet with no further programming.

The most disturbing problem with the described approach is probably the extra time necessary for XML generation and parsing. However, experience shows that the RMI calls from servlets to EJBs form a bottleneck that renders some extra string operations irrelevant.

Technically, XML documents can be received by the system in several ways:

1. Direct passing to an XML handler method. This is what always happens at the end of the following points.
2. Calling a servlet from an HTML form that converts the parameters to an XML request.
3. Calling a servlet and passing a complete XML request via HTTP POST.
4. Other transport ways, e.g. eMail attachments or body texts.

XML can also be used to exchange data for administrative reasons, e.g. database records. Due to the automatic conformance check by the XML parser (does a document confirm to its schema?), an XML based format allows for easier validation of transmitted data than other alternatives. However, to remain compatible with existing applications, also a simple CSV format (Comma Separated Value) should be supported.

2.3.6 Security

2.3.6.1 Identification and authentication

Before a user can access the system, he or she has to identify or authenticate him or herself. The difference between identification and authentication is that authentication only checks if someone is allowed to do something. It's not of interest who this person actually is. Identification means truly identifying a specific person. For the purpose of this thesis, both terms are treated equally in the sense of authentication since true identification can only be achieved by biometrical or similar methods, which are way beyond the scope of this work. For example, if a piece of text reads ”the user identifies him or herself with his or her username” this is in fact an authentication since the user might have given his or her username and password away to another person.

Authentication should happen from both parties involved towards each other, client and server. All the authentication mechanisms mentioned below can be used for client authentication, while only digital signatures and certificates are a practical way of server authentication since they are supported by the major web browsers without additional software.

The following sections describe the most commonly used authentication methods and provide an assessment of their security and practical use.

Username and password A user name and a password must be supplied to login. If the connection to the system is not encrypted, this is a security risk. Unencrypted connections over the internet should be avoided **completely** when using this approach. By using SSL (see 2.3.10), there is little risk that authentication information gets spied while travelling to the server. However, there is still the possibility to get the password by logging keystrokes, looking over someone's shoulder, or simply guessing.

One time passwords The risks mentioned in the preceding paragraph can be reduced by using one time passwords. A password generator (sometimes called *authenticator*), which is a piece of hardware and/or software accessible to the user, makes up a new password for every login and/or transaction. Verification is done using the same algorithm that generates the passwords. With this method there is no danger in the password getting spied since it is useless afterwards. However, the password generator must use a secure cryptographic algorithm to produce the passwords. Otherwise, new passwords may be predicted from the old ones. Also, the risk is only shifted to the generator which itself must be secured against unauthorized access. One approach is to use hardware generators (looking like calculators) which require a PIN to generate a one time password. Even if they are stolen, they cannot be used without the PIN (like an ec-card).

PINs and TANs This concept is based on two stages of authentication. A *Personal Identification Number (PIN)* serves as a password. If the PIN is spied, unauthorized persons are able to login. However, every sensitve action requires entering a *Transaction Number (TAN)*. These numbers are either pre-generated and sent to the user via postal mail or a secure electronic channel, or they are produced by a generator accessible to the user (see preceding paragraph). Like one time passwords, every TAN can be used only once.

Digital signatures and certificates Digital signatures are based on public key cryptography, where there is a pair of cryptographic keys. One key is used to cipher data while the other is needed to decipher it again. For this reason, public key methods are called *assymetrical*. One of the two keys is the *public key*. This key can be given away without concern while the *private key* must be kept secret. If the keys are long enough (depending on the algorithm used, e.g. 1024 bit RSA keys), there is no practical way known today to decipher a message without the necessary key.

To digitally sign a message (e.g. a login request), the user encrypts it with his or her private key. The system decrypts the message with the user's public key. If the result is the expected text or other data, the user is authenticated. To prevent a repeated login with a spied message, some unique data, e.g. the current date and time, should be encrypted on every login. Alternatively, the server can send a *challange* (usually a random number) to the client encrpyted with the client's public key. The client uses the private key to decrypt it and reencrypts it with the server's public key. By receiving the original challange upon decrpytion, the server can be sure about the client's identity. There are other variants, but the two mentioned above can be considered the most common. Note that the security of this approach is dependent on the secrecy of the private keys!

The problem with digital signatures is the attribution of a public key to a certain individual. To facilitate this, *certificates* can be used. In this scenario, a *Certifying Authority (CA)* digitally signs an electronic document stating ”Public key ... belongs to Mr xy”. The CA's public key must be publicly available to verify the certificate. Note that certificates are unnecessary if a user's key pair is generated by the organization that itself verifies documents signed by the user (e.g. a company generating login keys for their employees).

Secure socket layer (SSL) client certificates are one implementation of this concept. They can be used with Microsoft Internet Explorer (3.x and up) and Netscape Navigator (3.x and up). SSL also provides authentication of the server, so clients can be sure they are not sending orders to same fake system. More information on SSL and TLS (Transport Level Security), its presumable successor, can be found in [Net] and [IET].

Assessment of the presented authentication methods The first method mentioned (Username and password) should not be used since it has proven very vulnerable. There are a lot of possible attacks ranging from guessing a user's password as the name of his or her dog to placing trojan horse programs on the user's computer. By using secured transmission channels (e.g. SSL) to communicate the password, security can be increased. However, trojan horse attacks are still possible.

One time passwords in conjunction with hardware authenticators provide acceptable security in most cases, provided the authenticators themselves are sufficiently secure. They can be used on other people's computers as well as transmitted over unsecure communication channels. Note that this approach renders password spying useless, except that the message containing the password can be intercepted so it never reaches its target. This way, the stolen password can be used for a single unauthorized login.

PINs and TANs offer security similar to one time passwords. However, if the TANs cannot be generated by the user him or herself, the administrative overhead increases due to the need of issuing lists of TANs regularly. Giving the user the opportunity to generate TANs in a software based way is not recommended. This would allow for trojan horse attacks or, if TANs are transmitted over unsecure communation lines, man in the middle attacks. Unencrypted communication is not recommended to prevent interception of messages including a TAN and then using the TAN for an unauthorized transaction.

The most sophisticated authentication method is using digital signatures. There is no way known today to forge digitally signed messages, provided a proper algorithm is used for signing. The drawback of this method is that one cannot simply use an arbitrary computer with an internet connection to login since signing messages requires additional software and/or hardware that is usually not present. This is the same for PDAs,

WebPads, and the like. Also, a user's private key must be stored safely. Smart cards or other electronic devices with an own microprocessor can be used to both store the private key and sign messages, thus making trojan horse attacks impossible.

Summary The most flexible and still secure ways of authentication seem to be one time passwords and PINs/TANs, respectively. They should be used together with encryption (to prevent ”hijacking” of sessions) and hardware authenticators or centrally distributed lists (to defeat trojan horses). However, if the ability to login from arbitrary computers or other communication devices is not important in a particular context, e.g. when conducting automated business transactions, digital signatures are the better choice. They are more comfortable to use while offering state of the art authentication mechanisms. Digital signatures and certificates are already used in a broad range of applications and will presumably increase in popularity very rapidly.

The framework developed in this thesis should support a variety of authentication methods with the possibility to add more in the future. Server authentication should be provided by support for the SSL protocol.

2.3.6.2 Encryption

Encryption of messages exchanged between the eCommerce server and its clients should be based on modern cryptographic principles, more precisely public key cryptography (see 2.3.10). SSL with 128-bit session keys seems very well suited to this purpose. SSL can be used for both authentication and encryption and incorporates several cryptographic algorithms, including IDEA and Triple-DES (see 2.3.10). Another advantage is that SSL is supported by both the major web browsers and servers. There is no extra development work needed for supporting SSL. Ceritificates for SSL communication can also be used in S/MIME (Secure/Multipurpose Internet Mail Extensions) applications to provide another way to securely access the server. If necessary, SSL support can be integrated directly into the eCommerce framework by either using open source (OpenSSL, [Ope]) or commercial software. Note that the newer *Transport layer security protocol (TLS protocol)* is based on SSL 3.0. For this reason, I just use the term SSL here. This includes also *Wireless TLS (WTLS)*, the security protocol accompanying WAP (the wireless application protocol).

Given the popularity of SSL/TLS, its quality ([RSA]), flexibility, and ease of use, other (proprietary) encryption algorithms and products don't have to be examined in detail.

2.3.7 Presentation

For an eCommerce system today, HTML is a must as a presentation means since most users will interact with such systems through a web browser. It is important that the user interface can be designed seperately from the application logic to enable a separate company or department to do the web design. However, HTML (together with related technologies like JavaScript and CSS) is not the only WWW language anymore. XML is the emerging new standard for the WWW as well as for many data exchange processes. XML is a metalanguage that can be used to describe other languages like HTML or WML. Together with XSL(T) it is a powerful tool for addressing present and future internet appliances. There is almost no other choice than XML as the basis of a modern eCommerce solution's user interface.

The major problem with XML today is that its not fully supported by today's web browsers (and certainly not by other devices like mobile telephones). To display an XML page in a sensible way, layout information must be provided by the server and interpreted by the browser. However, the XSL formatting objects intended for this are still not finally specified. One can use CSS instead, but this is also problematic when using one of the major browsers, MS Internet Explorer or Netscape Navigator, which both support only subsets of CSS.

The most common solution is to transform XML documents to other XML-based languages (like XHTML, WML, etc.) on the server. To achieve this, XSLT can be used, but this has another drawback. Common XSLT engines like XalanJ ([Apa]) might be too slow to use them for online transformations (opposed to offline transformations from one set of static pages to another). This is discussed in more detail in chapters 3 and 4. Furthermore, XSLT in its current version lacks some features necessary for real-world applications that are implemented by XSLT engines in proprietary ways. As a consequence of both issues, other means of

transformation like Java Servlets or Java Server Pages should be considered. They can also be optimized to the specific XML-based language that is used as the transformation source.

2.3.8 Specific requirements for the pilot application

2.3.8.1 Connection to the ERP system

The only payment method in the pilot application is invoicing. To successfully place an SAP order for which SAP generates an invoice, the following data is needed (according to HDM order management):

1. Identification number of the products that should be ordered.
2. Debitor number of the ordering SSU. This number has to be specified when creating a new user of type SSU.
3. Distribution area number. Every SSU belongs to a specific distribution area. This number has to be specified when creating a new user of type SSU.
4. Various calender dates. These can all be set to the current date.
5. Type of order. Probably a new type (”online order”) will be introduced for that.

The connection between an SAP order and an order in the pilot application is made by a unique order number (from the view of the pilot application) that is transmitted to the SAP system together with the other data necessary to place an order.

The exact technical data exchange with SAP is depicted in detail in chapters 3 and 4.

2.3.8.2 The license key generator

The key generator functionality lies in a library that is yet available for 32 bit Windows and Macintosh. To make it callable from an application server environment, two approaches can be used. First, the library could be ported to java. Since it is also used in all programs that make use of the unified licensing process, this would mean to manage two versions in parallel. Second, a small application could be built around the library that makes it accessible over a network. This introduces the new problem of securing the network access to the generator (see 2.3.6). However, this seems to be the better approach, since it requires less maintenance effort and allows the key generator to run on a different platform as the eCommerce system. Note that using the *Java Native Interface (JNI)* directly in an EJB component is not a choice since this would introduce massive porting problems.

There must also be a component that generates license certificates that include the key. It must be possible to send these certificates via eMail in PDF and to fax them to a given number. This functionality could be combined with the key generator in one application or realized independently. To create PDF documents, either commercial software like Adobe Acrobat or free software like pdftex could be used.

To access the key and certificate generator functionality, the exchange of XML messages via TCP/IP seems suitable.

2.3.8.3 Deployment

Figure 2.24 shows the deployment of the pilot application with a maximum functional distribution. All functionality can as well be concentrated on a single machine. On the user tier, WAP and HTML are only two alternatives (probably the most interesting ones) that are possible by using XML (see 2.3.7). The core system and presentation servers can be clustered individually as needed without great efforts since many server products in this area provide transparent load balancing and failover capabilities. WWW and eMail servers can also be ”pseudo-clustered” by modifying DNS entries periodically.

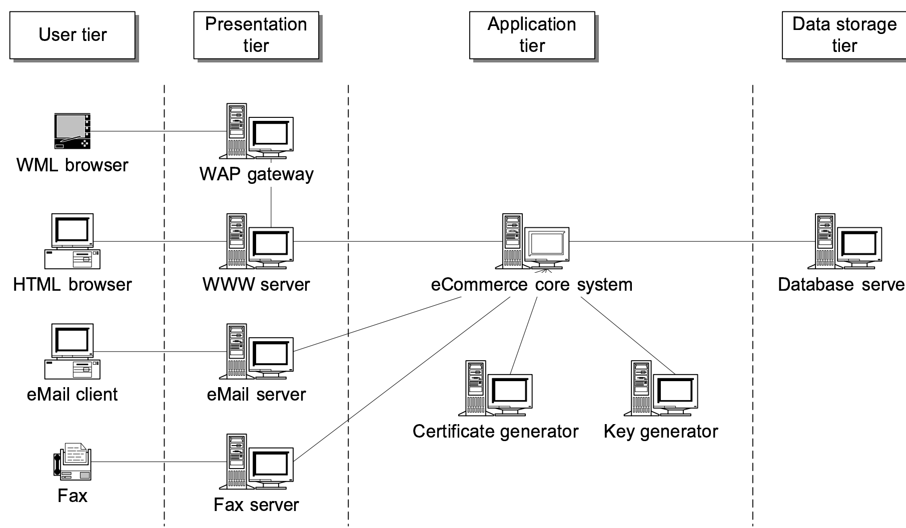


Figure 2.24: Physical distribution of the pilot application

2.3.9 Examination of Intershop enfinity

Intershop claim their latest product *enfinity* is suited specifially for B2B applications. However, studying the technical documentation gives a bit of a different impression.

There is only poor support for automation. While the product does have a rudimentary XML interface for orders and catalog queries, this feature is not usable in a sensible way. There is no authentication and only a minimum validation of incoming XML requests. Thus the interface may only be appropriate for internal use in a company given its current state. It will probably be extended in the future - though this is not much help right now.

Regarding security, enfinity offers encryption via SSL. This seems appropriate for business transactions as explained in 2.3.6.2.

Technically, enfinity is based on the *Oracle 8i* database, the *PowerTier* application server from Persistence Software (with support for EJB 1.0) and *VisiBroker for Java* from Inprise. PowerTier uses the *Oracle Call Interface (OCI)* to increase database access performance. The enfinity architecture employs EJBs simply as a bridge to the database. The problem is that enfinity uses EntityBeans with container-managed persistence, which are only supported in a proprietary way by the PowerTier application server. In fact, no standard complient EntityBeans with container-managed persistence can be deployed in PowerTier at all! If one wants to add support for custom persistent data, exactly the proprietary PowerTier approach must be used. A last point concerning enfinity and EJB is that the business logic runs on the web server and therefore doesn't make use of the EJB standard's efficient resource management concepts (e.g. swapping out "lazy" objects to secondary storage - an important point given the extensive resource consumption of Java).

As a summary, Intershop claims to support open standards and interfaces with enfinity. Taking a closer look, however, this support proves half hearted to rudimentary. Especially XML is not really supported, and only "PowerTier compliant" EJBs can be added to the system. B2B requirements not fulfilled by the base product are addressed by additional enfinity components (called *cartridges*). Since enfinity licenses are very expensive even without these cartridges, and given the product's architectural weaknesses, it is no longer taken into account for this thesis.

2.3.10 Glossary of terms

Application server A software product providing the necessary infrastructure for server-side distributed applications. Most application servers are based on open standards like EJB and CORBA.

CA Certifying authority. An institution that issues and signs digital certificates.

Component A software module with one or more defined interfaces. Usually, components support black-box reuse (i.e. reuse without knowing the source code) and are more coarse grained than classes. In most component architectures, components can be examined (*introspected*) and configured at design time and run time. Design time configuration is often assisted by graphical tools.

Container A run time environment for components. The container provides access to services like persistence, transactions, naming, etc. via a standard interface. It's other main purpose is to manage the life cycle of contained component instances.

CORBA Common Request Broker Architecture. CORBA is a standard defined by the *Object Management Group (OMG)* to support distributed, object oriented applications. With the latest version of the standard (CORBA 3), a component model similar to Enterprise JavaBeans was introduced.

CSS Cascading style sheets. A W3C specification for layouting web pages. The current specification is called CSS level 2, but its predecessor CSS level 1 is still not fully implemented by the major browsers.

DES Data encryption standard. A symmetric cryptographic method developed by the US secret service NSA (National Security Agency).

Digital certificate A digital document used to bind a public key to a certain person, company, or institution. Digital certificates are issued and digitally signed by *certifying authorities (CAs).*

Digital signature A digital document that proves that a certain other document has been written by or at least been approved by the owner of a certain private key. Digital signatures are based on public key cryptography and are often used together with digital certificates to bind the signature to a readable name.

DTD Document Type Definition. In XML, a DTD is used to describe the logical structure of a class of documents. The W3C is currently working on a specification called *XML Schema* with the intention to replace DTDs in cases where their possibilities are too limited.

EJB Enterprise JavaBeans. EJB is a specification by Sun Microsystems defining a server-side Java component model and run time environment.

HTML Hypertext Markup Language. With the first version developed by Tim Berners-Lee at CERN in 1992, HTML is still the standard markup language that web pages are written in. The current version of the standard is 4.01.

HTTP Hypertext Transfer Protocol. The data exchange protocol that is used to access the world wide web.

IDEA A symmetric encryption algorithm using 128 bit keys.

Internet appliance A device for accessing the internet. Usually, IAs are small and portable, like organizers or mobile telephones.

Java Servlets A specification by Sun Microsystems for server side Java classes that handle HTTP or similar requests.

JavaScript A script language developed by Netscape that can be embedded in HTML to improve the interactivity of web pages. Two major drawbacks are its incompatible implementation in different browsers and some security holes.

JSP Java Server Pages. JSP is a standard by Sun Microsystems for mixing HTML or other markup with Java programming language instructions. The Java code is processed before the page is sent to the client, thus allowing easy integration of dynamic content.

Middleware An infrastructure for distributed, object oriented systems. Middleware products usually offer data access, transaction, naming, distribution, life cycle management, and security services. Examples are Sun Microsystems' Enterprise JavaBeans, CORBA, and Microsoft's COM+.

ORB Object Request Broker. In the CORBA standard, ORBs are mainly responsible for passing method invocations and events between objects. They also provide access to the various CORBA services.

PIN Personal Identification Number. This number serves as a password in PIN/TAN based security.

POS Persistent Object Service. Part of the CORBA specification for persisting objects. POS was hardly ever implemented and is superseded by PSS.

PSS Persistent State Service. A CORBA service that allows CORBA objects to persist their state to a database or other form of secondary storage.

Public key cryptography An asymmetric cryptographic approach. PK systems use two keys for each user, the public and the private key. Both can be used to encrypt messages, but decryption requires the other key. For example, a message encrpyted with a persons public key can only be decrypted with the corresponding private key (and vice versa).

RSA Rivest, Shamir, Adleman. RSA is a public key cryptography algorithm and stands for the first letters of the last names of its inventors. It is based on the factoring of large numbers with two prime factors. RSA is considered secure when using keys with 1024 bits or more in size. Details about RSA can be found in [RSA].

Servlets See *Java Servlets*.

Session key A symmetric cryptographic key that is used only temporarily. Session keys are exchanged using public key cryptography, but the actual encryption of messages is left over to a symmetric algorithm. This saves a lot of processing time, since public key ciphers are, in general, quite slow compared to symmetric ones. A session key is only valid for a single communication connection ("session").

SSL Secure Socket Layer. SSL is an authentication and encryption protocol developed by Netscape. It can be placed on top of a TCP/IP stack to provide secure and authenticated communication for higher level protocols like HTTP. SSL is based on public key cryptography and digital certificates. SSL will presumably be superseded in the near future by *TLS*.

Symmetric cryptography This term refers to cryptographic methods using the same key for encryption and decryption. Most symmetric encryption algorithms are considered secure when used with 128 bit keys (e.g. IDEA).

TAN Transaction Number. TANs are used to authorize single transactions in a PIN/TAN based system. Every TAN can be used only once.

TLS Transport Layer Security. Described in RFC 2246, TLS is based on and a possible successor to the SSL protocol.

Triple-DES A security improved variant of DES.

W3C World Wide Web Consortium. The organisation responsible for WWW standards like HTML and XML.

WAP Wireless application protocol. The counterpart of HTTP for mobile internet applications.

WML Wireless markup language. The page description language for mobile internet applications that are based on the WAP.

WWW World Wide Web (in short: web or w3). The sum of all HTML pages and other resources accessible through the internet via HTTP.

XHTML A reformulation of HTML 4.01 in XML. XHTML is intended as an intermediate standard on the way to an XML based web.

XML eXtensible Markup Language. XML defines a markup syntax and a meta-language used to describe arbitrary markup (i.e. document structuring) languages. Currently, XML uses so called *Document Type Definitions (DTDs)* to define languages that can be used to form documents. XML is a subset of the *Standard generalized markup language (SGML)*.

