Integration of document-centric content in an enterprise portal

Design and prototypical implementation of a generic application architecture based on IBM Lotus Notes middleware

Diploma Thesis submitted by Jens Heipmann matriculation no. 6121038

Advisor: Prof. Dr. Ludwig Nastansky
Second Advisor: Prof. Dr. Leena Suhl
Tutor: Dipl.-Wirt. Inform. Bernd Hesse

January 10, 2007
Abstract

This diploma thesis is about the design of generic architecture used to integrate document-centric content stored in IBM Lotus Notes into a Java based and process driven enterprise portal. Starting with an introduction of the main concepts and technologies, an integration strategy is identified. This strategy is based on an evaluation of integration technologies and levels from both the technical and business perspective. Then requirements of a cross-system application architecture are presented and according to them a tiered architecture is designed. The applicability of this architecture is tested by a sample application developed to prove the concept. This application was realized with IBM Lotus Notes 6.5, IBM WebSphere Portal 5.1 and the open source application framework Spring 2.0. This paper gives hints on how to integrate existing Notes applications into a portal and how to design new applications to be integrated.
Acknowledgments

I would like to thank everybody who directly or indirectly assisted in the creation of this diploma thesis:

- Auguste Goldman and Axel Grocholesky who made it possible that I could write this diploma thesis in cooperation with Linde IT Services in Stockholm and who provided me with all needed resources.

- Ludwig Nastansky and Ingo Erdmann who woke up my interest in collaborative applications, Java and Lotus Notes.

- Bernd Hesse who guided and commented the writing of this paper. I appreciated the discussions about the outline and that he kept a level head when I doubted my concept.

- Michael Schädle who helped me understanding Linde’s IT infrastructure and who supported me with the configuration of Lotus Domino. He and Andreas Koch, Jens Hitzel, Marcel Damseaux, Patrik Hutnik, Wai Tai Chan and my other coworkers at LIS kept my spirits up when I was struggling with the portal, Domino and the problems you are confronted with when living abroad.

- Karim Lofti whose portal workshop was very interesting and productive and who patiently answered my questions.

- Stefan Kramkowksi who organized the printing of this paper and handed it in right before it was due.

- Barbara Miraftabi, Mikael Turner and Birke Jantz who helped me out with my English.

- Päivi who changed my mood and gave me new motivation when I felt bad because of working for this project.

With the words of 1 Corinthians 3:6-7 I would like to express my thanks to God: ‘I planted the seed, and Apollos watered it. But God is the One who made it grow. So the one who plants is not important, and the one who waters is not important. Only God, who makes things grow, is important.’
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Nomenclature</strong></td>
<td>viii</td>
</tr>
<tr>
<td><strong>List of Figures</strong></td>
<td>x</td>
</tr>
<tr>
<td><strong>1. Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1. Scenario</td>
<td>1</td>
</tr>
<tr>
<td>1.2. Objectives</td>
<td>2</td>
</tr>
<tr>
<td>1.3. Overview of the paper</td>
<td>3</td>
</tr>
<tr>
<td><strong>2. Basics</strong></td>
<td>4</td>
</tr>
<tr>
<td>2.1. Lotus Notes</td>
<td>4</td>
</tr>
<tr>
<td>2.1.1. Two-tier architecture</td>
<td>4</td>
</tr>
<tr>
<td>2.1.2. Components</td>
<td>5</td>
</tr>
<tr>
<td>2.1.3. Remote data access</td>
<td>7</td>
</tr>
<tr>
<td>2.2. Enterprise Portal</td>
<td>11</td>
</tr>
<tr>
<td>2.2.1. Concept</td>
<td>11</td>
</tr>
<tr>
<td>2.2.2. Multi-tier architecture</td>
<td>14</td>
</tr>
<tr>
<td>2.2.3. Portal architecture</td>
<td>18</td>
</tr>
<tr>
<td>2.2.4. Development standards</td>
<td>21</td>
</tr>
<tr>
<td><strong>3. Concept and Discussion</strong></td>
<td>24</td>
</tr>
<tr>
<td>3.1. Integration strategy</td>
<td>24</td>
</tr>
<tr>
<td>3.1.1. Patterns and technologies</td>
<td>24</td>
</tr>
<tr>
<td>3.1.2. Technical view</td>
<td>27</td>
</tr>
<tr>
<td>3.1.3. Business view</td>
<td>33</td>
</tr>
<tr>
<td>3.1.4. Conclusion</td>
<td>35</td>
</tr>
<tr>
<td>3.2. Architecture</td>
<td>40</td>
</tr>
<tr>
<td>3.2.1. Requirements</td>
<td>40</td>
</tr>
<tr>
<td>3.2.2. Overview</td>
<td>43</td>
</tr>
<tr>
<td>3.2.3. Resource tier</td>
<td>45</td>
</tr>
<tr>
<td>3.2.4. Integration tier</td>
<td>47</td>
</tr>
<tr>
<td>3.2.5. Business tier</td>
<td>51</td>
</tr>
<tr>
<td>3.2.6. Presentation tier</td>
<td>52</td>
</tr>
</tbody>
</table>
## Nomenclature