XML Schema An upcoming W3C recommendation for describing classes of XML documents. XML Schema is intended to replace DTDs.

XSL eXtensible Stylesheet Language. XSL is a coming W3C standard (currently with the status of a working draft) for formatting XML documents. It consists of two parts: XSLT to transform documents and XSL Formatting Objects (FO) to specify the layout and format for displaying the transformed documents.

XSLT XSL Transformations. An XML document transformation language that is part of the XSL standard.

Chapter 3

Design of an eCommerce framework

3.1 Goals

The framework developed in this thesis should be settled on a high level of abstraction and be easily adaptable to different business processes and IT infrastructures. This requires the designed components to be very generic, but at the same time more specific ones suitable for the most common situations should be provided.

From a more general point of view, the framework must realize a *product lookup and request* concept. Whether these products are physical or maybe dynamically generated for downloading or whether or not the user is charged are secondary aspects that must be taken into account but should not drive the design.

Another important aspect that should be considered right from the beginning is security. The user's access rights must be checked before any code is executed in order to process a request. These checks should be made on the basis of requests, not individual methods. However, the role based security concept of EJB can still be employed to distinguish methods that should only be callable from within the system from others that are available from outside.

The framework uses the Enterprise JavaBeans component model and should run in any EJB 1.1 compliant container. No specific data stores should be required. It must however be assumed that relational databases will be used in most applications built with the framework.

3.2 Tools used

A great deal of the design presented in the following sections involves code generation, ”plugable” components with no fixed relationship, and control flows based on the contents of XML documents. All these concepts are hard to express in CASE tools, so these would just be used to draw standard diagrams. Since Visio is better at that, it is used once again here.

3.3 Overview

3.3.1 Request/response pattern and schemas

This design is based on a request/response paradigm. The general control and information flow is as follows:

1. *Presentation* components react upon a user action and generate a request (section 3.9). In the non-interactive case, the request is generated by other means (e.g. programs written for that purpose).

2. The framework receives a request by means of *data exchange* components (section 3.5). The request is authenticated and it's privacy and integrity guaranteed by *security* components (section 3.7).

3. The request is handed to the *business process* layer that encapsulates the business logic of the system (section 3.4).

4. The *domain components* process the request and generate a response (section 3.8).

5. *Data storage* components are used during request processing to retrieve and store persistent data (section 3.6).

6. The response is returned and presented to the issuer of the request.

The requests and responses are XML documents. Their structure is specified in an *XML Schema* ([W3Ca]). Despite XML schemas are not yet a recommendation by the World Wide Web Consortium, they are already so powerful as a DTD replacement that they are used in this design. Schemas make it possible to have a *single source* describing the system's interface and data structures. Section 3.5 shows how schemas are used to validate incoming requests; section 3.6 explains how schemas are used to automatically generate EntityBeans and SQL scripts from the schema as well as an XML interface to access the beans.

Note that the interface of the framework is described in this chapter using example XML requests. Schemas are not very readable, so examples with some comments seem to deliver the better picture. Appendix A contains the schema fragment defining the basic `<request>` and `<response>` types to give an idea how formal request definitions in an XML schema look like.

The terms introduced in this chapter are summarized and shortly explained in section 3.10.

3.3.2 Framework approach

The basic pattern for constructing frameworks is the separation of *interface* and *implementation*. The framework interacts with most of its parts via their interfaces. Users of the frameworks can "plug in" their own classes/components by implementing these interfaces and telling the framework to use the customized versions. The latter can be achieved by *factories* (see [GHJV95]) or similar mechanisms. Frameworks often provide *default implementations* (where appropriate) of most of their interfaces. Own implementations can either be constructed from scratch or by derivation from the defaults.

The framework presented here is based on Enterprise JavaBeans (EJB, [Suna]). It interacts with the beans through their home and remote interfaces and provides default implementations where reasonable. It can be extended through own implementations of the interfaces, either from scratch or by inheritance from the defaults. Components can also be connected using process graphs (see 3.4) to change or extend the framework's behaviour.

3.3.3 Clustering

A *cluster* is the context of this section is a collection of network nodes that all run the same application. There are two main reasons for clustering: Load balancing and failover. Both is often achieved on the TCP level by servicing each connection from an arbitrary node. If one node fails, its connections can be taken over by another. The framework must be prepared to run in such a cluster, and this section explains how this requirement affects its design.

When looking at the EJB architecture there are two options for sessions beans (which contain only application logic with no persistent data). Session beans can be stateful or stateless. Stateful means that one bean instance is associated with a specific client and holds so called "conversational state" with that client, while a stateless bean can be called from any client at any time. As an example, a shopping cart could be realized as a stateful session bean that holds the cart's contents in memory until it is ordered or a timeout occurs. Assuming servlets are used on the web server tier, each servlet session would hold a reference to the shopping cart for that session. When clusters are involved, the web server might run on one node or cluster of nodes and the application server on another. When load balancing is active, the node that originally created a shopping cart instance might not be the one that receives another request within the same session. There are application servers that solve this problem by offering clusterable servlet engines. However, these servers are more expensive and currently don't offer clustering of stateful session beans. That means that the stateful shopping cart scenario would work in such a cluster, but all calls to the cart for a session would automatically

be directed to the same node. Load balancing can still be done on the session level, but there is no transparent failover since the shopping carts maintained on a failing node are lost.

The other alternative is using stateless session beans. Each client call arrives at an arbitrary instance, so no conversational state can be maintained within the instances. Instead, the concept of a session must be realized by making the conversational state persistent, preferably via entity beans. The advantage is that load balancing can occur more fine grained (on the method instead of the session level) and transparent failover is possible - provided the application server is capable of that.

Taking into account the above mentioned factors, making all beans that are reachable from outside the application server stateless seems to be the best choice. As a consequence, all other beans should also be stateless - it doesn't make sense to call a stateful session bean from a stateless session bean, since the "session" would be restricted to a single call. As an alternative, a handle to a stateful session bean could be persisted and reused within the same session. Since this introduces the same problems with clustering again that should be avoided by using stateless beans, this is not an option.

In summary, no stateful session beans are used at all in the framework. While stateless beans are a little more complex to handle, they are prepared for clustering - and there is nothing that could only be realized through stateful beans.

3.3.4 Transactions

All request processing is guarded by transactions which is achieved by using appropriate transaction attributes in the deployment descriptors of the EJBs. There is, however, no built-in support for long term transactions (i.e. transactions that span multiple requests). Such client controlled transactions can lead to severe resource consumption and locking problems like deadlocks or long waiting times. They can, of course, be supported by applications based on the framework - this is fairly easy using the EJB transaction concept. However, long-term transactions should only be used if *really* necessary and with great caution.

3.4 Business process encapsulation

3.4.1 Overview

Every business related application or framework has to model business processes in some way. The easiest possibility - from the developer's point of view - is to simply transform the processes into program code. The disadvantage is that a process is not easily understandable from the code and that every change to a process requires code changes and recompilation. On the other hand, representing a process in a purely abstract form might introduce limitations for the application that lead to complicated solutions for some problems. This is often the case when new technologies become available that could not be taken into account when designing the abstraction. A perfect example for this is eCommerce.

An example for modelling business processes is Intershop enfinity's concept of *pipelets*. These are basically Java classes, and a visual tool can be used to build a processing "pipeline" out of them. To change or add a business process, Intershop recommends to first try composing existing pipelets. As soon as this is not enough, one can write own pipelets to add necessary functionality. Other products use similar combinations of graphical and programmatic process representations.

The next sections explain the approach taken in this thesis. Figure 3.1 shows a class diagram that provides an additional overview. Note that some of the classes shown are not explained here but in section 3.5.

3.4.2 Contexts

Before explaining the framework's method of modelling business logic, the concept of *contexts* should be introduced. A context in the terminology of the framework is basically an information store with a certain scope. The base class for contexts is called `Context` and provides the same interface to key/value pairs as a

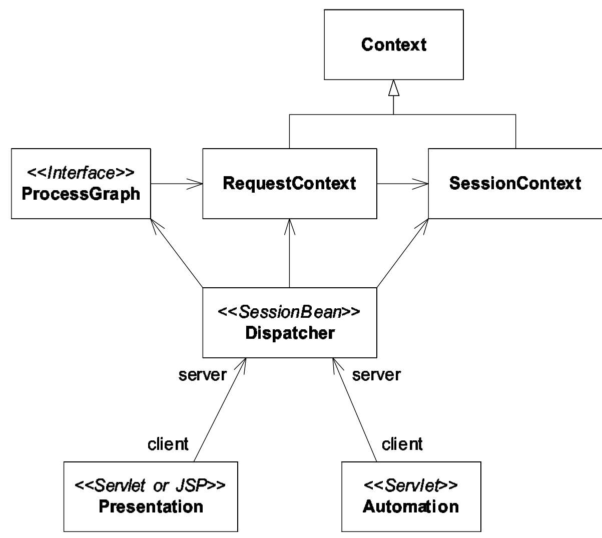


Figure 3.1: Class diagram for sections 3.4 and 3.5

`java.util.Hashtable` in the JDK 1.1 version. It doesn't provide any additional services. The key/value pairs are called *variables* here.

A *request context* holds *request variables* that are meaningful while processing a single client request. Besides that, it provides access to the current *session context*. A session is a logical group of requests that are - or at least can be - dependant on each other (see also section 3.5). A session context contains *session variables* and has the following additional method:

```
public void invalidate();
```

With the `invalidate()` method the context can be marked for disposal and must no longer be used.

3.4.3 Process graphs

Within the framework, so called *process graphs* are used to describe business processes. A process graph resembles the processing pipeline in Intershop enfinity but uses EJBs and Java Beans instead of pipelets. A graphical process manipulation tool is beyond the scope of this thesis. However, process graphs are XML documents with a simple structure, so it shouldn't be a problem to build such a tool sometime in the future.

Each process graph is associated with one type of XML request as described in section 3.3.1. For example, a request with the root element `put-into-cart-request` triggers traversal of a process graph that puts items into the shopping cart. Besides process graphs for individual requests, there are also three additional graphs that are executed at the beginning/end of a request and the beginning of a session (see section 3.5). This allows for easy pre and post processing.

As mentioned above, a process graph is an XML document. This document is converted to Java code (similar to a Java Server Page) instead of parsing it at runtime. The conversion is carried out using an XSLT stylesheet. Figure 3.2 shows a sample process graph, figure 3.3 on page 52 contains the Java code generated from it. Process graphs can be added, replaced, or removed at runtime, thus modifying the behaviour of the application without the need to stop and restart the application server. The current set of process graphs and the requests with which they are associated is saved in the database.

The DTD for process graphs can be seen in appendix B. The meaning of the elements is as follows:

component A component identifies either an EJB or a Java Bean that is used in a process graph. This element is mainly there to provide a short alias for calling a component.

component-call The component call element stands for a call to either an EJB or a Java Bean. The method to call must only take the current request context as a parameter and may only throw an exception of type `ProcessingException`[1]. The return type of the method must be void.

if This element allows for conditional processing. It consists of a condition (see below) and two subgraphs. One subgraph is traversed if the condition evaluates to true, the other if it evaluates to false.

loop A loop can be used to repeatedly traverse a subgraph. It consists of a condition (see below) and a subgraph. A loop element is equal to a `while` loop in Java and C(++). As long as the condition evaluates to true, the subgraph is traversed another time.

condition This element contains an expression with a boolean result. `get-var` and `is-var-set` elements can be used within a condition (or can be the sole content of it).

process-call As part of a process, a "call" to another process graph can be made. This graph is processed with the same request context as the calling graph. Note that the pre-request graph is *not* called again. When the called graph is also usable standalone, it is probably necessary to replace the current request with another one (indicating what the callee should actually do) before making such a call.

set-var This elements sets either a request or a session variable. The name and type of the variable must be specified together with the value. Note that the type is not directly a Java type (see below for treatment of the different types).

get-var This element returns the value of a request or session variable. The variable's name and type must be specified (see below for the treatment of the different types).

is-var-set Returns a boolean value indicating if the specified request or session variable exists (i.e. has been set before and not cleared after that).

clear-var Removes a variable from the current request or session context. The type of the variable does not have to be specified (note that there are no separate namespaces for the different types).

copy-var Makes a copy of one variable under another name.

processing-exception Makes the process graph throw an exception with a specific id (describing the cause) and optional parameters giving more precise information.

#PCDATA At several places within a process graph, arbitrary text can be inserted. This text consists of Java code that is executed as soon as it is reached during traversal.

Variables can be of types int, string, bool, and float. `set-var` generates instances of Java `Long`, `String`, `Boolean`, and `Double` for these types. `get-var` returns Java `long`, `String`, `boolean`, and `double` types. Note that `get-var` with type "string" always produces valid results, since it calls the `toString()` method instead of trying a cast to the `String` class. For the other types, an exception is thrown indicating an internal process graph error if a variable cannot be cast to the specified type. If the variable cannot be found, the Java default for the specified type is returned (0, false, or an empty string). Note that these rules only apply to elements of the process graph. Components or embedded Java code can get and set variables of arbitrary Java types as long as they are serializable.

Figure 3.2 is an example of a process graph. It calculates the price for a product based on whether the customer is a dealer or not. If this is the case, a discount is granted. If not, value added tax is added. This

[1] A `ProcessingException` can wrap another exception the same way a `java.rmi.RemoteException` can.

process graph can be called from other graphs for price calculations. As a prerequisite, a variable called "Product.price" must be set before the call. As a result, the calculated price is placed in the same variable. This is not a real world example, since the resulting price is not rounded, information about the customer is acquired for every calculation, etc. However, it points out the basic idea. Note that the multiplicative operators ("*") in the example are actually tiny pieces of Java code. The process graph could in fact contain all logic by itself without calls to any components. Naturally, this would turn it into a bunch of Java code surrounded by some XML. Like with Java Server Pages, one has to carefully decide how to split the functionality between process graphs and components.

Process graphs like the one in figure 3.2 are easy to understand and modify - even for people who are not specifically familiar with Java. Using a good XML editor and some samples, I believe they can be used even by non-IT people - which in the end is their main purpose. A graphical tool would of course be a valuable supplement. Visual graph representations would also be a good documentation of the behaviour of an application.

3.4.4 Component design

Components that are used in process graphs should be designed in a rather flexible way. As an example, let's look at the shopping cart controller described in section 3.8.4. This component has a method called `parseRequest()` or similar that is called from the process graph for a request and takes the appropriate actions. Now let's assume we want to add a check that ensures a user only puts products into the cart that he or she has been granted the right to purchase. If the parse method was doing everything on its own (i.e. without calling other overridable methods), we would have to provide a complete implementation of our own, including parsing. The reason is that there would be no possibility to interpose method execution and do an access check after the request was parsed but *before* the requested action is taken out.

To enable more fine grained customizing, the cart controller provides additional overridable methods that are called after parsing is complete. One of them is an `addItem()` method that perfectly suits the needs of the above example. In a derived class, the necessary access checks can be placed here and a processing exception thrown if access must be denied. Otherwise, the base class method can be called. This kind of design matches the *template method* pattern described in [GHJV95].

3.4.5 Limitations

A process graph not really describes a whole business process but only a part of it. For example, ordering products is a business process that spans several requests: looking up products in the catalog, placing them into the shopping cart, editing the shopping cart, and finally placing an order. However, a process graph only describes a single request. For a complete process, several process graphs are involved that have to exchange data via the session context.

The next point is that the whole idea of process graphs only really works with a good library of components. It is beyond the scope of this thesis to provide such a library. Instead, this is subject to further work. Since arbitrary java code can be embedded in process graphs, also third party components like the ones from IBM's San Francisco project (`http://www-4.ibm.com/software/ad/sanfrancisco/`) can be integrated. However, third party components do not support the method signature expected by `component-call` elements. Instead, their methods have to be called by Java code embedded in the graph, which makes the graph less readable to non-programmers. Alternatively, wrapper components could be written. Note that this is a minor problem - the important point is that it's possible to make calls to any Java component if necessary.

3.5 Data exchange components

3.5.1 Overview

Figure 3.1 on page 48 shows a class diagram that illustrates the following explanations.

```
<process-graph name="com.abc.xyz.MakePrice">
   <components>
      <component type="enterprise-bean" alias="customerInfo"
       name="de.isbka.ecom.framework.CustomerHelper"/>
      <component type="bean" alias="priceHelper"
       name="com.abc.xyz.Pricing"/>
   </components>

   <graph>
      <component-call alias="customerInfo" method="getCustomerInfo"/>
      <if>
         <condition>
            <get-var name="Customer.isDealer" type="bool"/>
               <!-- This variable has been set by getCustomerInfo. -->
         </condition>
         <true>   <!-- give a discount -->
            <set-var name="Product.price" type="float">
               <get-var name="Customer.discountFactor" type="float"/>
                  <!-- This variable has been set by getCustomerInfo. -->
               *
               <get-var name="Product.price" type="float"/>
            </set-var>
         </true>
         <false>   <!-- add VAT -->
            <component-call alias="priceHelper" method="getVATFactor"/>
            <set-var name="Product.price" type="float">
               <get-var name="Pricing.VATFactor" type="float"/>
                  <!-- This variable has been set by getVATFactor. -->
               *
               <get-var name="Product.price" type="float"/>
            </set-var>
         </false>
      </if>
   </graph>
</process-graph>
```

Figure 3.2: Sample process graph

```
package com.abc.xyz;

import de.isbka.ecom.framework.ProcessingException;
import de.isbka.ecom.framework.ProcessGraph;
import de.isbka.ecom.framework.RequestContext;

public class MakePrice implements ProcessGraph {

   de.isbka.ecom.framework.CustomerHelper customerInfo_;
   com.abc.xyz.Pricing priceHelper_;

   MakePrice() {
      // create component instances (left out for brevity)
   }

   void traverseGraph(RequestContext context) throws ProcessingException {
      try {
         customerInfo_.getCustomerInfo(context);
         if (((Boolean) context.get("Customer.isDealer)).booleanValue()) {
            context.put("Product.price", new Double(
               ((Double) context.get("Customer.discountFactor")).doubleValue()
               *
               ((Double) context.get("Product.price")).doubleValue()
            );
         } else {
            priceHelper_.getVATFactor(context);
            context.put("Product.price", new Double(
               ((Double) context.get("Pricing.VATFactor")).doubleValue()
               *
               ((Double) context.get("Product.price")).doubleValue()
            );
         }
      } catch (ProcessingException pe) {
         throw e;
      } catch (Exception e) {
         throw new ProcessingException(e);
      }
   }

}
```

Figure 3.3: Java code created from figure 3.2

3.5.2 Interface of the framework

As explained in section 3.3, the framework communicates purely via XML documents (plus additional data, e.g. pictures or other binaries). That reduces its interface to basically one simple method that can be found in the so called *dispatcher* component:

```
public Vector dispatch(String request, Vector additionalData, String sessionId);
```

The vector `additionalData` can contain arbitrary data to accompany the request. This is not only useful for binary data, but also for attaching additional XML documents[2]. The `sessionId` identifies the current session. It can be generated by a servlet or other means. Note that the session id is not checked in the dispatcher, so this method should never be callable from outside. Instead, there has to be some filter (e.g. a servlet) that sets this id in a reliable way (see also section 3.7.2).

The return value of the dispatch method is also a vector that carries the actual XML response as its first element. If no additional data has to be transmitted, this is also the only element. Otherwise, arbitrary data can be found in the next elements. Note that the meaning of additional request or response elements must be indicated (implicitly or explicitly) in the request or response document respectively. For example, the response might contain some pictures which could be indicated in the response document by tags like `<picture number="3">`, where 3 would be the number of the element in the response vector that holds the picture data.

The dispatcher also has a second dispatch method - with the same signature as the first one - that doesn't validate the request. This is useful for processing requests that are constructed within servlets that are part of the application. Since there is a "reliable source" in this case, validating the requests would be a waste of time. The security features of the EJB container must be employed to ensure this method cannot be called from outside.

3.5.3 Structure of requests/responses

Which requests (i.e. incoming XML documents) are valid is determined by an XML schema. For example, figure 3.11 on page 60 shows a schema excerpt that defines requests for working with persistent objects. Every request is based on the type `request` the definition of which can be found in appendix A. The request type has only four optional attributes and no fixed elements (i.e. elements that would have to be present in every request). Using the `lang` attribute, the client can specify which language is desired for the response. The second attribute called `minResponse` tells the framework to respond with only a status message. To see why this makes sense, consider the example of putting an item into the shopping cart. The response for the interactive user lists the current contents of the cart. In an automated scenario where an ERP system or other program communicates with the framework, this is not of interest. Instead, all that is necessary is a status message indicating success or failure. The `sessionId` and `requestId` attributes are also used for automated processing and are described in section 3.5.6. A response looks similar to a request, except the `minResponse` and `lang` attributes are missing. The latter is due to the fact that the response may contain several pieces of text that have their own language attributes. If not all text fragments are available in the desired language, the missing ones are returned in a configurable default language.

3.5.4 Request processing

The sequence of events for dispatching a request is shown in figure 3.4. Note that all objects except the session context and the request context may be reused for another request, so they are not destroyed. The individual steps of the sequence are explained in the next paragraphs.

Step 1 A client, e.g. a servlet, calls the `dispatch()` method to send an XML request to the application.

[2] It can be problematic to include one XML document in another if CDATA sections are used.

Step 2 The dispatcher checks if a session context for the specified id already exists. If this is the case, it is loaded from the database. Otherwise, a new session context is created and initialized with the variables shown in table 3.1 (see also section 3.4.2). The prefix "fw." is used for all variables set by the dispatcher.

Step 3 A request context is created and initialized with the variables shown in table 3.2.

Step 4 After the contexts have been initialized, the pre-session process graph (see 3.4) is traversed[3], provided the session context was newly created and not loaded from the database. If the pre-session graph terminates with an exception, the session context is invalidated. No further processing is done and an error message is returned to the client.

Step 5 If the pre-session graph was processed successfully (or not at all for existing sessions), the request is schema validated using one or more *schema processors*. A schema processor is simply a validating XML parser that can validate against XML schemas. Examples are *Xerces-J* from the Apache group ([Apa]) or *XML Schema Processor* from Oracle (`http://technet.oracle.com/tech/xml/`). Using more than one processor can be useful to get around bugs in individual implementations that would otherwise let invalid documents pass. Since powerful validation rules can be specified in schemas (much better than in DTDs), almost no validation is necessary after the schema processor agreed with the request. For example, regular expressions can be used in schemas to describe valid element and attribute values. Also, a lot of different datatypes are supported, including several decimal and date/time types. If validation fails, an error message is returned to the client.

Step 6 After the request has been validated, the dispatcher traverses the pre-request graph. At this stage, access rights checking and similar activities can take place. For example, in a simple system it might be enough to grant rights based on the types (i.e. root elements) of requests (see also section 3.7). When access is denied or another error occurs, the pre-request graph terminates with a processing exception. The dispatcher puts such an exception into the request context under the name "fw.ProcessingException" and execution continues with the post-request graph. This behaviour is different from the reaction to failures before that step. The reason is that the post-request graph can now rely on a valid request, which wasn't the case before.

Step 7 If the pre-request graph was traversed successfully, the dispatcher examines the root element of the request and traverses the process graph that is associated with it. Note that the graph is actually a Java class - as explained in section 3.4 - that is instanciated rather than a document that is parsed. If the process graph terminates with an exception, the dispatcher puts it into the request context using the name "fw.ProcessingException".

Step 8 Finally, the post-request graph is traversed (even if traversal of the previous graph resulted in an exception). Note that there is no post-session graph, since it depends on the actual application how a session is ended (e.g. by a specific logout request). Thus, processing that should be carried out at the end of a session should be placed either in a process graph for a logout request or in the post-request graph (of course inside an `if` element that checks if the session - and not just the request - is finished). Session timeouts can easily be realized within servlets that issue a logout request after the timeout period has elapsed[4]. A session can only be closed and its persistent representation removed through the `invalidate()` method of the session context.

Step 9 This step is explained in the following section about session context persistence.

[3] Note that the request has not been validated at this stage! This is an arbitrary design decision and means that a session can start with an invalid request.

[4] Though there must probably also be regular "clean ups" on the application server side.

Step 10 After processing of a request is complete, the dispatcher checks if the request variable "fw.-ProcessingException" is set or a processing exception has been thrown by the post-process graph. In this case, the dispatcher looks up an associated message text and generates an error message document as the response. An example for such a message is shown in figure 3.6. The `param` tag encloses all dynamically inserted information (e.g. to display it in an emphasized style when showing the message to the user). If no error occured, the dispatcher checks if a response has been generated during process graph traversal. If not, a default success message as shown in figure 3.5 is sent. Otherwise, the generated response and - if present in the request context - associated additional data is returned to the client.

If, for some reason, any of the process graphs cannot be instantiated or throws an exception different from a processing exception, the dispatcher generates a wrapping processing exception with an appropriate id and continues as if the process graph in question had thrown a processing exception.

If the `minResponse` attribute was specified in the request, the framework ignores the response generated by the process graphs and replaces it with a default success message. The process graphs themselves are responsible for saving processing time by not generating a response when the `minResponse` attribute is present.

The actual structure of the XML documents exchanged is not important for the work of the dispatcher or any filter elements (like servlets). The dispatcher only examines the root element to determine which process graph to traverse. It also checks if there is a `lang` attribute attached to the root element to determine the language to use for success or error messages.

3.5.5 Session context persistence

The session context must be preserved between requests. However, as explained in section 3.3.3 the dispatcher is a stateless component, so it cannot remember the session context as part of its state. Instead, the session context is written to the database as a serialized object and loaded again for the next request (using a simple entity bean as a wrapper). A session context can be marked invalid with the `invalidate()` method. Just before sending a response to a client, the dispatcher checks if the current session is still valid. If not, it gets removed from the database.

3.5.6 Automation

To support non-interactive (i.e. batch) processing, an *automation servlet* receives requests via HTTP POST. Since there is no browser that issues the requests, the usual session management does not work. Instead, the automation servlet inserts a `sessionId` attribute in the root element of every response. This id can then be specified by the client in subsequent requests.

To further support automation, *request bundles* can be used to send more than one request at a time to the automation servlet. An example is depicted in figure 3.7. Before the actual requests, two elements specify how to handle problems with request processing. The `do-not-report` element contains all message severities that should not be sent in the corresponding *response bundle* (figure 3.8). In the example, all success messages (severity="information") are suppressed in the response. Using `continue-if`, the issuer of the request bundle can specify what should happen when processing of one request fails. In the example, warnings are ignored, so processing stops only when an error occurs. By default, only information (i.e. success) status allows subsequent requests to be dispatched. Using the `requestId` attribute, the issuer can relate the returned responses to the individual requests he or she has sent. The automation servlet simply copies these ids to the responses.

All requests within a request bundle are handled as part of the same session. Session ids that are specified in the requests are ignored. Instead, the response bundle has a `sessionId` attribute that can be used in subsequent request bundles to make them part of the same session. Security issues - namely authentication and encryption - are handled in the same way as in the interactive case (see 3.7). This is probably not appropriate for many applications, so additional security features may have to be introduced in the future to better support automated use of the framework.

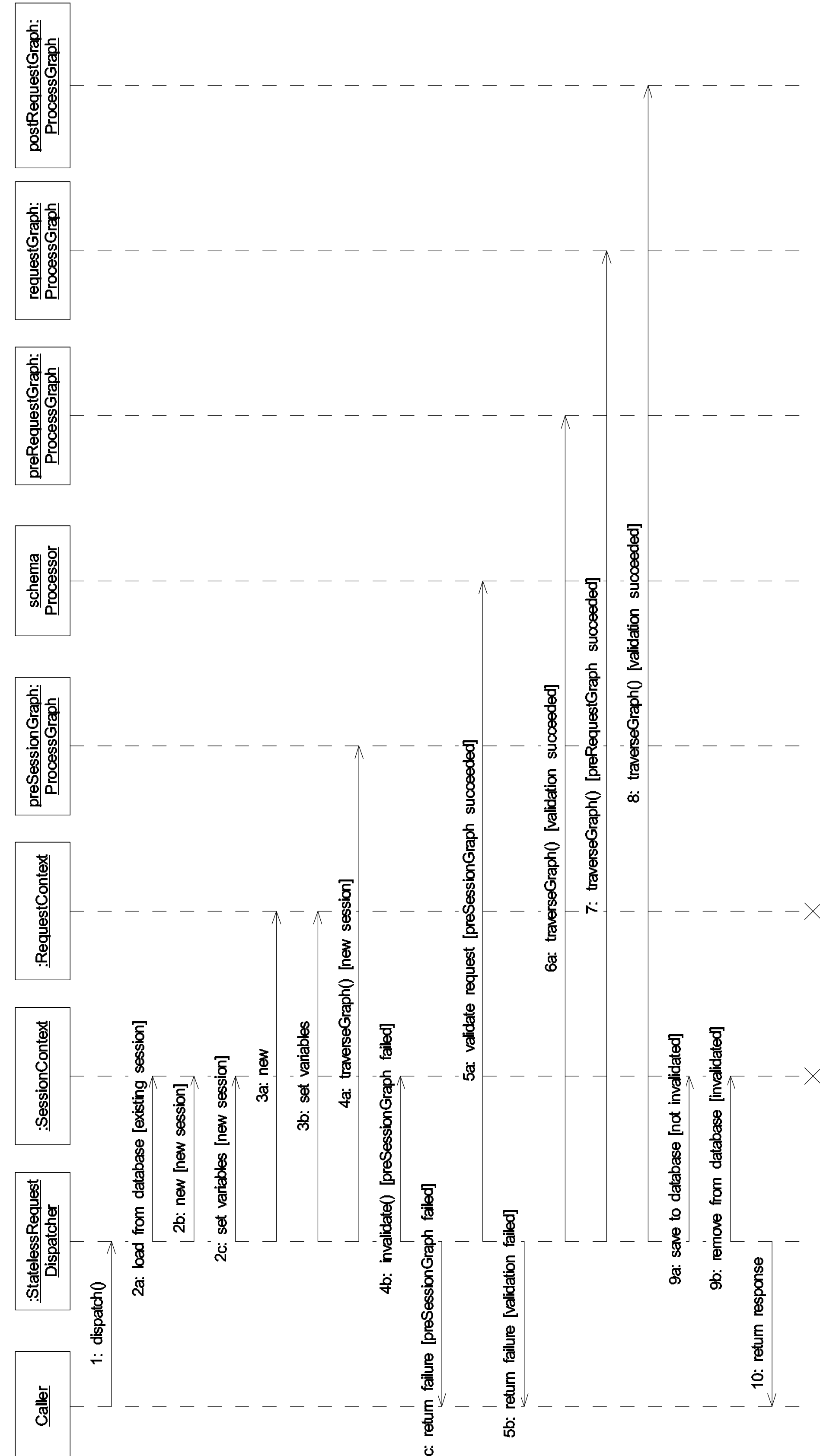


Figure 3.4: Sequence diagram for processing a request

fw.UserId (`Long`)	The id of the user that started the session. This id may change after an anonymous user is authenticated (see section 3.7.4).
fw.StartDate (`Date`)	The date and time this session was started.
fw.LastUsedDate (`Date`)	The date and time the most recent request in this session was made.

Table 3.1: Pre-initialized session variables

fw.Request (`StringBuffer`)	The current XML request.
fw.Response (`StringBuffer`)	The current XML response.
fw.AdditionalRequestData (`Vector`)	Additional data for the request (e.g. binary data).
fw.AdditionalResponseData (`Vector`)	Additional data for the response (e.g. binary data).

Table 3.2: Pre-initialized request variables

```
<message id="fw.Success" severity="information">
    <text lang="en">
        Your request of type <param>add-productAccessRight-request</param>
        was processed successfully.
    </text>
</message>
```

Figure 3.5: Example for a success message

```
<message id="fw.product.NotFound" severity="error">
    <text lang="en">
        The product with id <param>ABC-123</param> could not be found.
    </text>
</message>
```

Figure 3.6: Example for an error message

```
<request-bundle sessionId="abcd">
    <do-not-report>
        <severity>information</severity>
    </do-not-report>
    <continue-if>
        <severity>warning</severity>
    </continue-if>
    <requests>
        <first-request requestId="1" ... />
        <second-request requestId="2" ... />
    </requests>
</request-bundle>
```

Figure 3.7: Example for a request bundle

```
<response-bundle sessionId="abcd">
   <message id="some.message" requestId="2" severity="warning">
      ...
   </message>
   <message ... >
   </message>
</request-bundle>
```

Figure 3.8: Example for a response bundle

3.6 Data storage components

3.6.1 Support by Enterprise JavaBeans

When using Enterprise JavaBeans, there are two possible methods for persisting data. One is called *bean managed persistence* (BMP), the other is *container managed persistence* (CMP). BMP means that the persistent components are responsible for storing their state. CMP on the other hand is carried out completely by the EJB container.

CMP has one major advantage: There is no need for explicit calls to a data storage API (like JDBC). This way, the components are not tied to a specific kind of data store and are easier to develop. It also separates business and persistence logic that are tied together with BMP. On the other hand, version 1.1 of the EJB specification provides only limited CMP support. Relationships between components are not covered, nor are dependent objects (i.e. objects belonging to the state of a component that are of complex types). Also, there is no standard query facility - instead, queries are handled in a container specific way. All these issues are addressed by version 2.0 of the EJB specification (see `http://java.sun.com/ejb`). Unfortunately, it is still in the public draft status, and working implementations cannot be expected before mid 2001.

Another alternative would be to use an *object relational mapping* tool. However, these tools are expensive and often introduce dependencies that make it hard to switch to another vendor later on. The only satisfying mapping solution I found was TOPLink for WebLogic. With TOPLink, one can develop CMP beans without any special requirements. Instead, the mapping is accomplished at deployment time. The combination of TOPLink and BEA WebLogic indeed works very well. However, it is the only solution of this kind available today. With the upcoming EJB 2.0 standard in mind, it doesn't seem to be a clever idea to rely on a proprietary solution for a framework that is supposed to work in a variety of environments.

Despite all these problems, I still believe that CMP is the right way to go. As it will evolve, it will turn out much more powerful than any self-written persistence mechanism. For now, the most urgent problems with the current CMP version are solved by a special query component introduced in section 3.6.3. No mapping tool is used, so the framework can be deployed in any EJB 1.1 compliant container with no specialized runtime or development software necessary.

3.6.2 Using XML schema to describe persistent datatypes

3.6.2.1 Introduction

As mentioned in section 3.3, the structure of all persistent data of the framework itself and of custom extensions is described using an XML schema. Via XSLT, an input schema is transformed to EntityBeans, other Java classes, and SQL scripts.

XML schema is largely based on datatypes. It offers *simple types* - like integer or string - and *complex types* that itself can contain elements of other types (much like a Pascal record or a C struct). Special top level complex types (i.e. complex types that are not part of other complex types) are used by the framework as input for generating persistent components. These complex types are referred to in this section as *dataclasses*. The process of generating source code and other files out of them is called *persistence transformation* and is carried out by several XSLT stylesheets.

To explain the persistence transformation, figures 3.9 and 3.10 show an example of the XML schema input that is processed. They also demonstrate the framework paradigm on the persistence level.

Figure 3.9 shows a simplified excerpt from the framework's schema. The complex type "product" is marked as a dataclass using XML schema's annotation mechanism and special elements introduced by the framework (`generate` and `dataclass` in a namespace with prefix `gen`). The `appinfo` element can, in general, carry arbitrary application specific information and is ignored by schema processors. The product type has only a single element that is used to identify a product. It is of type "sequenceNumber" that is defined in the first lines of the example as an integer with a maximum of 15 digits. The `appinfo` element is again used to give hints to the stylesheets that cannot be found in the schema itself. This time, the element `ref` indicates that the product id is part of the primary key (in this example, it's the only part of the primary

key). The attribute `artificial` tells the stylesheets to generate code for producing artificial values for this element.

The whole product type is abstract, since it is by far no complete description of a product. It is, however, everything that is really required with exactly the specified element types by the framework. Other elements, like the product's name, can have custom validation rules and other properties that are not prescribed by the framework. Only the id has to follow the framework's rules, or it would be impossible to lookup or refer to a product in a generic way.

The second complex type shown is the default for a complete product (simplified here for clarity). ”defaultProduct” is derived from ”product” (via XML schema's `base` attribute) and adds the new element ”productName” of XML schema's built-in string type. The default product is also abstract, which doesn't seem reasonable at first. The explanation is that XML schema has no way of taking back a declaration like ”defaultProduct can be used anywhere product can be used”. To enable developers to define their own notion of ”product” - where the default is no valid replacement - the default must be abstract. If desired, it can be used without changes by deriving an empty dataclass from it.

Until now, there is no valid form of a product, so figure 3.10 introduces one. It shows an excerpt from a possible extension of the framework's schema. The default for products is extended by another attribute, and this time the dataclass is concrete. The `gen:interface`, `gen:not`, and `gen:edit` tags are explained in the next section.

```
<simpleType name="sequenceNumber" base="integer">
   <precision value="15"/>
</simpleType>

<complexType name="product" abstract="true">
   <annotation>
      <appinfo>
         <gen:generate>
            <gen:dataclass/>
         </gen:generate>
      </appinfo>
   </annotation>
   <element name="productId" type="sequenceNumber">
      <annotation>
         <appinfo>
            <gen:generate>
               <gen:ref artificial="artificial"/>
            </gen:generate>
         </appinfo>
      </annotation>
   </element>
</complexType>

<complexType name="defaultProduct" abstract="true" base="product">
   <annotation>
      <appinfo>
         <gen:generate>
            <gen:dataclass/>
         </gen:generate>
      </appinfo>
   </annotation>
   <element name="productName" type="string"/>
</complexType>
```

Figure 3.9: Sample schema fragment (framework)

```
<complexType name="myProduct" base="defaultProduct">
   <annotation>
      <appinfo>
         <gen:generate>
            <gen:dataclass/>
            <gen:interface/>
         </gen:generate>
      </appinfo>
   </annotation>
   <element name="productDescription" type="string"/>
   <element name="someAttribute" type="someType">
      <annotation>
         <appinfo>
            <gen:generate>
               <gen:not>
                  <gen:edit/>
               </gen:not>
            </gen:generate>
         </appinfo>
      </annotation>
   </element>
</complexType>
```

Figure 3.10: Sample schema fragment (framework customization)

```
<complexType name="add-myProduct-request" base="request">
   <element ref="myProduct-add"/>
</complexType>

<complexType name="edit-myProduct-request" base="request">
   <element ref="myProduct-ref"/>
   <element ref="myProduct-edit"/>
</complexType>

<complexType name="get-myProduct-request" base="request">
   <element ref="myProduct-ref"/>
</complexType>

<complexType name="get-myProduct-response" base="response">
   <element ref="myProduct-get"/>
</complexType>

<complexType name="search-myProduct-request" base="request">
   <element ref="myProduct-search"/>
</complexType>

<complexType name="search-myProduct-response" base="response">
   <element ref="myProduct-get" minOccurs="0" maxOccurs="unbounded">
</complexType>

<complexType name="delete-myProduct-request" base="request">
   <element ref="myProduct-ref"/>
</complexType>
```

Figure 3.11: Generated request types

```
<edit-myProduct-request>
    <myProduct-ref>
        <productId>12345</productId>

        <!-- If the primary key was comprised of more than one
         field, the other ones would appear here. -->

    </myProduct-ref>
    <myProduct-edit>
        <productName>Hot beans</productName>
        <productDescription>Software development tool</productDescription>
    </myProduct-edit>
</edit-myProduct-request>
```

Figure 3.12: Sample client request

3.6.2.2 Generated extension schema and XML interface

For every input schema, an additional schema (called *extension schema* here) is generated. The extension schema serves two main purposes: Firstly, it holds preprocessed versions of the dataclasses for later transformation steps (including entity bean generation). Secondly, it contains complex types that describe an XML interface to the dataclasses. The extension schema should be included (using XML schema's `<include>`) in the original schema to have a single document that can be fed to a schema processor.

To explain what the extension schema contains, take a look at figure 3.10. The tag `gen:interface` instructs the transformation to generate the request and response types shown in figure 3.11 (see also section 3.5). There are requests for adding a new instance to the database, editing an existing instance, searching an instance by example, retrieving an instance via a known reference (i.e. primary key), and removing an instance from the data store. The responses are messages as shown in figures 3.5 and 3.6 on page 57 except for successful get and search requests where the responses are defined by the extension schema. Note that a search request is considered successful even when no matching instance is found (and, of course, no errors occurred).

The `ref` attributes of the elements in figure 3.11 are an XML schema mechanism that allows referencing global elements (i.e. elements that are not part of a complex type or another element).[5] There are six of these elements that are generated in addition to the request types:

1. A -db element that contains all persistent elements of the dataclass. By default, all elements are included.

2. A -ref element containing the dataclasses elements that make up the primary key. By default, no elements are included.

3. A -get element that determines which elements are returned when a dataclass instance or a collection thereof is requested by a client (e.g. as search results). By default, all elements are included.

4. An -add element that describes all elements which are required for creating a new instance. By default, all elements but artificial primary keys are included.

5. An -edit element that contains all editable elements of the dataclass. By default, all elements but primary keys (artificial or not) are included.

6. A -search element that contains all elements appropriate for a query-by-example search. By default, all elements are included. For each numerical or date element in the source schema, two elements are generated in the extension schema for the upper and lower boundary.

[5] These elements are the equivalent of elements in a Document Type Definition (DTD).

The default behaviour can be changed by a set of simple tags. For example, the `gen:ref` tag in figure 3.9 specifies that ”productId” is part of the primary key, while the `gen:not` and `gen:edit` tags in figure 3.10 exclude ”someAttribute” from the set of editable elements (i.e. it can be specified on creation of an instance but not changed afterwards). Figure 3.12 shows a sample edit request that can be sent to the framework by a client.