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL</td>
<td>Access Control List</td>
</tr>
<tr>
<td>AJAX</td>
<td>Asynchronous Javascript and XML</td>
</tr>
<tr>
<td>AOP</td>
<td>Aspect Oriented Programming</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CSS</td>
<td>Cascading Style Sheets</td>
</tr>
<tr>
<td>DAO</td>
<td>Data Access Object</td>
</tr>
<tr>
<td>DAP</td>
<td>Domino Application Portlet</td>
</tr>
<tr>
<td>DECS</td>
<td>Domino Enterprise Connection Services</td>
</tr>
<tr>
<td>DI</td>
<td>Dependency Injection</td>
</tr>
<tr>
<td>DMZ</td>
<td>Demilitarized Zone</td>
</tr>
<tr>
<td>DNS</td>
<td>Domain Name System</td>
</tr>
<tr>
<td>DOM</td>
<td>Domino Object Model</td>
</tr>
<tr>
<td>DXL</td>
<td>Domino XML Language</td>
</tr>
<tr>
<td>EIS</td>
<td>Enterprise Information Systems</td>
</tr>
<tr>
<td>EJB</td>
<td>Enterprise Java Bean</td>
</tr>
<tr>
<td>ERP</td>
<td>Enterprise Resource Planning</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IFrame</td>
<td>Inline Frame</td>
</tr>
<tr>
<td>IHS</td>
<td>IBM HTTP Server</td>
</tr>
<tr>
<td>IMAP</td>
<td>Internet Message Access Protocol</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>Java EE</td>
<td>Java Platform, Enterprise Edition</td>
</tr>
<tr>
<td>Java ME</td>
<td>Java Platform, Micro Edition</td>
</tr>
<tr>
<td>Java SE</td>
<td>Java Platform, Standard Edition</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java Database Connectivity</td>
</tr>
<tr>
<td>JMS</td>
<td>Java Messaging Service</td>
</tr>
<tr>
<td>JSF</td>
<td>JavaServer Faces</td>
</tr>
<tr>
<td>JSP</td>
<td>Java Server Page</td>
</tr>
</tbody>
</table>
JSR  .................  Java Specification Request
JSTL ...............  JSP Standard Template Library
JVM ..................  Java Virtual Machine
LC LSX .............  Lotus Connector LotusScript Extension
LDAP ...............  Lightweight Directory Access Protocol
LEI ..................  Lotus Enterprise Integrator
MIME ...............  Multipurpose Internet Mail Extensions
NOS ..................  Notes Object Services
O/R ..................  Object-Relational
OASIS ...............  Organization for the Advancement of Structured Information Standards
ODBC ...............  Open Database Connectivity
OLE ..................  Object Linking and Embedding
OMG ..................  Object Management Group
OO ...................  Object-oriented
OOP ..................  Object-oriented programming
ORB ..................  Object Request Broker
POJO ................  Plain Old Java Object
POP ..................  Post Office Protocol
RDBMS ..............  Relational Database Management Systems
ROI ...................  Return on Investment
RSS ...................  Really Simple Syndication or Rich Site Summary
SMTP ................  Simple Mail Transfer Protocol
SPI ...................  Service Provider Interface
SQL ...................  Structured Query Language
SSO ...................  Single Sign-On
TCP ...................  Transmission Control Protocol
UI ......................  User Interface
URL ...................  Uniform Resource Locator
WAS ...................  IBM WebSphere Application Server
WML ...................  Wireless Markup Language
WPS ...................  IBM WebSphere Portal Server
WSDL ................  Web Service Description Language
WSRP ................  Web Services for Remote Portlets
XHTML ................  Extensible Hypertext Markup Language
XML ...................  Extensible Markup Language
XSL ...................  Extensible Stylesheet Language
XSLT ................  XSL Transformations
List of Figures