Besides the request types, also a *handler component* and a process graph (see section 3.4) are generated for each dataclass. The handler component transforms the requests into entity bean operations, while the process graph only includes a `component-call` to the handler component. The graph does not include any access checks, so it should be modified appropriately before registering it with the framework. Alternatively, a pre-processing graph can handle these checks. In addition to the logic on the application tier, also servlets are generated that provide a simple interactive interface for the requests using HTML forms (shown in figure 3.38 on page 90).

To provide for simple adjustment of the default behaviour, each generated handler component has the following overridable methods:

```
protected void addInstance(<ContentClass> newContent)
    throws ProcessingException;

protected void editInstance(<PrimkeyClass> primkey, <ContentClass> newContent)
    throws ProcessingException;

protected void getInstance(<PrimkeyClass> primkey)
    throws ProcessingException;

protected void searchInstances(<ContentClass> example)
    throws ProcessingException;

protected void deleteInstance(<PrimkeyClass> primkey)
    throws ProcessingException;

protected <ContentClass> getInstance_(<PrimkeyClass> primkey)
    throws ProcessingException;

protected Object[] searchInstances_(<ContentClass> example)
    throws ProcessingException;
```

One out of these methods - depending on the type of request - is called after the request has been parsed. `<ContentClass>` and `<PrimkeyClass>` stand for classes that are part of the generated entity bean (see next section). All public methods return `void` since the results of a successful call to `getInstance()` or `searchInstances()` are directly written to the current response (see section 3.5). The other methods leave it to the request dispatcher component to generate a success message. Errors result in a processing exception that is also translated to an XML message by the dispatcher. The last two methods are intermediate methods. For example, `searchInstances_()` is called during execution of `searchInstances()`. These methods can be used as helpers when overriding `getInstance()` or `searchInstances()`. Note that the search helper method returns an `Object` array since this is the return type it receives from the query component (see 3.6.3) and it would be a waste of time to convert this to a more specific array.

Note that binary data in get and search responses are returned as additional response elements (described in section 3.5) in the order they appear. The elements in the response are marked accordingly which is shown in figure 3.13.

3.6.2.3 Generated entity beans

Each dataclass (both abstract and concrete) is translated into several files. Together they form an entity bean or (in case of abstract dataclasses) a part of an entity bean. All generated beans receive and return their persistent

```
<get-image-response>
   <image-get>
      <imageId>123</imageId>
      <key>someImage</key>
      <mimeType>image/gif</mimeType>
      <imageData binary="binary"/>
        <!-- The actual image data can be found in the
         additional response data. -->
   </image-get>
</get-image-response>
```

Figure 3.13: Sample response containing binary data

state as instances of so called *content classes*. These classes serve as containers for a bean's state. If all attributes of an entity bean were accessed using individual getter and setter methods, this would mean a lot of RMI[6] calls which could affect performance heavily. By using content classes, a single call can get or set the bean's entire state. More fine grained access can be implemented as needed.

All primary key fields of a bean are represented by a primary key class (for consistency, this is even the case for single field primary keys). The primary key fields are also part of the content class. When the content of an existing instance is set, their values are ignored since the primary key cannot change.

Each content class also implements the `Content` interface with the following methods[7]:

```
public void copyToContext(Context ctx, String prefix);

public void assignFromContext(Context ctx, String prefix)
    throws IllegalArgumentException;

public void prepareQuery(Hashtable params, Vector skipExpressions)
    throws NullPointerException;
```

This interface is useful for working with the field values within a process graph. The first method copies all fields of the instance to the given context, creating one context variable per field with the name being the specified prefix followed by the field's name taken from the XML schema. The `assignFromContext` method copies the fields from the context to the content object. Fields that cannot be found are set to `null` while fields of the wrong datatype result in an `IllegalArgumentException`. Missing fields that are not nullable according to the schema are by default not detected by the content class. Instead, such fields lead to an exception when trying to write them to the database. This behaviour may need to be changed or made configurable after some experience in working with the framework.

All together, the following files are generated for each dataclass:

1. An entity bean's remote interface (even if the type is abstract). The remote interface contains methods to retrieve the bean's primary key and to retrieve and set its content.

2. An entity bean's home interface (even if the type is abstract). The home interface contains a create method that has the new instance's content (in form of a content class instance) as parameter. There are also methods for finding an instance by its primary key and for finding all instances. Note that the finders are implemented by the EJB container.

3. An entity bean's implementation (even if the type is abstract). This class implements the methods in the remote and home interface (except for the finders). If the dataclass is abstract, the implementation class is also abstract.

[6] Remote method invocation, the Java equivalent to a remote procedure call.

[7] During implementation of the framework, the need to introduce additional methods to this interface might arise.

4. An entity bean's primary key class (even if the type is abstract). Primary key fields must only occur on one inheritance level. That means that a composed primary key must not be split over several dataclasses. However, a primary key class is generated for each dataclass, regardless of whether or not it contains primary key fields. This behaviour enables more readable and consistent implementations, even if there is only one "real" primary key class.

5. A content class as explained above.

6. An adaptor to the type of database used (see 3.6.3).

7. If the dataclass is not abstract, a deployment descriptor for the generated bean.

8. If the dataclass is not abstract, a container specific deployment descriptor. Most EJB containers need such an additional descriptor for deployment. It is often generated by tools provided with the container. The container specific descriptor contains bean to table mapping information and definitions of the queries that should be used to implement the finder methods. Generating this descriptor as part of the persistence transformation rather than an extra deployment step simplifies deployment. However, this part of the transformation has to be individually adapted to the target container.

The classes and interfaces are not generated directly but rather the source code for them. As an example, section C.1 shows the stylesheet for generating the remote interface of a bean. Note that many files are also generated for abstract dataclasses to enable abstract access to the generated entity beans. This is necessary to enable "plug and play" for the beans.

The generated beans can be customized (e.g. in order to add business logic) by inheriting from the generated classes rather than populating them with own code. This way, the usual "safety comments" like

```
// --- abc123-preserved-code-begin ---
// Insert your own code here
// --- abc123-preserved-code-end ---
```

can be avoided.[8] A similar approach is taken for the deployment descriptors. For every DD, a custom version can be written that only contains additional or overridden information. These custom descriptors are merged together with the generated ones as part of the persistence transformation. Custom descriptors always take precedence over generated ones.

For every file mentioned above, also a customization template is generated that contains a skeleton of how a customization looks like. For example, a bean implementation template contains an empty class that inherits from the generated implementation. Figure 3.14 gives an overview over the resulting inheritance hierarchy for the schema fragments shown in figures 3.9 and 3.10. Note that the customization templates (located in `de.isbka.ecom.framework.entitybeans` and `com.heidelberg.HILS2.entitybeans`) are part of the inheritance hierarchy. This is necessary to propagate customizations to the classes further down the tree. In fact, when generated classes or interfaces use other generated classes/interfaces, they reference the customization classes rather than the generated ones. Otherwise, there would be no use customizing anything!

The package names shown are only an example and can be freely configured. The "class inflation" visible in figure 3.14 (36 classes for a simple example, not including the ones generated by the EJB container!) could be reduced in several ways:

- Don't generate customization templates. This way, safety comments would have to be used.

- Don't generate a primary key hierarchy. Only one primary key class is really necessary, the rest is there to be consistent with the other generated classes and to enable adding "convenience methods" like generating the contents of the actual primary key fields from some other data.

[8]For example, Persistence PowerTier relies on safety comments.

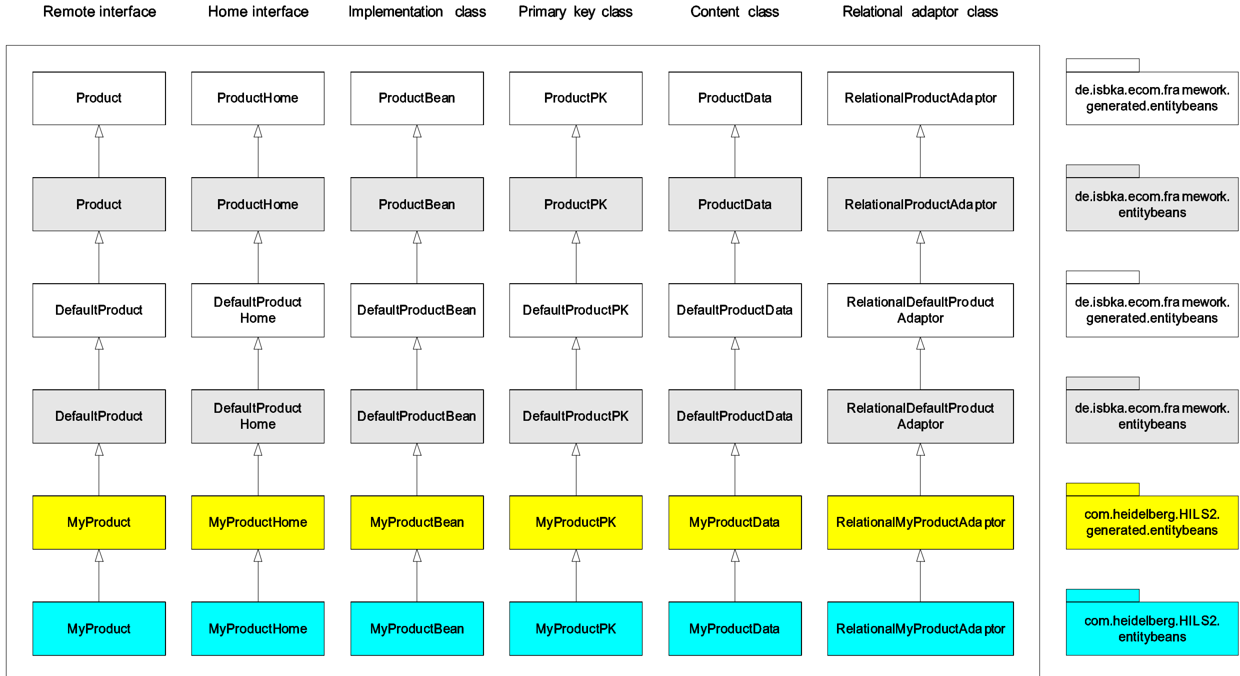


Figure 3.14: Hierarchy of generated classes

- Don't generate a content class for the default product since it's abstract and never used directly. However, the generation of subclasses would be more complicated then.

None of the above is currently considered for the framework since the number of classes is not really a problem. Most of them are tiny, and an experimental JAR containing about 150 of these classes turned out to be less than 100 KB in size. At runtime, they surely eat up memory, but this is nothing compared to what application server, web server, and database need. Performance is hardly affected, since there are many *classes* but only two *objects* (the container generated EJB object and the bean implementation[9]) that make up a single bean instance at runtime. Thus, object creation (which can be quite expensive in Java) is kept to minimum. There are some methods that form a chain of base class calls (i.e. every method calls the same method in the base class). However, compared to runtime costs caused by RMI and database access, this is negligible.

3.6.2.4 Data type conversions

Conversion of schema datatypes to Java and SQL datatypes is done according to table 3.3. The SQL types are partly Oracle 8 specific. For other databases, according types have to be used.

3.6.2.5 Other files

For every simple type in the source schema that contains an `enumeration` facet (see [W3Ca]), a Java class is generated that contains the possible enumeration values as static final strings. However, the Java type in the entity beans using that simple type is still just a string (which should simplify things a lot and, since all input is schema-validated, should do no harm).

For all dataclasses, SQL CREATE and DROP statements are generated to simplify database generation. Of course, this is only useful when working with a relational database.

[9]Primary key and content class instances are only generated on request.

Schema type	*Java type*	*SQL type*
string [maxLength not specified]	`java.lang.String`	LONG
string [maxLength specified]	`java.lang.String`	VARCHAR2(maxLength)
language	`java.lang.String`	VARCHAR2(10)
integer [precision < 10]	`java.lang.Integer`	NUMBER(10)
integer [precision < 19]	`java.lang.Long`	NUMBER(19)
integer [precision >= 19]	`java.math.BigDecimal`	NUMBER(precision)
decimal [precision < 15]	`java.lang.Double`	NUMBER(precision, scale)
decimal [precision >= 15]	`java.math.BigDecimal`	NUMBER(precision, scale)
timeInstant	`java.sql.Timestamp`	DATE
binary	`byte[]`	LONG RAW

Table 3.3: Datatype conversions

3.6.3 The query component

3.6.3.1 EJB 1.1 issues

As mentioned in section 3.6.1, one of the weaknesses of EJB 1.1 container-managed persistence is the lack of a standardized query facility. Every container handles the implementation of finder methods differently. Bull's JOnAS wants to be provided with an SQL WHERE clause during deployment, while BEA WebLogic has its own, rather primitive query language. Both of them only support relational databases - at least without third party tools.

Another problem that can grow rather huge is performance. Consider the example given in figure 3.15. This query finds all purchase order positions with a product price above a certain limit and a product taxation tag that is equal to one of two parameters. Returned are the date of the order and the name of the purchaser as well as name and price of the product. This example is almost trivial to formulate in SQL. The first problem for EJB 1.1 containers is that there are three tables/entity beans involved. In case a container doesn't support joins (like BEA WebLogic), at least three database queries are necessary, one for the order heads, one for the positions, and one for the products. For complex queries, this is a true performance killer. Things get even worse if we assume that the product table has a lot of fields or even contains binary data like pictures. If one database record has a size of 1 to 20 KB, then for 100 records 100 to 2000 KB are transmitted from the database to the application server. But what is needed are only three fields, which might have a size of 100 bytes. For 100 records, this would mean only transmitting about 10 KB[10]!

Yet another aspect are dynamic queries. If we want a simple query by example, this cannot be done with finder methods. A finder method can only have a static set of parameters, so it would have to be designed for the maximum possible number (i.e. the number of fields in the bean). Even then, it won't function properly. The reason is that container-generated finder implementations are static[11], so there can only be one SQL statement per finder. But if we have something like

```
SELECT ... FROM ... WHERE field1=? AND field2=? ...
```

there is no way to make the query based on fewer fields. If only field1 is known, there must be no AND clauses for the other fields. The only EJB 1.1 conformant solution would be to create finders for every possible combination of fields, which is absolutely infeasible.

In summary, there are three main obstacles to overcome. The first one are joins, the second one are projections (i.e. reading only some fields of a bean), and the third one are dynamic queries[12].

[10] Not including the necessary overhead. Of course the XML transmitted from and to the application server is also a waste of bandwidth - but XML can easily be compressed, and there are good reasons for an XML interface.

[11] With the exception of specialized containers or third party tools like the BEA WebLogic/TOPLink combination.

[12] Note that dynamic queries also solve the projection problem.

3.6.3.2 Solutions on the query expression level

One common solution to the above mentioned problems is to design an abstract query framework that is used to make up complex and/or dynamic queries. While this is nice to have, it cannot reasonably be done within the scope of this thesis (at least with a proper design). The other extreme is letting the business components generate SQL as they need and feed it to the database. This, of course, ties the business logic to the relational paradigm and makes changes complicated.

For this thesis, I've decided for a compromise that is not too hard to realize while maintaining a reasonable abstraction from the database. If a complete query framework should be avoided, then how else can queries be built? First of all, without an abstract query language (of whatever kind), queries have to be stored in their "natural" language - which is SQL in case of relational databases. To support different kinds of datastores, the same query has to be expressed in different query languages (SQL, OQL, ...), so the basic idea is to store predefined queries in different dialects with the possibility to add more as needed. On the query expression level, this solves both the join and projection issues depicted above. Projections can be realized by using one query per desired projection. Later on, I will show how to handle projections on the object level.

What remains is the problem of dynamic queries. Completely dynamic queries are impossible with the proposed "multi-lingual" query approach. Instead, I looked at what is needed most of the time: Leaving out parts of complex queries, like in a query by example where only some fields are known. This leads to the idea of marking parts of the query statements as "optional" in some way.

As an example, take a look at figure 3.15. This query finds all purchase order positions with a product price above a certain limit and a product taxation tag that is equal to one of two parameters. Returned are the date of the order and the name of the purchaser as well as name and price of the product. Now suppose only one alternative for the taxation tag is known. Then one of `p.taxation=?` comparisons has to be left out. If no information about the taxation is given at all, let's assume "AB" should be used. To achieve this, the query is reformulated (using XML) as shown if figure 3.16.

```
SELECT o.orderDate, o.purchaser, p.productName, p.productPrice
FROM Product p, Order o, OrderPosition opos
WHERE
    opos.productId = p.productId
AND
    opos.orderId = o.orderId
AND (
    p.taxation=? OR p.taxation=?
) AND
   p.productPrice > ?
```

Figure 3.15: Sample query (SQL)

Using the `language` attribute, the same query can be expressed in different query languages. Every optional part of the query is enclosed in an `expression` element. Such an element contains both the original expression and a replacement in case the expression should be left out. If nothing about the taxation is known, the whole expression named "taxation" is replaced by an expression that compares it to "AB". If only one possibility for the taxation is given, the "taxation2" expression is replaced by an expression that always evaluates to false. This example shows that expressions can be nested as needed. The opportunity is also taken to introduce named parameters instead of JDBC's optional parameters. When a query is reformulated and the order in which the parameters appear is changed, the code that uses this query must also be changed. This is avoided here by giving the parameters names.

Creating the final query text is fairly straightforward. The whole XML for the query can be parsed in-order. If an expression element is encountered, it is checked whether or not the client of the query component wants it to be replaced. Then either parsing continues with the expression text or the replacement. Note that by using replacements instead of just leaving out text, the ANDs and ORs can just stay where they are. Otherwise more complex mechanisms would be necessary to optionally keep or throw away such operators.

```
<query id="test-query">
    <objects>
        <object alias="Order_1" class="Order"/>
        <object alias="Product_1" class="Product"/>
    </objects>
    <query-text language="SQL">
        SELECT
            o.orderDate AS Order_1_orderData,
            o.purchaser AS Order_1_purchaser,
            p.productName AS Product_1_productName,
            p.productPrice AS Product_1_productPrice
        FROM
            Product p, Order o, OrderPosition opos
        WHERE
            opos.productId = p.productId
        AND
            opos.orderId = o.orderId
        AND (
            <expression key="taxation">
                <expression-text>
                    p.taxation = <param key="tax1"/> OR
                    <expression key="taxation2">
                        <expression-text>
                            p.taxation = <param key="tax2"/>
                        </expression-text>
                        <replacement>1=0</replacement>
                    </expression>
                </expression-text>
                <replacement>
                 p.taxation = 'AB'
                </replacement>
            </expression>
        ) AND
            p.productPrice &gt; <param key="price"/>
    </query-text>
</query>
```

Figure 3.16: Sample query (spiced with XML)

If a `param` element is found, it is replaced by a '?' and the query component remembers the param key together with the current count of question marks that have been generated. This makes it possible to set the parameters later on[13]. The replacement approach works for arbitrary query languages since the actual query syntax stays completely transparent. Of course, there is still the need for specialized query components per datastore (RDBMS, OODBMS) that know how to specify parameters and submit the generated query (e.g. via JDBC).

3.6.3.3 Solutions on the object level

The previous section showed how various EJB problems can be solved on the query expression level. Now there must also be a way to make use of that on the object level. Again, several complications must be overcome, where the first one is once more about performance. If we have a JDBC resultset, how can we turn that into entity bean instances? The only way that doesn't involve container-specific "hacks" is by calling `findByPrimaryKey()` for each and every row in the resultset. This means an additional database call per result row, which is just unacceptable regarding performance. So instead of *entity beans* the query component instantiates *content classes* as described in section 3.6.2.3.

[13] And also to set the same parameter more than once. With pure JDBC, it sometimes cannot be avoided to set the same value for different '?' parameters that should actually be the same.

The next question is how the state of the content objects is set. This could be achieved in a generic way by dynamically figuring out which setter methods to call, converting the resultset column to the correct type and issuing the call via dynamic method invocation. However, this is both complicated to implement and slows down object creation (though the performance loss would probably be rather small). Instead, a relational adaptor class is generated for each content class. The adaptor is called for each row and converts it into a corresponding content object.

More than one object of the same class can be returned for a row in a query's resultset. For this reason, all returned objects for one row are identified by an alias as shown in figure 3.16. In this example, two relational adaptors are instantiated, one for the alias "Order_1" and one for "Product_1". In the query's resultset, the columns belonging to a specific alias are prefixed with the alias and a following underscore. The remaining part of the column label identifies the individual fields.

If an update of a result object is desired, the returned content object is not sufficient. Instead, the client has to lookup the corresponding bean instance via its `findByPrimaryKey()` method and then set its content as needed. To enforce data integrity, query and update can be part of the same transaction (see 3.6.3.6).

3.6.3.4 Interfaces of the query component and the relational adaptors

The query component itself is an entity bean. It has a string id to lookup a certain query and a field containing the query texts in at least one dialect (SQL, OQL, ...). Which dialect is chosen at runtime depends on the implementation of the bean. A relational implementation takes the SQL text, an OO implementation uses OQL and so on. The implementation that is actually used at runtime is transparent to the clients. If the queries are stored in several dialects and implementations for all these dialects are available, the underlying database can be switched without the need to recompile or even change any classes. Only the deployment descriptor has to be modified to indicate which version of the query component is to be used. The query component itself has one environment entry per relational adaptor class. This entry contains the fully qualified name of the adaptor class to instantiate. Thus, if a new class is derived from an existing one and consequently a new adaptor becomes necessary, only the environment entry has to be changed. The code can remain as it is. In figure 3.16, the `class` attributes of the two `object` elements identify the names of entries in the query bean's environment that in turn contain fully qualified class names of the adaptor classes.

There is yet another performance issue that has be taken into account when designing the interface for the query bean. If the bean would directly return the query results from a method call, every result object would be created twice. The reason is that all bean communication occurs during the bean's remote interface. As a result, all parameters and return values are passed by value rather than by reference, even if all beans run in the same Java virtual machine. So after the query bean has created all the result objects, they are copied before returning them to the caller which enlarges memory consumption and prolongs overall query time.

The solution is to let an ordinary Java class carry out the query. Since the query component is an entity bean, the natural choice is to let it's content class execute the query.[14] In fact the query bean itself has only the standard interface of all entity beans within the framework. A query's content object can be retrieved using the `getContent()` method of the query bean. The returned object (of a class derived from the auto-generated content class) implements the `Query` interface with the following method:

```
public long executeQuery(
    Hashtable params, Vector skipExpressions, Hashtable results,
    long beginIndex, long numberOfRows
) throws NullPointerException;
```

In `params`, the named parameters for the query are passed. Both keys and values of the Hashtable must be strings. The vector `skipExpressions` contains one string for every expression that should not be included in the query and consequently be replaced according to section 3.6.3.2. The results are returned in the hashtable passed by the caller, which must be an empty `Hashtable` instance when the call is made. On

[14] Note that the query bean is generated from an XML schema the same way that "domain beans" are.

return, it contains one Vector per alias (see figure 3.16) that in turn contains one element per result row. The keys in the hashtable are equal to the aliases. This way, the caller can easily pick the vector containing the objects for a specific alias. With the `beginIndex` and `numberOfRows` parameters, a subset of the resultset can be specified. Only this subset or a smaller one - in case less than `numberOfRows` results are available at the specified `beginIndex` - is returned to the caller (see also next section). The return value of `executeQuery()` is the total number of results, not only the ones in the returned subset. This way, it is easy to create navigable search result pages like the ones used by common internet search engines.

The relational adaptors implement the `RelationalAdaptor` interface. This interface has two important methods:

```
void relationalAdaptorInit(String columnPrefix,
                           java.sql.ResultSetMetaData metaData);

Object processRow(java.sql.ResultSet result);
```

The init method is called before an adaptor is used to construct objects. It allows the adaptor to prepare for processing several rows. For each row, `processRow()` is called. The adaptor has to return the content object for the current row by copying the appropriate columns to the object's fields. Figure 3.17 shows a sequence diagram for executing a query. Note that all content classes by default end in "Data" so that's where the names "QueryData" and "ProductData" come from. The diagram ends when the first result object's construction is finished. After all result objects have been created, they are returned to the caller.

To support queries by example, each content class has a `prepareQuery` method with the following signature:

```
public void prepareQuery(Hashtable params, Vector skipExpressions)
    throws NullPointerException;
```

This method fills the parameters and expressions to be replaced for a query. Both the parameters and expressions are named like the classes fields (e.g. "productName"). An expression is skipped when the corresponding field in the content class instance is `null`. By providing this method, an abstract content class can be used as an example for a query, even if the caller doesn't know anything about the concrete object. This can be further assisted by naming the queries after the content classes' names, maybe with a pre- or postfix (e.g. "MyProductData-query"), so the query to use can be found out by asking a content object for its class name and adding the pre- or postfix. The catalog makes use of this (see section 3.8.3.3). Note that the two parameters must contain valid, but empty `Hashtable` and `Vector` objects that are filled by this method.

3.6.3.5 Browsing query results

For many queries (especially keyword searches and the like) it is necessary to return only a subset of the actual resultset, e.g. results 21 to 40 of a specific search. However, it is generally not feasible to leave a resultset open on the server until the client has finished browsing the results. First of all, there is no way to determine if the client still needs the resultset. The user might just go away and drink some coffee, then he or she gets back 15 minutes later and clicks "next" in the browser. A timeout mechanism would be necessary to determine if a resultset must still be kept open. But what should the timeout value be? Short timeouts annoy the user with error messages after the timeout period is over. In contrast, longer periods lead to significant resource consumption on the server for resultsets that might not ever be needed again.

The solution chosen in this thesis is to query the database every time a new result subset is requested. While this slows down performance when a client really browses a large resultset one page after another, it drastically reduces resource consumption and thus enhances scalability. If a JDBC 2.0 compliant driver is available, the cursor within the resultset can easily be positioned as desired (see [Sunf]). If only a JDBC 1.0 driver can be used, the rows at the beginning of the resultset have to be skipped one after another until the desired start row is reached. Naturally, this slows down retrieval of records that are not located at the beginning of large resultsets. An alternative would be to provide the query with the last result that was already displayed

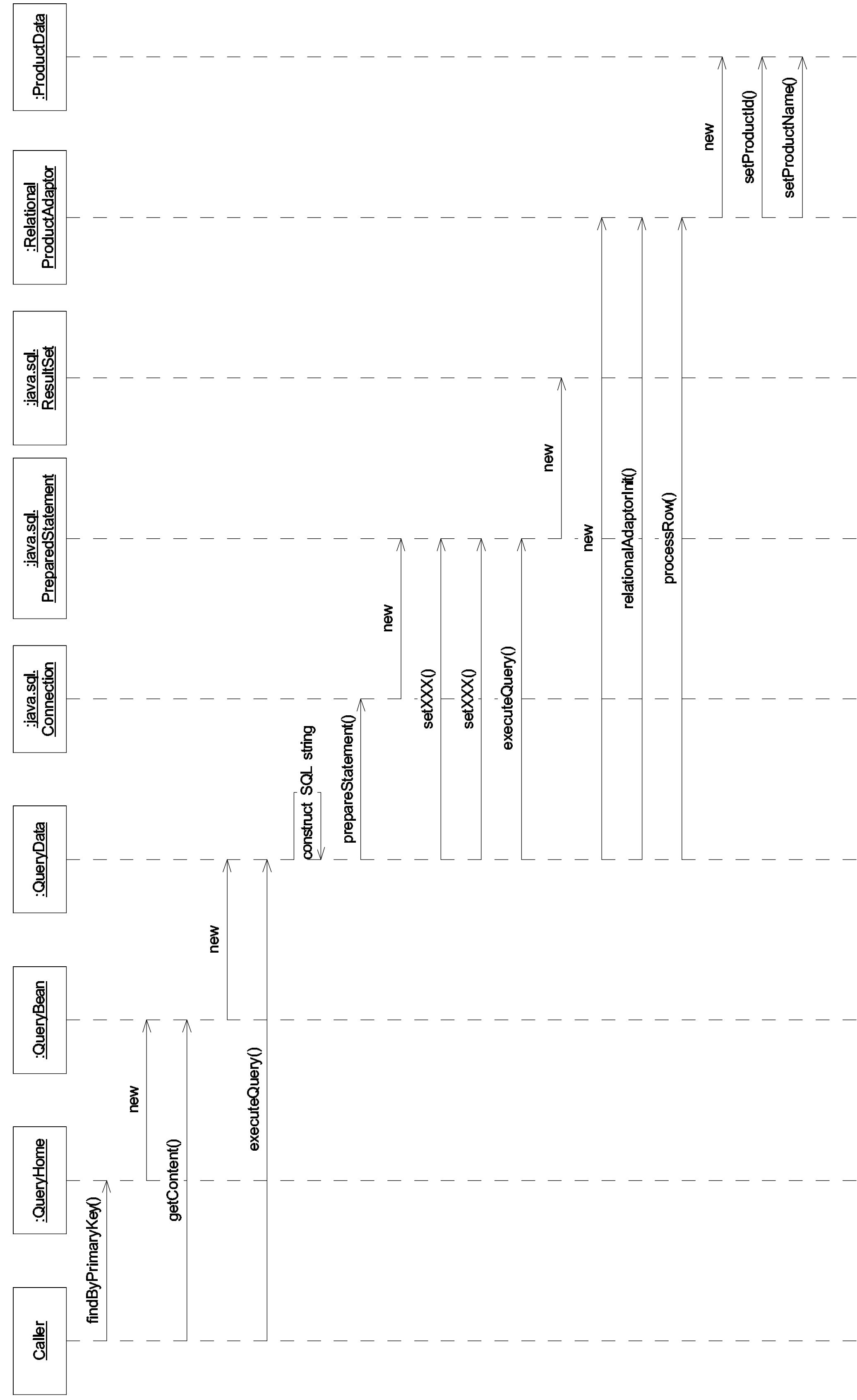


Figure 3.17: Sequence diagram for executing a query

(in form of content objects). This way, only records greater than the last result (in terms of the ordering criteria of the query) can be fetched from the database without the need to skip records at the beginning of the resultset. However, this limits the navigation possibilities to simple next and previous page browsing. Since JDBC 2.0 is already widely adopted, it seems acceptable to rely on resultset navigation. Also, row skipping in the JDBC 1.0 style shouldn't be too expensive when dealing with resultsets of reasonable size.

3.6.3.6 Transactions

In the case of relational databases, all access from the query component is made through a resource manager as recommended by the EJB specification. With container-managed transaction demarcation, the resource managers used by a bean automatically get enlisted with the current transaction context. Thus, database access by the container on behalf of entity beans and direct JDBC access by the query component both fall under the container's automatic transaction management without conflict.

3.6.4 Limitations

The solutions depicted in the previous sections seem appropriate within the scope of this thesis. They have, however, several limitations.

First of all, the granularity of the entity beans is rather fine. For example, the positions of a purchase order are modelled as bean instances, which is exactly the way it should *not* be according to the specification. However, EJB 1.1 with container-managed persistence leaves no other choice if a wide range of containers are to be supported. Also due to EJB 1.1 CMP is the lack of complex attributes (e.g. a vector of strings as an entity bean's field). Without special mapping tools, there is no way no support this.

Another conceptual problem is that primary/foreign key relationships are visible on the application logic level. EJB 1.1 doesn't support automatic conversion to object relationships. This is a complex issue that requires either specialized containers like Persistence PowerTier or mapping tools like TOPLink. The presented query component is a compromise that allows relationships to be hidden in queries. However, there is still no possibility to directly traverse relationships on the object level.

What is also a little disturbing is that one cannot choose an arbitrary polymorphism approach (see [Kel97] for more information). Instead, if you have two subclasses of something (e.g. product) and want the framework to be able to treat them in a polymorphic way, you have to use a common base table - that means you cannot just create two tables where each carries all basic attributes plus the more specific ones, which is a quite common approach. Instead, you have to put the extended attributes in separate tables while keeping the base table. The reason why it has to be that way is that `findByPrimaryKey()` cannot be made polymorphic when using CMP (and no special mapping tool).

As shown in figure 3.9, the framework relies on the type of the primary key of important beans/tables. This is necessary to allow handling of beans and their relationships in a generic way (i.e. being able to create, find, and relate beans without knowing their actual structure). If the primary key type was changed, all parts of the framework that use it would have to be altered too. The primary key should be looked at as on object id rather than a field to place meaningful information in. This approach leads to complications when information from another source than the main database is to be integrated. For example, the products might already be present in an existing SAP installation. They can be used by describing their fields in an XML schema and replacing the generated entity bean with one based on bean-managed persistence. This bean accesses (directly or indirectly) the SAP database to read product data from it. However, if the primary key differs from what the framework expects - which is a long integer - the integration gets more difficult. One solution is to create a map in the framework's database that relates ”framework-compliant” keys to the actual primary keys. The disadvantage is that two database operations are required for actually reading in a product known by primary key. The better way is to insert an additional column in the foreign database (if this is possible) that holds a unique long integer. This column can then be used as a primary key for the framework.

No matter which way is chosen, the queries related to products have to be modified to span the two databases. Taking into account all these complications, it seems easier to import the products (or whatever

data is already present) into the framework's database regularly[15]. Since all fields but the primary key are freely definable, this should be quite simple.

3.7 Security components

3.7.1 Overview

Figure 3.18 shows the dataclasses involved in security checks.

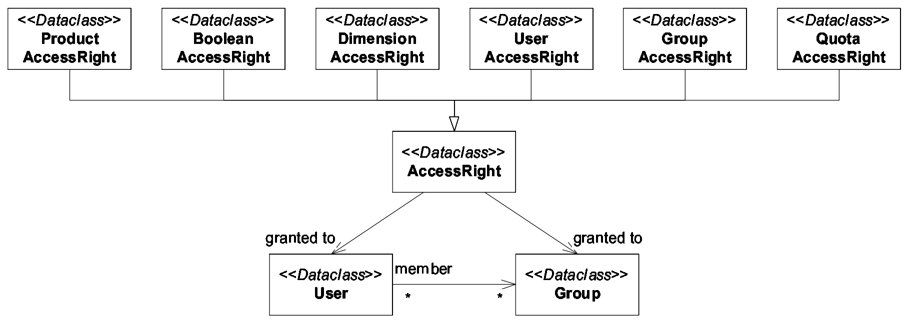


Figure 3.18: Class diagram for security related classes

3.7.2 Authentication, privacy, integrity

In a B2C environment there can be thousands to millions of users that must be managed within an eCommerce application. Only username/password combinations can be reasonably used in this case. It is, however, infeasible to let the webserver manage such masses of login information. Therefore, the B2C case requires a database of all users. Also, users are usually allowed to introduce themselves, i.e. without involvement of the application hoster (see `http://www.amazon.com` for an example).

In the B2B area, things are different. Users can be authenticated in a variety of ways; their number usually doesn't exceed 1000 users; users are usually not allowed to introduce themselves; and there is often already some kind of user management in place (e.g. a directory service). Fortunately, servlets already offer different kinds of authentication, most importantly HTTP basic authentication and SSL client certificates[16]. Consequently, the framework doesn't offer any authentication mechanisms of its own. There is only a "dummy" authentication request that tells the framework which user is to be associated with the current session[17]. Note that, as mentioned in section 3.5.2, the framework isn't exposed directly to outside callers but always through a "filter" (like servlets). However, it may become necessary for specific applications to do "real" authentication within the application server.

Encryption of messages is left to the Secure Socket Layer (SSL) protocol. Since right now the only way to access the framework is via servlets (even in case of automated document exchange), this can safely be left to the web server. SSL also ensures the integrity of all exchanged packets.

3.7.3 Access rights

To support environments where no directory service or similar facility is present, the framework must be able to manage users, user groups, and their access rights completely on its own. However, there must be a way to import user and group information from a directory to ease overall administration of the target IT environment. In this case, only access rights are managed by the framework, while users and groups come

[15] An example for this approach is Intershop enfinity.
[16] Though servlet engines are not required to support certificates.
[17] It is still checked if the user has the right to use the framework, but his or her identity is trusted just as given.

from the directory. The import can, for example, happen twice a day. An online access to the directory would also be possible, but this would complicate the design since access rights checks could no longer be integrated in or expressed by queries (see below). Furthermore, the group memberships of a user would have to be cached on login to avoid directory access on every request.

The actual access rights of a user or group are stored in entity beans. There are several bean classes, one for each kind of right that can be granted. Note that this is a consequence of the user centric approach described in section 2.2.5. In an object centric paradigm, the individual objects (e.g. products) would carry information about which users may access them. If the underlying database is a relational one, this doesn't make a difference on the database level. With both the user centric and the object centric approach, there must be separate tables holding the access rights which contain the id of the user or group and a foreign key that points to access restricted tables.

All access rights beans have a name and associated additional information. For example, a bean instance that grants the right to edit data of a certain product would be of class "ProductAccessRight", carry a name like "ChangeProductRight", and have a reference (in form of a foreign key) to a product.

The following access rights beans are part of the framework (more can be added by individual applications):

BooleanAccessRight An access right which can be either be completely granted or completely denied. If a BooleanAccessRight with a specified name exists, the right is granted; otherwise, it is denied.

ProductAccessRight An access right to certain products identified by a product id. One bean instance stands for one product.

DimensionAccessRight An access right to certain dimensions. One bean instance stands for one dimension key (see section 3.8.3 for what this means and is useful for).

UserAccessRight An access right that refers to users and/or groups of users. One bean instance stands for one user or group.

GroupAccessRight An access right that grants access to user groups. One bean instance stands for one group.

Quota An access right that grants a limited amount of something (e.g. number or value of transactions per month). If no instance with a specified name is present, zero is assumed.

All access rights can be granted to users or groups. Note that there is no such concept as "all users" (e.g. expressed by a `null` value for the user id). This would make the design more complicated and lead to security leaks easier. As an alternative, a group can be created for that purpose and all users put into that group (which can easily be automated by extending the bean responsible for user creation).

The access rights have been designed such that most of them can be applied in queries. This enhances performance and simplifies rights checking. For example, figure 3.19 shows an SQL query for products that takes into account the rights of the current user and the groups he or she is a member of. In the example, a product is only returned if a ProductAccessRight exists that

- is named "ViewProduct" and
- refers to the product in question and
- is either granted to a user and this user is the current user (passed as query parameter) or to a group and the group is one the current user belongs to.

This example is not directly usable for the framework (since extra information is missing, see section 3.6.3), but it shows the principle. Especially for queries with large resultsets - of which the user is allowed to see only a small subset - this can enhance performance quite a bit compared to retrieving all objects and removing forbidden objects later. On the other hand, if there are huge numbers of products (or other access restricted objects) and most users are allowed to access a large subset of them, this approach is a waste of disk space and processing time. In such cases, the access checks should be based on other criteria than simply the product ids (e.g. a status field in the products).

Queries can also be used for ”pure” access checks, i.e. without doing anything else. An example would be a query similar to figure 3.19 where a product id is passed as an additional query parameter and the query only returns the product with this id. If such a query returns nothing, access is denied.

In simple cases, access checks can be based on request types. That means that in a pre-request process graph a boolean access right for the current user with the name of the request is looked up. If the right is not granted, a `ProcessingException` is thrown and the request is not fulfilled. This is the framework's default behaviour.

```
SELECT * FROM Product p
WHERE p.productId IN
   (SELECT pr.productId FROM ProductAccessRight pr
    WHERE pr.name='ViewProduct' AND
       (pr.userId=? OR
          (pr.groupId IN
             (SELECT gm.groupId FROM GroupMember gm
              WHERE gm.userId=?)
          )
       )
   )
```

Figure 3.19: Product query with access rights check

3.7.4 User management

The framework default of a user consists of an artificial id as the primary key and a readable name. The same is true for groups. To store user/group relationships, a bean called `GroupMember` is used. The framework default of this bean only stores a pair of user and group id per instance and doesn't maintain a history (like ”user a was member of group b from ... to ...”).

User id '0' is reserved for the anonymous user. It is necessary to assign such an id (and not simply use `null`) to be able to assign rights to the anonymous user. User id '1' is reserved for the administrator. There is no need to store any rights for the admin, and all components doing access checks must suppress these checks if the current user id equals '1'.[18]

Managing users and groups is done using the auto-generated handler components (described in section 3.6.2.2) for the `User`, `Group`, `GroupMember`, and access rights beans.

3.7.5 User management interface

Figures 3.20 and 3.21 show examples for requests regarding user management. Creation and removal of users and groups is subject to the auto-generated requests explained in section 3.6.2.2. The same is true for granting or revoking access rights since these the rights are also represented by entity beans (figure 3.21 shows an example). Group membership is handled the same way (see figure 3.20), so there are no special requests besides the generated ones.

[18]There may, however, be exceptions to that rule. For example, users may be allowed to store private data that the admin may delete but not view.

As mentioned in section 3.6.2.2, a simple interactive interface is already generated during the persistence transformation. Though this is probably not sufficient, no additional user interface is presented here. This is subject to future work.

```
<add-groupMember-request>
    <userId>123</userId>
    <groupId>123</groupId>

    <!-- This makes user 123 a member of group 123. -->
</add-groupMember-request>
```

Figure 3.20: Making a user member of a user group

```
<delete-productAccessRight-request>
    <productAccessRight-ref>
        <name>PurchaseProduct</name>
        <userId>123</userId>
        <productId>123</productId>
    </productAccessRight-ref>

    <!-- This request revokes the right for user 123 to
     purchase product 123. -->
</delete-productAccessRight-request>
```

Figure 3.21: Revoking an access right from a user

3.8 Domain components

3.8.1 Overview

Figure 3.22 gives an overview over the domain components of the framework. Note that products are not separately explained in this section since the framework only works with their ids. Their actual structure is application dependant.