2.1. Lotus Notes two-tier architecture .................................. 5
2.2. Notes and Domino components ...................................... 6
2.3. CORBA object communication ........................................ 8
2.4. Web service scenario .................................................. 10
2.5. Mind map of portals ................................................... 12
2.6. Java EE multi-tier architecture ....................................... 15
2.7. Application server configuration .................................... 17
2.8. Portal architecture ..................................................... 18
2.9. Application and portal server configuration ....................... 20
2.10. Portlet interface (JSR-168) .......................................... 21
2.11. WSRP usage ........................................................... 23

3.1. Bottom-up approach for identifying an integration strategy ..... 25
3.2. Common portal topology ................................................ 28
3.3. Portal integration levels ............................................... 34
3.4. Integration technology decision tree ................................ 38
3.5. Integration architecture ................................................. 44
3.6. Editing of simple structured data .................................... 46
3.7. Navigation tree concept ................................................ 47
3.8. Data Access Object (DAO) pattern .................................. 48
3.9. Domino session pooling ............................................... 49
3.10. Model View Controller (MVC) pattern ............................. 53
3.11. Implementation of MVC pattern .................................... 54
3.12. Portlet file and image handling ..................................... 55

4.1. Requirements of sample application ................................. 57
4.2. Portal infrastructure ..................................................... 58
4.3. Screenshot of article Form, basics tab .............................. 61
4.4. Relationship between article and file attachments .......................... 62
4.5. Screenshot of article Form, files tab ............................................. 63
4.6. Screenshot of the front-end navigation tree View ............................... 64
4.7. Class diagram of session pool service .............................................. 66
4.8. Spring beans of session pool service ............................................... 67
4.9. Class diagram of navigation tree service ......................................... 68
4.10. Spring beans of navigation tree service ......................................... 69
4.11. Class diagram of portlet application ............................................... 70
4.12. Spring beans of portlet application ............................................... 72
4.13. Portlet wiring tool of WebSphere Portal ........................................ 73
4.14. Inter-portlet communication ......................................................... 74
4.15. Screenshot of portlet application .................................................... 75

5.1. Integration architecture with data-broker ....................................... 77
1. Introduction

1.1. Scenario

‘Data tends to stick where it lands.’ (Johnson 2003, p. 57)

This introductory statement expresses an unexpected truth: Data tends to be immobile. Even though a huge and increasing amount of data is flying everyday around the earth and is exchanged back and forth, it is a huge effort to move or reorganize data once it is persisted, i.e. saved. Probably every user of a digital camera has experienced this if pictures are not immediately organized in a proper way. How much more does it apply to data being stored in databases. The effort for reorganization increases with the amount of metadata, the complexity of the domain model and the storage system itself. This is the reason why databases, as the heart of enterprise applications, are seldom switched and organizations as well as individuals carefully consider in which system they ‘sink’ (Weber 2006) their data.

Due to different requirements or related to new emerging technologies, it is natural that several data pools develop within an organization. A research paper of Accenture comments on this data proliferation as follows: ‘Ever since the first company built their second computer application, integrating data across applications has been a perplexing problem.’ (Harris & Cantrell 2002, p. 1)