Only the XML interfaces of the components introduced in this section are mandatory. Anything else is only the default implementation of the framework. All default process graphs consist of only a single component call that parses incoming requests and reacts accordingly. Exceptions to this are explicitly mentioned - an example is the process graph for converting a shopping cart to an order (see section 3.8.5). To change the default behaviour, methods of the components' default implementations can be overridden or a completely different implementation can be supplied (see also section 3.4.4). Furthermore, the default process graphs for individual requests can be replaced by extended or completely new versions.

3.8.2 Internationalization

The most important area concerned regarding internationalization is the catalog of products presented to the user. This aspect is covered in section 3.8.3.4. However, all sorts of other texts may also need to be translated. For this purpose, the framework provides a *translation repository*. Basically, this is just an entity bean named `Translation` that has a string key identifying an expression, a translation of that expression, and the language code specifying which language the translation is in.

By using a string to identify an expression in the translation repository, mono-lingual applications can be developed without any internationalization concerns or overhead. If the switch to a multi-lingual environment is made, the strings that were formerly read directly from the database can then be used as pointers to the

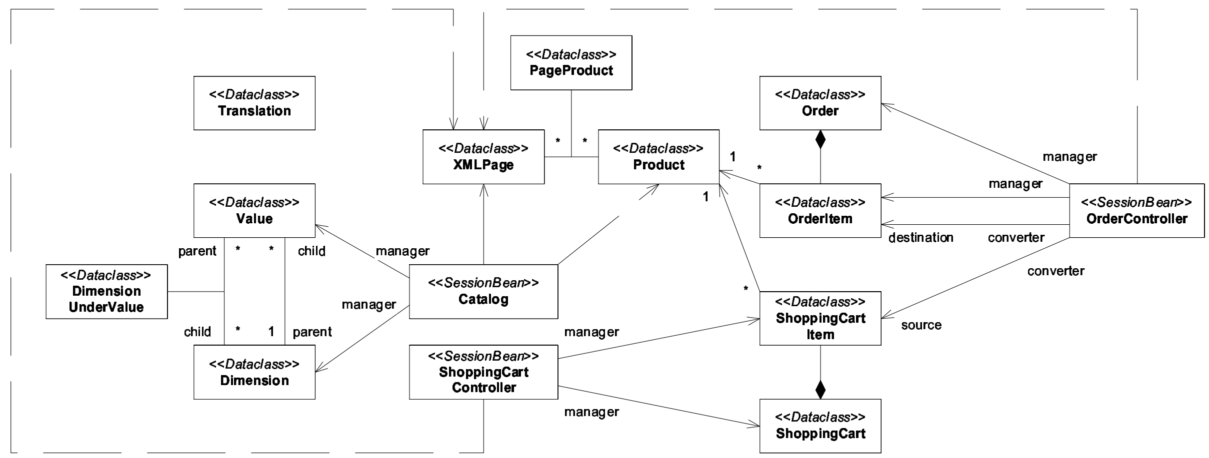


Figure 3.22: Class diagram for domain components

translation repository. This is a fairly common technique that is also used by the company I worked for during my second practical semester (PTV AG, Karlsruhe, Germany). To avoid unnecessary indirections, the standard behaviour of the framework is not to use the translation repository. If desired, this can be changed by adjusting the queries that operate with internationalized data.

3.8.3 The catalog

3.8.3.1 Beans comprising the catalog

All client interaction regarding read access to the catalog is handled by a stateless session bean called `Catalog`. The catalog bean uses several other beans - described in the following sections - that hold information about the catalog hierarchy and pages. The auto-generated handler components for these beans (see section 3.6.2.2) are responsible for write access to catalog information.

3.8.3.2 Dimensions and values

The basic concept of the catalog is that of *dimensions*. A dimension is basically a categorization of products with respect to a certain property. Examples are "suitable for which age" or "minimum processor requirements" that could be applied to computer games. Every dimension has an associated set of *values*. In the examples given these might be "6, 12, 18 and up" and "200 Mhz, 500 Mhz, 1 GHz or above". How a truly multi-dimensional hierarchy looks like is shown in figure 3.23. Under each dimension, the values for this dimension are listed. Under each value, the remaining dimensions appear to allow further restriction of the set of matching products.

A dimension is represented by an entity bean instance with the attributes shown in table 3.4. There is also an entity bean for values as detailed in table 3.5. As root of the catalog hierarchy, the value with id '0' is taken. If necessary due to database referential integrity constraints, it has a `dimensionId` referencing a dummy root dimension. Otherwise, its `dimensionId` is set to `null`.

Every value has a set of dimensions that are its children in the hierarchy. Note that the same dimension can appear under multiple values. Since n:m relationships cannot be handled transparently due to the chosen persistence approach (see section 3.6.1), there is another bean called `DimensionUnderValue` that is responsible for the dimensions that appear under individual values. Each instance references a value by its `valueId` and one or more dimensions by their `key`, which is possible since the keys are not unique. Thus, a single instance relates one value to several dimensions, all with the same key. This is useful in scenarios where different people are allowed access to different parts of the catalog hierarchy. Lets say we specify that key "dim_18andUp" is allowed under the value "age 18 and up". Now we can give a content manager the right to

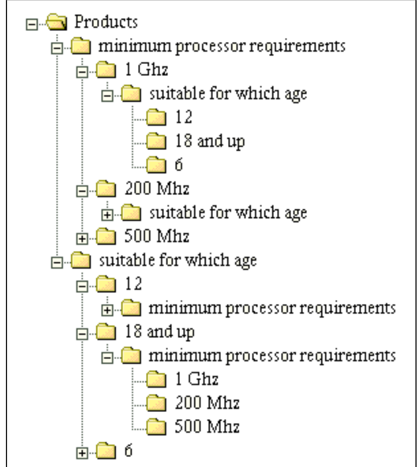


Figure 3.23: Example for a multi-dimensional catalog hierarchy

create and edit dimensions that have this key (using a `DimensionAccessRight`, see section 3.7.3). These dimensions will automatically show up under the value "age 18 and up".[19]

Navigation through the hierarchy is achieved using a *path* of value ids and an optional dimension id. A path always starts with value id '0' (as this is the root) and may reference any node - dimension or value, leaf node or not. If a value is referenced by a path, it consists solely of value ids. The dimensions along the path can easily be retrieved since each value belongs to exactly one dimension. If the path identifies a dimension, it ends with a dimension id. Note that a path is usually acquired from a client request, so it may be illegal in several ways, e.g. an illegal sequence of ids, bogus ids, etc. When implementing the catalog, this has to be taken into account. The easiest way is to simply ignore all possible errors in the path and blindly use it to execute a query. If the query component is implemented in a robust fashion, just an exception has to be caught and an error message returned. It may even be the case that the illegal path doesn't lead to an exception but simply returns a wrong or empty resultset. However, filtering of the results according to the user's access rights takes place anyway, so the "optimistic behaviour" of simply trusting the client's path introduces no security risk.

If a simple hierarchy is desired that ignores the multi-dimensional features, it can be built as follows:

1. For each node in the hierarchy, create a dimension with an arbitrary `name` and `displayName` and a value with a `displayName` that should be used when presenting the catalog to the user. Set `canDisplayResults` to true only in leaf nodes.

2. Make the new value the only value of the new dimension.

3. For every value (including the root value), set `displayDimensions` to false.

4. Construct the hierarchy relationships by making only a single dimension appear under each value. This applies also to the root node.

By doing as proposed above, all dimensions are hidden and only the values carry meaningful information. When using this approach, a consistent naming pattern should be applied, e.g. naming all dimensions "dim_<displayName of only dimension value>".

For such simple cases, finding products for which catalog pages must be returned can be implemented very simply. It's enough to introduce a product field that contains the value id the product should belong to. If a path identifying a value is received from the client, a query is executed that simply finds all products with

[19] Of course this example also requires some access rights management regarding the association of products with dimension values (which is not covered in this thesis); however, the example shows the general usefulness of the key attribute.

a value id identical to the last id in the path. This works because value ids are unique accross all dimensions and, in this scenario, only leaves are associated with products.

When making use of multiple dimensions, things are not that easy. There can be several paths that should return the same resultset (see figure 3.23) but end in different value ids. To return correct results, the whole path has to be examined. According to what is found, a set of query parameters has to be constructed and fed to a query that uses one parameter per dimension. It may also be necessary to introduce a new field to the dimension bean that identifies the parameter key used in the part of the query that relates to a certain dimension. The value bean must probably be given an additional field too that contains the parameter value to use in a query - the id and display name might not be appropriate for looking up the products. Furthermore, a way must be found to categorize products in a manner usable for a multi-dimensional catalogue.

In summary, the catalog in its current stage of design is prepared for multiple dimensions, but additional work has to be done to really make use of this functionality. The important point is that a simple hierarchy can be built now and used in future versions without changes. Also, full support for truly multi-dimensional hierarchies doesn't require severe implementation and interface changes. In fact, the interface of the catalog shouldn't have to change at all.

dimensionId (`Long`)	An artificial primary key.
key (`String`)	An arbitrary string for referencing the dimension.
displayName (`String`)	A string to display for this dimension (e.g. ”by color”).

Table 3.4: Fields of the `Dimension` bean.

valueId (`Long`)	An artificial primary key. This id is unique across values for all dimensions, enabling unique identification of a certain value.
dimensionId (`Long`)	The id of the dimension this value is attributed to.
displayName (`String`)	A string to display for this value, e.g. ”age 18 and up”.
displayDimensions (`Boolean`)	Displaying dimensions (e.g. ”Games/by age/age 18 and up”) can be suppressed for the dimensions under individual values, so only values are displayed (e.g. ”Games/age 18 and up”). This is useful if only one dimension is displayed under a value (e.g. when constructing simple hierarchies).
canDisplayResults (`Boolean`)	Specifies whether this value can be used to show a list of catalog pages. In simple hierarchies, this property is set to true only for leaf nodes.

Table 3.5: Fields of the `Value` bean.

3.8.3.3 Queries

By default, the catalog only supports a query by example for products. How corresponding XML requests look like is shown in section 3.8.3.5. A keyword search is not yet part of the framework, but SQL LIKE patterns can be used for string fields.

To enable the catalog bean to work independently from the actual product structure (which fields of which types) used in an application, the catalog never works with product fields itself. To see how this is achieved when searching the catalog, take a look at the example request in figure 3.26 on page 83. The query used to actually execute the search has a WHERE clause with comparison expressions for every product field. Each comparison expression is surrounded by an `expression` element as shown in figure 3.16 on page 68. The problem is that the catalog bean cannot tell which expressions to skip based on the information given

in the request. Instead, the content class responsible for products is employed for this task. However, the name of this class is probably different in every application, so the catalog bean deducts it from the search element used in the request. In this example, the element is called ”myProduct-search”, so the corresponding content classes name must be something like ”MyProductData” (depending on the naming rules used in the persistence transformation). Alternatively, the class name could be stored in the catalog bean’s environment.

Now that the right content class is known, it can be instantiated using `Class.forName(...).newInstance()`. When parsing the request, the catalog puts all elements found under the search element into the request context or a newly created instance of `Context`. Then it calls `assignFromContext()` on the content class instance, thus populating it with the data specified in the search request. Finally, the content object’s `prepareQuery()` method is used to get the necessary parameters and expression information for the query which can then be executed.

While the described strategy may seem complicated, it works very well. Regardless of what a product looks like in an application using the framework, the catalog bean’s code does not have to be changed. This example also shows how contexts and content classes can be used to shift around information without depending on specific classes. More information on that can be found in section 3.6.

3.8.3.4 Catalog pages

The heart of every catalog are the pages describing the products. In this design, there is no tight coupling between products and catalog pages. Instead, a product can be described on several pages and one page can describe several products (e.g. the license key for a software product and the manuals for it). The catalog pages are written in XML and saved in the database together with some additional information, all packed together in an `XMLPage` entity bean with the fields shown in table 3.6. A catalog page is associated with one or more products via the `PageProduct` entity bean with the fields shown in table 3.7. Each product is referenced in a catalog page using the alias specified in the corresponding `PageProduct` instance.

Figure 3.24 shows an example of a catalog page. Each page can contain texts in several languages. In environments with many languages and/or large catalog pages it may become necessary to split the texts into separate bean instances. However, this introduces an additional indirection that is usually not necessary. Furthermore, the page texts are rather small in most cases since they don’t (or at least should not) contain any formatting specifics.

Catalog pages are processed by the `Catalog` bean - plus additional logic within the process graphs regarding catalog requests - before being returned to the client. Most importantly, data from the products associated with the page is inserted. To enable this, all products referenced by the page are loaded using the query component (see section 3.6.3) before processing starts. Their fields are copied to the request context via the `copyToContext()` method (see section 3.6.2.3), using the product alias described above as a prefix.

Information from the products can be inserted during page processing using the `insert` tag. It refers to an arbitrary variable in the request context. Most of the time, this will be a product field. The optional `format` attribute does not really do any formatting but is used in special cases to control output of product fields. In this example, the price is output with decimal separators according to the language given in the `page-text` element and rounded to 2 fractional digits.[20] `link` elements are basically copied during page processing since only the presentation layer can replace them with whatever elements are needed to present the end user with a link. The `type` attribute gives hints as to what should be addressed by this link. A ”cart” link should be replaced by the presentation layer with a link that puts a product into the shopping cart. The `dest` attribute in this type of link contains a product alias. This alias is replaced by the catalog bean with the id of the product. A ”catalog” link points to another catalog page. The `dest` attribute contains the key of the referenced page. Finally, an ”image” link points to an image (which is just a simple bean with an artificial id, a unique string as a readable key, a MIME type, and a field containing binary data). Other types of binary data are not yet part of the framework but can be added easily. As an alternative, images and other data can also be stored on the web server instead of the framework’s database.

[20]Which set of formats should be supported has not yet been decided. This is subject to future work.

Using aliases and keys, respectively, to reference the link destinations is mainly done to allow writing catalog pages without knowing these ids. Note that the keys specified in catalog and image links are not resolved to ids during processing. The main reason is that unnecessary queries for these ids can be avoided when returning several catalog pages to the client[21]. To support using keys instead of ids, the catalog accepts page requests by key instead of id. For images or other binary data, auto-generated search requests as described in section 3.6.2.2 can be used where only the key is specified. This means that the presentation layer must create such a search request when the user's browser asks for an image.

Other types of links can be added by specific applications based on the framework. For example, a kind of link that has a query behind it could be useful. This way, links like "click here for related products" could be realized without updating the page every time a new related product is introduced. Of course it's also possible to replace the set of tags used by the catalog bean altogether by providing an alternative bean implementation. This way, existing XML DTDs/schemas can be used in the catalog.

In automated scenarios, the automatically generated requests for the product bean (described in 3.6.2.2) should be used for product lookup rather than the catalog.

```
<?xml version="1.0"?>
<catalog-page xmlns:cat="http://www.isb-ka.de/ecom/catalog">
    <page-text lang="en">
        Superb product ... Costs only
        <price>
            <cat:insert variable="prod1.price"
             format="currency"/>
        </price>
        ...
        <cat:link type="cart" dest="prod1"/>
          <!-- 'prod1' is a product alias used in this page -->
        <cat:link type="catalog" dest="someOtherPage"/>
          <!-- 'someOtherPage' is the key of another catalog page -->
           Click here for more info.
        </cat:link>
        <cat:link type="image" dest="somePicture"/>
          <!-- 'somePicture' is the key of an image -->
           Picture of nice product
        </cat:link>
    </page-text>
    <!-- texts in more languages could be added here -->
</catalog-page>
```

Figure 3.24: Example for a catalog page

pageId (`Long`)	An artificial primary key.
key (`String`)	An arbitrary unique string that can be used for referencing pages.
canBeSearchResult (`Boolean`)	Specifying whether the page can be shown in a list of search results. This is used to realize the short/long functionality described in section 2.2.4.
pageText (`String`)	The actual page in an XML based format.

Table 3.6: Fields of the `XMLPage` bean.

[21] In the case of cart links, this is not an issue since the products related to a page must be loaded anyway to include information from them in the page.

pageId (Long)	The id of the catalog page that is concerned.
productId (Long)	The id of a product that should be associated with a page.
productAlias (String)	Used to reference this product in the catalog page.

Table 3.7: Fields of the PageProduct bean.

3.8.3.5 Interface of the catalog bean

Figures 3.25 to 3.29 show typical XML requests and responses that are handled by the catalog bean. The request in figure 3.28 shows how a catalog hierarchy of dimensions and values is requested. The given path ends with a value, so what should be returned are the dimensions under this value (see section 3.8.3.2). The return-whole-tree attribute tells the catalog to return a whole catalog tree, but with only the given path expanded (i.e. only including the children of dimensions/values on the path). Figure 3.29 is the response to that request. The top node returned is the root node. Under the root, a value appears, so it can be concluded that displayDimensions is set to false for the root. Next in line is a dimension, so displayDimensions must be true in the parent value. Note that this dimension has not been specified on the path (according to section 3.8.3.2). The following value is the last entry on the path, so its children are the ”real” answer to the request. Only they would have been returned if return-whole-tree was not specified in the request. One would expect that dimensions are returned here - since the path in the request ended with a value - but in this example it is assumed that displayDimensions is false for value ’45’, so the children *values* of all dimensions under ’45’ are returned. These values can be used to actually ask the query bean for catalog pages, which is indicated by the canDisplayResults attribute.

Using return-whole-tree, it is easy to build a servlet or XSLT stylesheet that displays an HTML catalog tree with one branch open at a time. More sophisticated navigation requires additional request types for the catalog that are not yet part of the framework. However, it is easy to add a request that returns *really* the whole tree (with all branches open). This could be used in a Java applet to provide better navigation possibilities.

Figure 3.25 shows a request for the catalog pages that match a certain dimension value, in figure 3.26, a query by example is made. Note that the query is about products, but catalog pages are returned. Finally, figure 3.27 shows a response containing catalog pages.

Currently, there is no GUI design for editing the catalog hierarchy or pages except the auto-generated forms described in section 3.6.2.2. This is subject to future work.

```
<get-catalog-pages-request>
   <path>
      <value id="0"/>
      <value id="23"/>
      <value id="45"/>
      <value id="55"/>
   </path>
</get-catalog-pages-request>
```

Figure 3.25: Request for catalog pages from the hierarchy

```
<search-catalog-request beginIndex="0" numberOfResults="20">
   <myProduct-search>
      <productName>%Studio</productName>
      <productDescription>%Software%</productDescription>
   </myProduct-search>
</search-catalog-request>

<!-- This request returns a "get-catalog-pages-response". -->
```

Figure 3.26: Query by example for products

```
<get-catalog-pages-response beginIndex="0" numberOfResults="20"
 totalResultCount="47">
   <page-text lang="en">
      Superb product ... Costs only
      <price>15.99</price>
      <cat:link type="cart" dest="123"/>
      <cat:link type="catalog" dest="someOtherPage">
         Click here for more info.
      </cat:link>
      <cat:link type="image" dest="somePicture">
         Picture of nice product
      </cat:link>
   </page-text>
   <page-text lang="en">
      ...
   </page-text>
</get-catalog-pages-response>
```

Figure 3.27: Response containing catalog pages

```
<get-catalog-hierarchy-request return-whole-tree="return-whole-tree">
   <path>
      <value id="0"/>
      <value id="23"/>
      <value id="45"/>
   </path>
</get-catalog-hierarchy-request>
```

Figure 3.28: Request for a catalog hierarchy

```
<get-catalog-hierarchy-response>
   <value id="0" displayName="">
      <value id="23" displayName="Computer games">
         <dimension id="15" displayName="by age">
            <value id="45" displayName="6">
               <value id="55" displayName="Math learning games"
                canDisplayResults="canDisplayResults"/>
               <value id="56" displayName="Language learning games"
                canDisplayResults="canDisplayResults"/>
            </value>
         </dimension>
         <dimension ... />
           <!-- This dimension is returned without children. -->
         ...
      </value>
      <value ... />
        <!-- This value is returned without children. -->
     ...
   </value>
</get-catalog-hierarchy-response>
```

Figure 3.29: Response to figure 3.28

3.8.3.6 Limitations

With the dimension approach as explained above, it's not easily possible to say "every dimension can appear under value xyz (except the ones that lie on the path to that value)". This functionality could be implemented in an additional layer that uses `DimensionUnderValue` as a mere index and provides a higher level of abstraction. However, such a layer is not part of this thesis.

3.8.4 The shopping cart

3.8.4.1 Beans comprising the cart

The default shopping cart of the framework only provides limited functionality. Products can be added to the cart; the cart's contents can be manipulated by changing the quantity of or deleting contained items; and the cart's contents can be ordered (which is the responsibility of the order controller, see section 3.8.5).

Technically, the cart consists of two entity beans and a stateless session bean. The session bean is called `ShoppingCartController` and is responsible for parsing XML requests regarding the shopping cart. The `ShoppingCart` entity bean represents a whole cart. It contains the fields shown in table 3.8. The session id is not used as the primary key to enable multiple carts per session. However, the current design only uses one cart per session. As explained in section 3.6.1, only simple attributes are possible with the chosen persistence approach. Therefore, the items contained in the cart are managed using a `ShoppingCartItem` entity bean. This bean has the fields shown in table 3.9. Note that different quantity units are not supported by the default framework implementation.