The research domain of this diploma thesis is about data integration. Since software systems mature and the amount of handled data increases rapidly, organizations aim to consolidate the view of their data in different data pools (and related applications) into one unified knowledge desktop (Collins 2000, p. 8). This knowledge desktop offers a single point of access that integrates, in addition to data itself, business processes and stakeholders like customers, employees and business partners (Goebel & Ritthaler 2005, p. 13). Thus, not only existing data is leveraged and redundant data storage is prevented, but also the quality of data increases and business processes are sped up (Vlachakis et al. 2005, p. 9).
A software product that addresses this concept of a knowledge desktop is an enterprise portal. According to (Hahnl 2004, p. 19) the origin of the term enterprise portal refers back to a Merrill-Lynch report of 1998 (Shilakes & Tylman 1998). Nowadays enterprise portals are one of the leading ideas in the development and implementation of enterprise software. Even new functionalities of existing products might be accessible only via the portal interface. Klaus Kreplin, member of the extended Board of SAP, states: ‘We do not disable the SAPgui\(^1\), but new features are only implemented for the portal’ (Niemann 2003).

It is not astonishing that enterprise portals are covered extensively in the literature. The concept is described, e.g. in the book ‘Corporate Portals: Revolutionizing Information Access to Increase the Productivity and Drive the Bottom Line’ (Collins 2000), and since there is a common development standard the technical point of view is also described in detail by books like ‘Portlets’ (Zörner 2006) or ‘Portlets and Apache Portals’ (Hepper et al. 2005).

1.2. Objectives

Even though several publications address the concept and technical handling of enterprise portals, there are not many papers that deal with the integration of data stored in semi-structured compound-documents within IBM Lotus Notes databases. The reason might be found in the fact that this data storage system is a bit exotic in the field of enterprise portals since most portal servers offer connectivity support only for highly structured data accessible via relational database systems. The only available publications that fully focus on integrating Lotus Notes based content into an enterprise portal are the very similar IBM Redbooks entitled ‘Portalizing Domino Applications for WebSphere Portal’ (Tulisalo et al. 2003) and ‘Portalizing Domino Applications: Integration with Portal 5.02 and Lotus Workplace 2.0.1’ (Bergland et al. 2005).

The objective of this diploma thesis is to target this gap and design a generic architecture that can be used to integrate document-centric content stored within IBM Lotus Notes databases into an enterprise portal. The above mentioned IBM Redbooks are taken as a starting point, and related to sound object oriented design principles, a multi-tier architecture is developed that answers the practical question: ‘What is the best way to integrate my Lotus Notes application into a Java based and

\(^1\)SAPgui is SAP’s universal client for accessing SAP functionality.
process driven enterprise portal?’ Even though ‘there are many options for Domino integration into the Portal’ (Tulisalo et al. 2003, p. 1), not many of these options fulfill the practical requirements organizations need for their business applications.

After the architecture has been designed and discussed, its applicability is tested by a prototypical implementation of a content management system realized with IBM Lotus Notes 6.5, IBM WebSphere Portal 5.1 and the open source application framework Spring 2.0.

1.3. Overview of the paper

This diploma thesis is structured as follows: The second chapter deals with the basics relevant to this paper. It starts with a mainly technical introduction of Lotus Notes and its components and explains which possibilities there are to remotely interact with Notes databases. Afterwards enterprise portals are brought into the picture. An introduction of the concept behind them is given from both the technical and business perspective. Then Java driven multi-tier and portal architectures are explained as well as related technology standards.

The third chapter is the main chapter of this paper. In the beginning, different approaches are presented of how data stored within Lotus Notes databases could be integrated within a portal. Based on these, requirements for generic application architecture are derived that address both the Lotus Notes and the portlet application. Then an architecture is designed according to these requirements and its different layers are analyzed in detail.

In the forth chapter the discussed architecture is turned into action by a Content Management System that has been developed to prove the concept. The layers of this application are described in detail so that it can be used as a sample for other Notes applications to be integrated within a portal. This is especially true for the Domino session pool and navigation tree service.

Finally an outlook is given how the architecture and the sample application could be improved and which problems would need further investigation.
2. Basics

This chapter is divided into two sections and introduces the basics relevant for this paper. The first section gives an overview about two-tier architectures, the Lotus Notes client and the Domino server, and explains the possibilities how Notes databases can be remotely accessed. The second section of this chapter is about enterprise portals. It starts with an introduction of the concept behind enterprise portals from both the technical and business perspective. Then it moves on and explains Java driven multi-tier and portal architectures and closes with an explanation of related technology standards.

2.1. Lotus Notes

In the year 1989 the Lotus Developer Corporation announced Lotus Notes, a group communication software and according to Jim Manzi (second CEO of Notes) ‘the first personal computer software product to comprehensively address workgroups in the corporate and government environments’ (Lotus 1995). IBM acquired Lotus in 1995 and integrated it into its Software Group. Nowadays the IBM Lotus Notes client is used by more than 120 million users (Gartner Dataquest 2004).

2.1.1. Two-tier architecture

Lotus Notes, a software for communication, collaboration and coordination, is based on a two-tier client/server architecture. (Keogh 2002, p. 27) describes a tier as an ‘abstract concept that defines a group of technologies that provide one or more services to its clients’. Tiered-design has multiple benefits, e.g. the code-base and the data of an application can be spread across multiple machines so that a large and maybe changing amount of users and data can be handled flexibly. Another benefit is that the unique concerns of each tier can be hidden from other tiers. Like this, applications are ‘easy to modify without changes cascading into other tiers’ (Johnson 2003, p. 27).
Figure 2.1. Lotus Notes two-tier architecture

Figure 2.1 visualizes a two-tier architecture and presents a client tier and a resource tier. Since a typical two-tier enterprise application can be divided into user interface and back-end, ‘the user interface talks directly to the database, which in many cases is located on the same machine’ (Couch & Steinberg 2002, p. 4).

This also applies to Lotus Notes, where local databases (or remote databases hosted on a Domino server) can be directly accessed via a Lotus Notes client, a Domino Designer client or a Domino Administrator client. As their names imply, the Domino Designer is used to (rapidly) develop Notes databases and the purpose of the Domino Administrator is to administer Domino servers. Other front-end clients to Lotus databases can be normal web browsers or POP (Post Office Protocol) / IMAP (Internet Message Access Protocol) mail clients.

2.1.2. Components

The following gives a brief technical overview about the Notes client and the Domino server and is mainly based on the document ‘Inside Notes: The Architecture of Notes and the Domino Server’ (IBM 2000).

Notes Object Services (NOS) are portable C/C++ functions that ‘create and access information in databases and files, compile and interpret formulas and scripts and interface to operating systems services’ (IBM 2000, p. 2). Figure 2.2 shows the key software components that use NOS. The Notes client leverages NOS and gives access to databases local to the client computer or to shared databases on a Domino server. The Domino server supports connections to clients and manages a group of server tasks. Server tasks are ‘programs that either perform schedule-driven database chores (such as routing messages to mailboxes and updating user accounts) or connect various types of clients’ (IBM 2000, p. 2) to the server.
already introduced, possible clients can be e.g. a Notes client or a web browser.

A Notes database is a single file that contains both data and application code and to which is referred to as an ‘object store’ (Kern & Lynd 2003, ch. 1). Richard Schwarz (Benz & Oliver 2003, ch. 3) describes this database as ‘a container of a container of containers’: A database contains Notes that contain Items that contain the data. A Note is the basic data structure that stores in its Items the design elements of a database (like Forms or Views), user created data or administrative information. When a Note that stores user created data is rendered in a Form, it is addressed as a semi-structured compound-document (Fischer et al. 2002, p. 253f) because it can contain structured and unstructured information. Single Notes in different replicas of the same database can be synchronized via replication. This resolves the problem in a distributed environment that multiple copies of databases have to be kept consistent while multiple users and programs work on them.

Clients and servers use a special Notes database, a Domino directory, to look for user-, group- or configuration-information of a Domino domain\(^1\). Each Domino directory may contain paths to other additional databases which can be 1) Directory Catalog databases or 2) a Directory Assistance database. A Directory Catalog stores a compact version of some user and group information from one or more Domino directories while the Directory Assistance is a directory of secondary directories. Secondary Domino directories contain usually directory information about other Domino domains.

\(^1\)A Domino domain is a collection of Domino servers and users that share a common Domino Directory. (IBM 2002, Glossary)
The Lightweight Directory Access Protocol (LDAP) is a protocol to access directory information over TCP/IP. Lotus Domino supports LDAP with a special LDAP-task so that entries in the Domino directory can be read or edited. This service can be used from other applications that have to authenticate users; Domino would search the user in its Domino directory. If the user cannot be found the search would be expanded at first to the Directory Catalogs and then to directories referenced to in the Directory Assistance.