The framework itself does not currently include any support for customizing products. If customization is desired, the shopping cart beans can be changed in two ways to support this: Either fields for customizations are added to the `ShoppingCartItem` bean or a reference to an additional customization bean is introduced. Furthermore, a new type of link must be introduced for catalog pages that allow customization (see section 3.8.3.4). This link can be converted to a submit button when using HTML as the user interface. Future versions of the framework might introduce a default customization facility.

To specify what fields should be returned when the cart's content is requested, XML pages are used (see section 3.8.3.4). They are stored as `XMLPage` instances with `canBeSearchResult` set to false. There is one page for the cart's general data (the *master*) and one for a single item (a *detail*). The information referenced in `cat:insert` elements is taken from the request context. It is put there by the shopping cart controller which uses a query (the id of which is specified in the controller's environment) to obtain it. The field prefixes in the request context are the same as the aliases used in the query (see section 3.6.3).

Field	Description
cartId (`Long`)	An artificial primary key.
sessionId (`String`)	A string identifying the session this cart belongs to.
dataCreated (`Date`)	Specifying when this cart was first used.
lastUsedDate (`Date`)	Specifying when this cart has last been queried or manipulated.

Table 3.8: Fields of the `ShoppingCart` bean.

Field	Description
cartItemId (`Long`)	An artificial primary key.
cartId (`Long`)	The id of the shopping cart this item belongs to.
productId (`Long`)	The product referenced by this cart item.
quantity (`Double`)	Specifying the quantity of the product referenced by this item.

Table 3.9: Fields of the `ShoppingCartItem` bean.

3.8.4.2 Interface of the shopping cart

Figures 3.30 to 3.32 show examples for requests and responses accepted and returned by the shopping cart controller.

```
<put-item-into-cart-request>
   <productId>1234</productId>
   <quantity>1</quantity>
</put-item-into-cart-request>

<!-- This request returns a display-cart-contents-response. -->
```

Figure 3.30: Request for putting an item into the shopping cart

```
<display-cart-contents-request>
</display-cart-contenst-request>
```

Figure 3.31: Request for displaying the shopping cart's contents

```
<display-cart-contents-response>
   <head>
      <page-text lang="en">
         <cartId>12</cartId>
         ...
      </page-text>
   </head>
   <detail>
      <page-text lang="en">
         <cartItemId>123</cartItemId>
         ... some product info like name, etc. ...
      </page-text>
      <page-text lang="en">
         ...
      </page-text>
   </detail>
</display-cart-contents-response>
```

Figure 3.32: Response containing the cart's contents

3.8.5 Orders

3.8.5.1 Beans related to orders

Orders are similar to shopping carts. There is a controller bean called `OrderController`, a "head" bean called `Order` and an `OrderItem` bean that manages individual items. An alternative would be to use the same beans for both orders and shopping carts, since they basically contain the same information. However, it seems conceptionally cleaner to separate them from each other. This approach also allows for more flexibility regarding support for existing or coming B2B standards, customization, etc. The fields of the `Order` and `OrderItem` beans are shown in tables 3.10 and 3.11.

The process graph in figure 3.36 shows that shopping cart items are copied to order items via the request context. This loose coupling allows for changing the fields of the items without the need to change any hand-written code. Only the schema and the process graph for ordering a cart have to be changed. It's also possible to add fields to shopping cart items that are not present in order items. In this case, the additional fields are ignored when a copy is made. On the other hand, order items can contain fields not present in cart items that are set to appropriate values in the process graph.

The order controller is responsible for confirmation or cancellation, respectively, of a preliminary order. Also cancellation of a confirmed order would belong to its tasks but is not supported by the default implementation. There is also no default support for shipping and billing addresses, ways of payment, etc. For example, in the pilot application there is no need for this data. Shipping and billing addresses are taken from SAP where the logical connection is made by a debitor number that is stored as part of a user's data. The only way of payment in this application is by invoice. Future versions of the framework might introduce default support for addresses and/or payment (possibly including virtual money).

For displaying orders, the same approach as for the shopping cart is used, i.e. XML pages for head and detail data. See section 3.8.4.1 for further information.

orderId (`Long`)	An artificial primary key.
status (`String`)	One of "preliminary" or "confirmed" (see section 2.2.4). Every order is first created with a status of "preliminary".
dateCreated (`Date`)	The date and time the order was created.
dateConfirmed (`Date`)	The date the order was confirmed. This field is only valid for confirmed orders.
userId (`Long`)	Identifies the user that made the order.

Table 3.10: Fields of the `Order` bean.

orderItemId (`Long`)	An artificial primary key.
orderId (`Long`)	The id of the order this item belongs to.
productId (`Long`)	The product referenced by this order item.
quantity (`Double`)	Specifying the quantity of the product referenced by this item.

Table 3.11: Fields of the `OrderItem` bean.

3.8.5.2 Interface of the order controller bean

Figures 3.33 to 3.35 show sample requests and responses accepted and returned by the order controller. Figure 3.36 shows the process graph that is traversed when an `order-cart-request` is received. All other requests are handled by process graphs containing a single `component-call` element.

```
<order-cart-request>
   <cartId>123</cartId>
</order-cart-request>

<!-- This request is answered by a "display-order-contents-response". -->
```

Figure 3.33: Request for ordering a shopping cart

```
<search-order-request beginIndex="0" numberOfResults="20">
   <defaultOrder-search>
      <status>Confirmed</status>
      <dateConfirmed-min>2000-08-20</dateConfirmed-min>
        <!-- return all orders made since 20th of August 2000 ->
   </defaultOrder-search>
</search-catalog-request>

<!-- This request returns a "display-order-contents-response". -->
```

Figure 3.34: Request for searching confirmed orders

```
<display-order-contents-response>
   <order>
      <head>
         <page-text lang="en">
            <orderId>12</orderId>
            ...
         </page-text>
      </head>
      <detail>
         <productId>123</productId>
         <productName>someProduct</productName>
         ...
      </detail>
   </order>

   <!-- More order elements appear here if the request was
    a search request. -->
</display-order-contents-response>
```

Figure 3.35: Response containing the contents of one or more orders

```
<process-graph name="de.isbka.ecom.framework.pg.GraphForOrderRequest">
   <components>
      <component type="enterprise-bean" alias="orderController"
       name="de.isbka.ecom.framework.OrderController"/>
   </components>

   <graph>
      <component-call alias="orderController" method="parseRequest"/>
      <loop>
         <condition>
            <get-var name="OrderController.hasMoreItems" type="bool"/>
               <!-- This has been set by the previous component-call. -->
         </condition>
         <body>
            <copy-var source="CartItem.productId" dest="OrderItem.productId"/>
            <copy-var source="CartItem.quantity" dest="OrderItem.quantity"/>

            <!-- At this stage, some further processing could occur. -->

            <component-call alias="orderController" method="addOrderItem"/>
               <!-- This method also sets "OrderController.hasMoreItems" and
                puts the next cart item into the context. -->
         </body>
      </loop>
   </graph>
</process-graph>
```

Figure 3.36: Process graph for ordering a shopping cart

3.9 Presentation components

The presentation paradigm of the framework mostly follows the model-view-controller pattern (described for example in [GHJV95]). The enterprise beans are responsible for modelling information that is visualized by means of the presentation layer. The latter is also responsible for accepting user input and sending appropriate information to the application layer. For example, if the user clicks on a link or on a submit button, the presentation layer must generate the appropriate XML requests and send them to the framework.

Formatting tasks can be fulfilled by the presentation layer using XSLT stylesheets. If needed, some or all of the transformations can be shifted from stylesheets to servlets, thus increasing performance. However, manageability of the presentation layer suffers from that. Additionally, some tasks can be delegated to applets (e.g. displaying a catalog tree). If necessary or desirable, the whole user interface can also be presented by an application (Java or other). This flexibility is possible by interacting with the application layer solely by exchanging XML documents (plus arbitrary additional data if necessary).

To see how the presentation layer works, take a look at figure 3.27 on page 83. This is the response to a search request (or some other request that returns a number of catalog pages). The presentation layer produces a result looking like figure 3.37 with the HTML source shown in figure 3.39 (simplified example). The stylesheet used to produce this output can be found in section C.2. It automatically converts the link elements to HTML anchor tags and puts out a default text for the cart link (if none is supplied in the catalog page). In this example, a servlet called "gateway" is responsible for generating XML requests when the user clicks a link. Note that formatting of error messages can be handled by the same stylesheet that transforms the catalog pages. In fact, a single stylesheet - possibly incorporating additional styleheets using XSLT's `<include>` facility - can do all the formatting work. Of course also a combination of HTML and CSS can be used as output to the client. A good introduction to XSLT can be found in [Har00].

In case of the auto-generated requests for persistent objects (described in section 3.6.2.2), simple user interface components are also generated that are based on forms like the one shown in figure 3.38. To be really useful, these components should be modified with respect to design and functionality. For example, references to other persistent objects should be selectable via drop down lists. Note that by changing the XSLT stylesheets used to generate the presentation components, the interface for managing *all* persistent objects can be changed. The form field labels are taken from the source XML schema using one `gen:displayName` element per field. These elements simply contain a string.

In the current design, the forms are "hard-coded" into the generated presentation components, including the strings specified in `gen:displayName`. In future versions this should be changed in order to better support multi-lingual environments. The forms should (structurally) be generated on the application layer rather than the presentation layer with the field labels being looked up in the translation repository (see section 3.8.2). However, these enhancements are not really essential since all auto-generated forms only serve simple management purposes. Only managers or administrators of the application should ever see them.

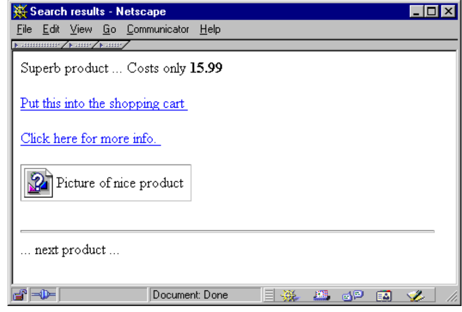


Figure 3.37: Example for a catalog page

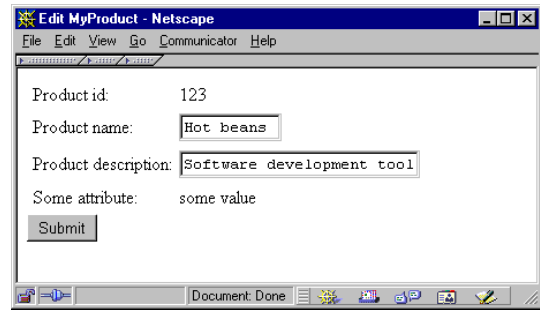


Figure 3.38: Example for an auto-generated form

```
<html>
   <head>
      <title>Search results</title>
   </head>
   <body>
      Superb product ... Costs only
      <b>15.99</b>
      <p>
         <a href="/servlet/gateway?command=cartLink&dest=123">
            Put this into the shopping cart
         </a>
      </p>
      <p>
         <a href="/servlet/gateway?command=catalogLink&dest=someOtherPage">
            Click here for more info.
         </a>
      </p>
      <p>
         <img src="/servlet/gateway?command=imageLink&dest=somePicture"
          alt="Picture of nice product"/>
      </p>
      <hr />
      ... next product ...
   </body>
</html>
```

Figure 3.39: HTML source of figure 3.37

Only the basic working principles of the presentation layer as depicted above have been designed as part of this thesis. The actual user interface design is strongly application dependent and should be done together with specialized design agencies. Future versions of the framework should provide a default user interface.

3.10 Glossary of terms

This glossary explains special terms that are used throughout chapter 3. All terms explained here are ”self made” and not generally used - at least not in the way they are used here.

Application A concrete application that uses the framework presented in this thesis as its foundation.

Catalog dimension Short: dimension. A categorization of products with respect to a certain aspect, e.g. minimum processor requirements of software products. Catalog dimensions are described in section 3.8.3.

Catalog dimension value Short: dimension value or just value. A concrete value within a *catalog dimension*, e.g. ”500 Mhz”.

Content class A class used for transporting an entity bean’s fields to or from a client with a single RMI call. Content classes also offer additional services for use with *process graphs* and queries. They are generated during the *persistence transformation*.

Content object An instance of a content class.

Context A set of key/value pairs plus optional additional information. A context has a certain scope (e.g. one client request). Contexts are explained in section 3.4.2.

Dataclass — A complex type in an XML schema that is converted to an entity beans plus additional files during the *persistence transformation*.

Extension schema — An XML schema generated during the *persistence transformation* that includes the definition of requests and responses for *dataclasses*.

Handler component — An auto-generated stateless session bean that processes XML requests that deal with persistent objects. Handler components are explained in section 3.6.2.2.

Persistence transformation — The generation of entity beans and other persistence related files from an XML schema using XSLT. The persistence transformation is explained in section 3.6.2.

Process graph — A directed graph that is traversed to process a user request. Process graphs are represented by XML documents. They are explained in section 3.4.3.

Request context — A *context* with a scope of a single request. Additionally, it provides access to the current *session context*.

Request variable — A key/value pair contained in a *request context*.

Session context — A *context* with a scope of a session (i.e. a range of connected requests).

Session variable — A key/value pair contained in a *session context*.

Shopping cart page — Short: cart page. An XML document describing the information to include when displaying the shopping cart's contents to an interactive user. There are cart pages for the head and detail sections of a shopping cart. Cart pages are described in section 3.8.4.

Order page — An XML document describing the information to include when displaying an order and order item information to an interactive user. There are order pages for the head and detail sections of an order. Order pages are described in section 3.8.5.

Chapter 4

Iterative development of the most important components

4.1 Tools used

The following software has been used during development:

- Sun's JDK 1.2.2. The performance was good enough during development, though for production purposes, the HotSpot virtual machine (http://jsp2.java.sun.com/products/hotspot/) is probably a better choice.
- JOnAS from Bull Information Systems (http://www.bullsoft.com/ejb/). JOnAS stands for "Java Open Application Server" and is an open source, EJB 1.1 compliant application server. It may however be used for commercial purposes. There are also other open source alternatives such as OpenEJB (http://www.openejb.org) or jBoss (http://www.jboss.org/) that have not been evaluated since JOnAS was sufficient for development purposes.
- Oracle 8i (Version 8.1.5).
- Xerces-J from the Apache XML project (http://xml.apache.org/xerces-j/index.html). Xerces is an open source XML parser written entirely in Java. While it's not one of the fastest parsers, it's probably one of the most complete ones, including good XML schema support.
- Xalan-J from the Apache XML project (http://xml.apache.org/xalan/index.html). This XSLT engine is also open source and shares with Xerces its completeness and its relatively poor performance (compared to XT from James Clark or other alternatives). However, it turned out to be fast enough for development and has a very active developer community. Xalan-J is a pure Java implementation.
- Tomcat from the Apache Jakarta project (http://jakarta.apache.org/). Since this servlet engine is the official reference implementation, one should expect it to be complete and fully conformant. It also works well with the Apache HTTP server.
- The Apache HTTP server (http://www.apache.org/httpd.html). This open source web server runs on several platforms and is known to be robust and fast.
- Ant from the Apache Jakarta project (http://jakarta.apache.org/ant/index.html). Starting with simple batch files, I soon turned to Ant as a build tool. It's a complete Java solution that takes XML files as input to describe the build process. Ant is platform independent and easy to use. Also, extensions can easily be written in Java and "plugged in".

The machine used for development had an 600Mhz AMD Athlon processor and 512 MB RAM. The operating system was Windows NT 4.0 SP5. The database was running on separate server under Linux.

4.2 Overview

The focus for development lay on important and possibly critical parts of the framework. Since the persistence transformation is the basis for all other functionality - including persistence of the session context when processing a client request - it has been first in line. Section 4.4 shortly describes the prototypical implementation that has been realized within the time frame of this thesis. A significant problem that occurred during development of this part is - together with the chosen solution - depicted in section 4.3. Besides that, also a rudimentary version of the request dispatcher was developed together with a simple test servlet. As far as process graphs are concerned, a "proof of concept" stylesheet was written that shows the feasibility of this approach. Together with the experience collected with the stylesheets performing the persistence transformation, this is a solid basis for the final realization.

Unfortunately, there was not enough time to do any other implementation work. Especially the query component might impose problems during implementation and is crucial for many parts of the framework. Most read access to the underlying database is designed to use the query component, including access checks. Thus, the query bean is the next part that should be developed. Furthermore, none of the domain components could be realized, but they don't appear to be critical in any way. The same applies to the presentation layer where the main effort consists of writing simple stylesheets[1] and constructing relatively small XML documents from HTTP request parameters.

4.3 Developing a framework with Enterprise JavaBeans

While implementing the design presented in the previous chapter, I came across a fundamental problem related to the framework approach and the EJB architecture. To illustrate this, take the following example: The framework uses an abstract interface to interact with products. It also provides a default implementation of that interface. Users of the framework can both use their own implementation as well as extend the interface for their purposes. For instance, a developer might extend the interface in a way that a product can be queried for tax rates depending on where the customer lives. For that purpose, he or she derives an own remote interface from the one provided by the framework. According to the EJB spec, the developer now also has to provide a new home interface. The reason is that every bean must provide a `findByPrimaryKey()` method in its home interface with the Bean's remote interface as return type. Since the remote interface has changed and the home interface provided by the framework does not return this new interface from `findByPrimaryKey()`, a new home interface is necessary.

Unfortunately, the new home interface cannot be derived from the old one. Now why is that? The new `findByPrimaryKey()` method must replace the old one. In terms of Java interface inheritance, the new method must *override* the old one. The problem is that overriding only works when the return types of the two methods are the same, which is not the case (remember that the new `findByPrimaryKey()` must return the new remote interface). Now we have two home interfaces with no inheritance relationship. In the framework's code, the old interface type is used, while the new product bean has no choice but specify the new one in its deployment descriptor. This makes the product Bean inaccessible to the framework![2]

The specification relies on the means of the Java language when it comes to inheritance. This enables both *implementation inheritance* (i.e. deriving a Bean's implementation from some other class) and *interface inheritance* (e.g. deriving the remote interface from some superinterface). What was described in the previous paragraphs is referred to in the EJB spec as *component inheritance*. This type of inheritance is not supported by the spec, but it is vital for frameworks! No framework works without a way to extend its parts while staying compatible to the original interfaces.

I've chosen a solution that uses the *adaptor* pattern (see [GHJV95]). If a Bean wants to access another Bean, normally the following code is used:

[1] Simple because the transformations go from XML to (X)HTML which is by far easier than generating text output (which is necessary for the persistence transformation).

[2] At least when the framework makes use of the home interface. It is, however, neither practical nor reasonable to avoid working with the home interface. For example, in order to edit a product's fields, the framework must be able to find it by primary key.

```
BeanHome beanHome = (BeanHome) javax.rmi.PortableRemoteObject.narrow(
    namingContext.lookup("ejb/Bean"), BeanHome.class);
```

In contrast, the framework accesses every Bean like this:

```
Object beanAdaptor =
    Class.forName((String) namingContext.lookup("BeanAdaptor")).newInstance();
((HomeAdaptor) beanAdaptor).init(namingContext, "ejb/Bean");
BeanHome beanHome = (BeanHome) beanAdaptor;
```

The adaptor class implements the home interface that is expected by the calling bean and forwards calls to the ”real” interface. In cases where there is no extension (yet), a dummy adaptor is used that ”adapts” a home interface to itself. The remote interface is not affected by the adaptor class, since it doesn't have the `findByPrimaryKey()` problem. Extensions can be made by simply inheriting from the original remote interface, thus staying compatible with the base interface.

If a developer wants to extend one of the framework's beans, he or she writes the remote and home interfaces plus the additional implementation code and the adaptor class. All but the adaptor class is necessary anyway, and this class is trivial to implement. In case of auto-generated entity beans (see 3.6.2), e.g. when simply adding a field to the default implementation, the appropriate adaptor class is auto-generated.

Which adaptor class is actually used can be determined by modifying the environment entries in the framework's beans' deployment descriptors.

4.4 Data storage components

The persistence transformation as described in section 3.6.2 is mostly implemented by a number of stylesheets. The generation of home and remote interfaces, bean implementation classes, primary key and content classes, and deployment descriptors is already working - including customization templates -, though not all datatypes listed in table 3.3 on page 66 are supported. Automatically creating a deployable JAR file (containing a mixture of framework and arbitrary additional classes) as well as merging generated and hand coded deployment descriptors is working as well. A sequence manager that allows portable generation of artificial primary keys with a minimum of database access is in place, and also most of the XML interface for dataclasses is generated. Automatic construction of SQL scripts for creating and dropping necessary database tables is supported too.

Still on the todo list are the handler components and process graphs for the dataclasses. Furthermore, the content classes can only be used to transfer an entity bean's state, i.e. they don't implement the `Content` interface yet. Also missing are the relational adaptors.