2.1.3. Remote data access

There are various possibilities how data stored within Lotus Notes databases could be remotely accessed. Next to the already introduced Notes client or web browser, it is possible to access databases via CORBA, ODBC/JDBC, XML, Web services or via different Lotus connectors. As these technologies are neither directly related to the client nor to the resource tier, figure 2.1 places them in between these tiers. The following will introduce these technologies at a glance.

CORBA

According to (Eberhart & Fischer 2001, p. 195), the Common Object Request Broker Architecture (CORBA) is an open architecture for distributed object systems. It is defined by the Object Management Group (OMG) and supports writing applications that are composed of heterogeneous objects. This means that related objects do not have to be written in the same programming language and they have also not to run in the same operating system or platform.

A simplified CORBA architecture can be seen in figure 2.3. The Object Request Broker (ORB) is the central component enabling transparent communication between objects. Transparency signifies that there is no difference from the clients perspective in instantiating remote or local objects; remote objects seem to be local ones. To support this behavior, every application utilizing CORBA has to have local access to an ORB\(^2\). Multiple ORBs on different computers use IIOP (Internet Inter-Orb Protocol over TCP/IP) to communicate with each other.

With the help of CORBA, Lotus Domino can support remote Java clients even though it is itself written in C++. Thus remote Java applications can use the server

\(^2\)CORBA ‘is implemented as a core part of the Java 2 platform in the org.omg.CORBA package and its subpackages. The implementation includes an ORB that a Java application can use to communicate, as both a client and a server (...)’ (Flanagan et al. 2002, p. 11)
side C++ Domino object model via a C++ CORBA server interface’ (IBM 2000, p. 73) and access Domino’s design elements or read and create data as if the Domino server would be locally available.

It should be noticed that via CORBA personalized user sessions can be handled so that remote Java applications can leverage Domino’s security features, e.g. Reader Fields (IBM 2000, p. 146). It is Domino’s IIOP task which enables CORBA functionality for a Domino server.

**ODBC / JDBC**

‘A Domino database is not relational’ (Tulisalo et al. 2002, p. 794) and contains only one real table, referred to as the Universal Relation (Toulemonde et al. 1999, p. 52). This table has the same name as the database and contains all Fields defined in all Forms of the database. Anyway, it is possible to use special ODBC and JDBC drivers so that a Domino database looks like a relational data source. With the help of these drivers, it is possible to retrieve and manipulate data from Domino Forms, Views and from the Universal Relation using the Structured Query Language (SQL). In the following, both drivers are shortly introduced.

Open Database Connectivity (ODBC) is a relational database driver API and was designed so that Relational Database Management Systems (RDBMS) from different vendors could be accessed in a standard way (Farley 2001, p. 199). The ODBC driver implementation for Domino databases is NotesSQL (Collins et al. 1999, p. 471). NotesSQL requires that the client from which a Lotus Notes database is queried should run on a Windows operating system and should have NotesSQL and a Lotus Notes related product (like the Lotus Notes client) installed. More detailed information can be found on the NotesSQL website³.

Like ODBC, Java Database Connectivity (JDBC) is also a relational database driver API but it is included into the core Java API. JDBC allows Java applications

to read and edit data in RDBMSs. The implementation of this API for Lotus Domino is called *Lotus Domino Driver for JDBC*.

To both of these drivers applies that they can only be used by a technical user (Nielsen et al. 2001, p. 501) and that design elements like Domino Agents cannot be accessed (Toulemonde et al. 1999, p. 158).

**XML**

Markup languages (like HTML) combine text with meta-information about the text. The Extensible Markup Language (XML) is such a markup language, but opposed to HTML it is not designed to describe websites but to define other languages.

According to (McLaughlin 2000, p. 11), XML ‘brings with it a huge promise: what Java did for portability of code, XML claims to do for portability of data.’ The reason for this promise can be found in XML’s support for creating custom tag-sets and language grammars which can be used to write flexible and well defined ‘data-interfaces’. Applications can implement these data-interfaces and use them to exchange data with others.