Chapter 5

Summary and assessment

The subject in section 1.2 stated that the main purpose of this thesis was to compare an Intershop based solution with a newly developed framework based on open standards. Since development with Intershop enfinity is non-trivial - at least if more than just basic customization is desired - and enfinity has the disadvantages mentioned in section 2.3.9, it was left aside in the end. Also, given the limited time frame of a thesis, no properly designed framework could have been developed if an Intershop based solution would have been realized.

However, some comparison can be made anyway. Firstly, enfinity is one large bundle where the individual components cannot be replaced easily. It is tied together with the PowerTier application server and the Oracle 8i database in a way that makes replacement of one of these very difficult. Secondly, Intershop licenses are very expensive. In contrast, the approach taken in this thesis is completely independent of a particular application server or database[1]. Also, the license costs can be kept to minimum by using a free application server like JOnAS from Bull Information Systems and an inexpensive database (or one already installed at the customer's site). Also totally free databases can be used, provided they have support for transactions. Unfortunately, this is often not the case. In fact, except for the database, no commercial software was used during development. On the other hand, enfinity offers a lot of standard functionality, e.g. product customizations, that is not (yet) part of the framework. Thus, for standard tasks, a solution can be developed more quickly using enfinity. However, special cases and automated transactions can be handled in a more flexible way by the framework developed in this thesis.

Heidelberger Druckmaschinen AG, who were supposed to act as a pilot customer for an application based on the framework, still haven't decided for or against development of the pilot application. For this reason, no design or development regarding the pilot application is presented in this document. Because of this, chapters 2 and 3 are somehow disconnected. However, the requirements analysis for the pilot application provided valuable input for the design of the framework. In fact, it's almost impossible to develop a framework without concrete requirements. Furthermore, the framework will probably have to be adjusted as applications are developed with it.

While designing the framework it became clear that Enterprise JavaBeans ([Suna]) don't provide a complete persistence solution (see section 3.6). Especially queries are only poorly supported. However, it can be used to do database *updates* automatically. Furthermore, EJB provides transparent distribution of components, declarative transactions on a per-method basis, and automatic life cycle management and caching for component instances. In summary, the EJB specification is not yet complete with respect to automatic persistence management but can already be used as a solid foundation. Future versions of the specification will address the problems depicted in this thesis.

One of the basic principles of the framework is code generation. During development, it is used to generate persistence related files (mainly Java source and SQL scripts). At runtime, the process graphs introduced in section 3.4 are transformed to Java source code. Since all transformation input is based on XML, XSLT ([W3Cb]) seemed a natural choice to perform the transformations. However, as section C.1 shows, XSLT is not well suited to producing text (instead of XML) output. Such stylesheets can grow rather huge even if they

[1] With the exception of generated SQL scripts, but this is a minor problem.

produce only simple output files. An alternative would be to use languages like Perl that are better at creating arbitrary text. On the other hand, XSLT provides powerful selection, filter, and sorting mechanisms that can be applied to the source document. In the end, I still believe XSLT was a good choice.

Since the interface of the framework is based on exchanging XML documents, one might ask why the Simple Object Access Protocol (SOAP, http://www.w3.org/TR/SOAP/) was not taken into account. The main reason is that SOAP merely provides a remote method call via HTTP, packaged in an XML document. For the framework, a higher level interface (i.e. more oriented on the domain requirements) seemed to be the better solution. On the other hand, SOAP would have eliminated the need to parse XML documents, thus easing the development of components for the framework. However, since components should be designed in a way that separates parsing of request documents from methods doing the actual work (see section 3.4.4), it should be possible to add SOAP support for these methods in future versions of the framework.

Business process encapsulation using process graphs - described in section 3.4 - is hopefully a good compromise between abstraction, flexibility, and maintenance concerns. This approach can also be adapted to other architectures than Enterprise JavaBeans. For example, Visual Basic Script code could be generated in a COM (Component Object Model, Microsoft) environment.

Unfortunately, not much development work could be done within the given time frame. Due to that fact, the development included only one iteration and was focused on the persistence transformation introduced in section 3.6. One point that should be considered in future development is compressing the XML requests and responses. For large documents, this might enhance performance by reducing network traffic.

In summary, the framework proves that open standards provide for flexible real-world solutions that can be used in a variety of IT environments. Furthermore, it shows that enterprise level applications can be developed and deployed using open source software to a great extent. I think that open source will play a great role in future software development. It is an interesting development model that makes high quality software free for everyone and shifts profit making over to higher level services and products. You could say that open source is the ultimate reuse model, providing basic functionality for free while allowing to adapt the source code to specific needs and to participate in the development.

Appendix A

XML schema for the framework (excerpt)

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<schema
 xmlns="http://www.w3.org/1999/XMLSchema"
 xmlns:ecm="http://www.isb-ka.de/ecom" xmlns:gen="http://www.isb-ka.de/ecom/gen"
 targetNamespace="http://www.isb-ka.de/ecom" elementFormDefault="qualified"
>

   <!-- Include the extension schema -->
   <include schemaLocation="../generated/schema/schema.ext"/>


   <!--
   # Simple types used in several complex types.
   #-->

   <simpleType name="sessionOrRequestId" base="string">
      <maxLength value="260"/>
   </simpleType>
   <simpleType name="severity" base="string">
      <enumeration value="information"/>
      <enumeration value="warning"/>
      <enumeration value="error"/>
      <enumeration value="fatalError"/>
   </simpleType>
   <simpleType name="sequenceNumber" base="integer">
      <precision value="15"/>      <!-- this has to fit into a Java long -->
   </simpleType>
   <simpleType name="key" base="string">
      <maxLength value="150"/>
   </simpleType>


   <!--
   # <request>, <response>
   #
   # Base types for requests/responses.
   #-->

   <complexType name="request" content="empty">
      <attribute name="minResponse" use="fixed" value="minResponse" type="NMTOKEN"/>
      <attribute name="sessionId" use="optional" type="ecm:sessionOrRequestId"/>
      <attribute name="requestId" use="optional" type="ecm:sessionOrRequestId"/>
      <attribute name="lang" use="optional" type="language"/>
   </complexType>
   <complexType name="response" content="empty">
      <attribute name="sessionId" use="optional" type="ecm:sessionOrRequestId"/>
```

```
    <attribute name="requestId" use="optional" type="ecm:sessionOrRequestId"/>
  </complexType>

  <element name="request" type="ecm:request" abstract="true"/>
  <element name="response" type="ecm:response" abstract="true"/>


  <!--
  # <request-bundle>, <response-bundle>
  #
  # Types for automated processing.
  #-->

  <complexType name="request-bundle" content="elementOnly">
    <attribute name="sessionId" type="sessionOrRequestId"/>
    <element name="do-not-report">
      <complexType content="elementOnly">
        <element name="severity" type="ecm:severity"
         minOccurs="0" maxOccurs="unbounded"/>
      </complexType>
    </element>
    <element name="continue-if">
      <complexType content="elementOnly">
        <element name="severity" type="ecm:severity"
         minOccurs="0" maxOccures="unbounded"/>
      </complexType>
    </element>
    <element name="requests" content="elementOnly">
      <element ref="ecm:request" minOccurs="1" maxOccurs="unbounded"/>
    </element>
  </complexType>
  <complexType name="response-bundle" content="elementOnly">
    <element ref="ecm:response" minOccurs="0" maxOccurs="unbounded"/>
  </complexType>

  <element name="request-bundle" type="ecm:request-bundle"/>
  <element name="response-bundle" type="ecm:response-bundle"/>


  <!--
  # <message>
  #
  # Type for returning success or error messages.
  #-->

  <complexType name="message-response" content="elementOnly"
   base="ecm:response" derivedBy="extension">
    <attribute name="id" type="string"/>
    <attribute name="severity" type="ecm:severity"/>
    <element name="text">
      <complexType content="mixed">
        <attribute name="lang" type="language"/>
        <element name="param" type="string"/>
      </complexType>
    </element>
  </complexType>

  <element name="message" type="ecm:message-response" equivClass="ecm:response"/>
   <!-- The equivClass attribute allows this element to appear everywhere the
    "response" element can appear. -->

</schema>
```

Appendix B

DTD for process graphs

```
<!ENTITY % code "#PCDATA">
<!ENTITY % subgraph "%code; | component-call | get-var | is-var-set | copy-var |
 set-var | clear-var | process-call | processing-exception | if | loop">
<!ENTITY % expression "%code; | get-var | is-var-set">
<!ENTITY % types "int | string | bool | float">
<!ENTITY % scopes "Session | Request">

<!ELEMENT process-graph (prologue?, components?, graph)>
<!ATTLIST process-graph
   name CDATA #REQUIRED
>

<!ELEMENT prologue (%code;)>
   <!-- The prologue is inserted right after the
   package declaration. Thus, it can be used to
   specify additional imports. -->

<!ELEMENT components (component+)>

<!ELEMENT component EMPTY>
<!ATTLIST component
   type (bean | enterprise-bean) #REQUIRED
   alias CDATA #REQUIRED
   name CDATA #REQUIRED
>

<!ELEMENT graph (%subgraph;)*>

<!ELEMENT component-call EMPTY>
<!ATTLIST component-call
   alias CDATA #REQUIRED
   method CDATA #REQUIRED
>

<!ELEMENT get-var EMPTY>
<!ATTLIST get-var
   name CDATA #REQUIRED
   type (%types;) #REQUIRED
   scope (%scopes;) "Request"
>

<!ELEMENT is-var-set EMPTY>
<!ATTLIST is-var-set
   name CDATA #REQUIRED
   scope (%scopes;) "Request"
>
```

```
<!ELEMENT set-var (%expression;)*>
<!ATTLIST set-var
   name CDATA #REQUIRED
   type (%types;) #REQUIRED
   scope (%scopes;) "Request"
>

<!ELEMENT clear-var EMPTY>
<!ATTLIST clear-var
   name CDATA #REQUIRED
   scope (%scopes;) "Request"
>

<!ELEMENT copy-var EMPTY>
<!ATTLIST copy-var
   source CDATA #REQUIRED
   dest CDATA #REQUIRED
>

<!ELEMENT process-call EMPTY>
<!ATTLIST process-call
   name CDATA #REQUIRED
>

<!ELEMENT processing-exception (parameters)?>
<!ATTLIST processing-exception
   id CDATA #REQUIRED
>

<!ELEMENT parameters (parameter)*>

<!ELEMENT parameter (%expression;)*>

<!ELEMENT if (condition, true, false)>

<!ELEMENT condition (%expression;)*>

<!ELEMENT true (%subgraph;)*>

<!ELEMENT false (%subgraph;)*>

<!ELEMENT loop (condition, body)>

<!ELEMENT body (%subgraph;)*>
```

Appendix C

Example XSLT stylesheets

C.1 Stylesheet for generating the remote interface of an entity bean

```
<?xml version="1.0"?>

<!--
# Entities to make life easier.
#-->

<!DOCTYPE xsl:stylesheet [

<!ENTITY bean "<xsl:value-of select='$bean'/>">
<!ENTITY class "<xsl:value-of select='$class'/>">
<!ENTITY package "<xsl:value-of select='$package'/>">
<!ENTITY tmpl-package "<xsl:value-of select='$tmpl-package'/>">
<!ENTITY content-class "<xsl:value-of select='$content-class'/>">
<!ENTITY primkey-class "<xsl:value-of select='$primkey-class'/>">
<!ENTITY base-tmpl-package "<xsl:value-of select='$base-tmpl-package'/>">
<!ENTITY base-remote-class "<xsl:value-of select='$base-remote-class'/>">

<!ENTITY t "<xsl:text>   </xsl:text>">
<!ENTITY t2 "<xsl:text>      </xsl:text>">
<!ENTITY t3 "<xsl:text>         </xsl:text>">
<!ENTITY n "<xsl:text>
</xsl:text>">
<!ENTITY n2 "<xsl:text>

</xsl:text>">
<!ENTITY n3 "<xsl:text>


</xsl:text>">
<!ENTITY s "<xsl:text> </xsl:text>">
<!ENTITY e "<xsl:text/>">   <!-- helper entity for pretty-printed formatting -->
]>


<!--
# The stylesheet.
#-->

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
 xmlns:xsd="http://www.w3.org/1999/XMLSchema"
>
<xsl:output method="text"/>
<xsl:strip-space elements="*"/>
<xsl:include href="gen_bean.xsl"/>
```

```
<!--
# Parameters
#
# bean: which bean to generate (name of the data class)
# class: name of the generated class
# package: package where the generated classes are placed in
# tmpl-package: package where the template classes are placed in
# content-class: name of the associated content-class
# primkey-class: name of the primary key class
# base-tmpl-package: package where the base bean templates are located
# base-remote-class: the remote interface of the base bean
#-->
<xsl:param name="bean"/>
<xsl:param name="class"/>
<xsl:param name="package"/>
<xsl:param name="tmpl-package"/>
<xsl:param name="content-class"/>
<xsl:param name="primkey-class"/>
<xsl:param name="base-tmpl-package"/>
<xsl:param name="base-remote-class"/>


<!--
# Template for the data class.
#-->
<xsl:template match="/xsd:schema/xsd:complexType[@name=concat($bean, '-db')]">

   <!--
   # Output the header comment.
   #-->
   &e;/*****************************************************************&n;
    &e; * Datei:        &class;.java&n;
    &e; *****************************************************************&n;
    &e; * Erstellt am : @DATE@ von: gen_remote-intf.xsl&n;
    &e; *****************************************************************/&n;
   &n;

   <!--
   # Output the package name.
   #-->
   &e;package &package;;&n2;

   <!--
   # Output the imports.
   #-->
   <xsl:call-template name="general-imports"/>
   &e;import java.rmi.RemoteException;&n;
   &e;import javax.ejb.EJBObject;&n;
   &n2;

   <!--
   # Output the class.
   #-->
   &e;/**&n;
   &e; * This is the remote interface for the&s;
   <xsl:call-template name="first-to-upper">
      <xsl:with-param name="string">&bean;</xsl:with-param>
   </xsl:call-template>
   &s;bean.&n;
   &e; *&n;
   &e; * This class has been auto-generated.&n;
   &e; */&n;
   &n;
```

```
      &e;public interface &class; extends&s;
      <xsl:choose>
         <xsl:when test="$base-remote-class">
            &e;&base-tmpl-package;.&base-remote-class; {&n2;
         </xsl:when>
         <xsl:otherwise>
            &e;EJBObject {&n2;
         </xsl:otherwise>
      </xsl:choose>

         <xsl:if test="not($base-remote-class)">
            <!--
            # Output the getter and setter for the content class
            # (plus a getter for the primary key - just for
            # convenience).
            # Currently, no exceptions (except RemoteException) are
            # declared. Maybe this will have to be changed.
            #-->
            &t;/**&n;
            &t; * Returns the primary key of this instance.&n;
            &t; *&n;
            &t; * @return      the primary key.&n;
            &t; *&n;
            &t; * @exception   RemoteException      mandatory EJB exception.&n;
            &t; */&n2;
            &t;public Object getPrimkey() throws RemoteException;&n2;
            &n;
            &t;/**&n;
            &t; * Returns the content (including the primary key fields) of this instance.&n;
            &t; *&n;
            &t; * @return      the content.&n;
            &t; *&n;
            &t; * @exception   RemoteException      mandatory EJB exception.&n;
            &t; */&n2;
            &t;public Object getContent() throws RemoteException;&n2;
            &n;
         </xsl:if>

         &t;/**&n;
         &t; * Sets the content of this instance.&n;
         &t; *&n;
         &t; * The primary key fields in the content parameter are ignored.&n;
         &t; *&n;
         &t; * @param      content             the new content.&n;
         &t; *&n;
         &t; * @exception   RemoteException      mandatory EJB exception.&n;
         &t; */&n2;
         &t;public void setContent(&tmpl-package;.&content-class; content)&s;
         &e;throws java.rmi.RemoteException;&n2;

      &e;}&n;

   </xsl:template>

</xsl:stylesheet>
```

C.2 Stylesheet for transforming catalog pages to HTML

```
<?xml version="1.0"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
 xmlns:ecm="http://www.isb-ka.de/ecom"
 xmlns:cat="http://www.isb-ka.de/ecom/cat"
 exclude-result-prefixes="ecm cat"
>
<xsl:output method="html" indent="yes"/>
<xsl:strip-space elements="*"/>

   <xsl:template match="ecm:get-catalog-pages-response">
      <html>
         <head>
            <title>Search results</title>
         </head>
         <body>
            <xsl:apply-templates/>
         </body>
      </html>
   </xsl:template>

   <xsl:template match="ecm:page-text[position() != 1]">
      <hr />
      <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="ecm:price">
      <b>
         <xsl:apply-templates/>
      </b>
   </xsl:template>

   <xsl:template name="make-link-url">
      <xsl:text>/servlet/gateway?command=</xsl:text>
      <xsl:value-of select="@type"/>
      <xsl:text>Link&amp;dest=</xsl:text>
      <xsl:value-of select="@dest"/>
   </xsl:template>

   <xsl:template match="cat:link">
      <p>
         <xsl:choose>
            <xsl:when test="@type='image'">
               <img>
                  <xsl:attribute name="src">
                     <xsl:call-template name="make-link-url"/>
                  </xsl:attribute>
                  <xsl:attribute name="alt">
                     <xsl:value-of select="normalize-space(.)"/>
                  </xsl:attribute>
               </img>
            </xsl:when>
            <xsl:otherwise>
               <a>
                  <xsl:attribute name="href">
                     <xsl:call-template name="make-link-url"/>
                  </xsl:attribute>
                  <xsl:value-of select="."/>
                  <xsl:if test="string(.)='' and @type='cart'">
                     <xsl:text>Put this into the shopping cart</xsl:text>
                  </xsl:if>
```

```
                </a>
            </xsl:otherwise>
        </xsl:choose>
    </p>
</xsl:template>

</xsl:stylesheet>
```

Appendix D

Bibliography

[ALB] Fraunhofer Anwendungszentrum für Logistikorientierte Betriebswirtschaft (ALB), Paderborn. *Virtuelles Competence-Center zum Thema E-Commerce*.
http://www.e-commerce-systeme.de/competence/ccecom.nsf.

[Apa] The Apache XML Project. *Xalan-Java*.
http://xml.apache.org/xalan/.

[BMW99] Bundesministerium für Wirtschaft und Technologie (BMWi). *Was ist Elektronischer Geschäftsverkehr*, Oct 1999.
http://www.ec-net.de/info/Was_ist_EC.html.

[ECM] ECML Alliance. *ECML.org Home Page*.
http://www.ecml.org/.

[EDI] The XML/EDI Group. *XML/EDI Group's Home Page*.
http://www.xmledi.com/.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Pub Co, 1995. ISBN 0-201-63361-2.

[Har00] Elliotte Rusty Harold. *XSL Transformations*, 2000.
http://metalab.unc.edu/xml/books/bible/updates/14.html.

[IET] Internet Engineering Task Force. *Transport Layer Security (tls) Charter*.
http://www.ietf.org/html.charters/tls-charter.html.

[Kel97] Wolfgang Keller. *Mapping Objects to Tables*, 1997.
http://www.objectarchitects.de/ObjectArchitects/orpatterns/index.htm.

[MS] Microsoft Corporation. *BizTalk Home Page*.
http://www.biztalk.org/BizTalk/default.asp.

[Net] Netscape Communications. *SSL 3.0 Specification*.
http://www.netscape.com/eng/ssl3/.

[OMG] The Object Management Group (OMG). *CORBA*.
http://www.omg.org/corba.

[Ope] OpenSSL team. *OpenSSL: The Open Source toolkit for SSL/TLS*.
http://www.openssl.org/.

[RSA] RSA Security Inc. *Cryptography FAQ*.
http://www.rsasecurity.com/rsalabs/faq/.

[Sie] Siemens IT Service. *eCommerce Glossary / Buzzwords*.
http://www.siemens-it-service.com/e/facts/glossary.html.

[Spe99] Johannes Spechtl. *Electronic Commerce - Einsatzpotential in kleinen und mittleren Unternehmen*, May 1999.
http://members.aon.at/jspechtl/Diplomarbeit.htm.

[Suna] Sun Microsystems, Inc. *Enterprise JavaBeans Technology*.
http://java.sun.com/products/ejb.

[Sunb] Sun Microsystems, Inc. *J2EE Platform Specification*.
http://java.sun.com/j2ee/download.html\#platformspec.

[Sunc] Sun Microsystems, Inc. *Java Blend - Home*.
http://www.sun.com/software/javablend/.

[Sund] Sun Microsystems, Inc. *Java Servlet API*.
http://java.sun.com/products/servlet/index.html.

[Sune] Sun Microsystems, Inc. *JavaServer Pages Technology*.
http://java.sun.com/products/jsp/index.html.

[Sunf] Sun Microsystems, Inc. *JDBC Technology*.
http://java.sun.com/products/jdbc/.

[TOP] The Object People Inc. *TOPLink Home Page*.
http://www.objectpeople.com/toplink/index.htm.

[W3Ca] World Wide Web Consortium (W3C). *XML Schema*.
http://www.w3.org/XML/Schema.

[W3Cb] World Wide Web Consortium (W3C). *XSL Transformations (XSLT)*.
http://www.w3.org/TR/xslt.