This use of XML is often called *data-oriented* since XML is applied as an interchange format. One system sends a XML-defined message to another system and this message typically contains information that began or ends up in a (relational) database. In comparison to this, a second class of XML use can be identified. This class is usually known as *document-oriented* and XML is used ‘to impose structure on information that rarely fits into a relational database - particularly information intended for publishing.’ (DuCharme 2004)

The Domino XML Language (DXL) is such a document oriented use of XML and ‘describes Domino specific data and design elements such as embedded views, forms, and documents. As XML becomes the standard basis for exchanging information, DXL provides a basis for importing and exporting XML representations of data to and from a Domino application.’ (Tulisalo et al. 2002, p. 38)

**Web service**

While CORBA and ODBC / JDBC are used to tie systems directly together, XML and web services make it possible to loosely couple systems. This loose coupling enables a more flexible handling of the relation between systems, but is in special situations payed with performance. (Johnson 2003, p. 35)
In common usage, web services enable systems to expose a ‘programmatic interface’ (Iyengar et al. 2005, p. 484) to clients with the help of XML defined SOAP (originally Simple Object Access Protocol) messages. The scenario of figure 2.4 shows a web service consumer sending a SOAP message, i.e. a remote procedure call, to a web service provider. In return, the web service provider answers the request with another SOAP message containing the result. To enable such a communication, the Web Service Description Language (WSDL) is used to define how SOAP messages need to be structured and sent. This is the public interface of the service. (McLaughlin 2001, p. 303)

Lotus Notes supports web services as it can run custom Java Agents. Since version 7.0, Lotus Notes also supports web services natively, but with the two constrains that only provider entities are supported and that SOAP messages have to be bound to HTTP POST (Monson et al. 2006, p. 45).

**Lotus connectors**

Unlike the above mentioned technologies, Lotus connectors are not an open standard but a group of ‘links’ that permit Domino to interact with external systems like ‘RDBMS, Enterprise Resource Planning (ERP) systems, transaction processing systems, directory services, and other services.’ (Nielsen et al. 1999, p. 2) Individual Connectors exist, e.g. for systems like SAP or Oracle, and can be used only in cooperation with Domino Enterprise Connection Services (DECS) or from the product Lotus Enterprise Integrator (LEI). Additionally, it is possible to utilize them with the help of the Lotus Connector LotusScript Extension (LC LSX), the Lotus Connector Java Classes or the Lotus Connector C/C++ API.
2.2. Enterprise Portal

(Collins 2000, p. 7) defines an enterprise portal as:

‘A browser-based application that allows knowledge workers to gain access to, collaborate with, make decisions, and take action on a wide variety of business-related information regardless of the employee’s virtual location or departmental affiliations, the location of the information, or the format in which the information is stored.’

This definition means in brief that a portal helps users to access ‘everything they need to get their tasks done’ (Credle et al. 2006, p. 2). Thus a portal offers a ‘window’ that presents information and a ‘door’ that allows users to pass through to reach selected destinations (Collins 2000, p. 7). According to this description, the task of a portal is to consolidate the view of data, business processes and stakeholders with the aim to modify back-end systems as less as possible.

2.2.1. Concept

Technical view

According to (Stelzer et al. 2004, p. 9), the mainspring behind enterprise portals were intranets which had grown over time and therefor were offering a confusing amount of badly structured and inconsistent information. Uncontrolled growth had contradicted their basic intention, which was to offer user friendly access to relevant information. Users had to spend more and more time, or were not able to find what they needed to fulfill their tasks. The new concept of a personalized knowledge desktop (see page 1) promised to improve the area of information retrieval by pre-structuring information.

Software vendors targeted knowledge desktops in the late 1990s and started to develop enterprise portals. (Zörner 2006, p. 13f) identifies the following four main topics these portals nowadays address (also see figure 2.5).

1. Aggregation: As already introduced, portals are designed to be ‘integration platforms’ (Vlachakis et al. 2005, p. 35) that aggregate the view of data and applications and offer a single point of access.

2. Personalization: One problem of intranets was their unstructured amount of data with the result that organizations did not knew what they knew (Collins
et al. 1999, p. 11). Portals try to reduce the amount of displayed and accessible content by presenting only what is needed in the user’s context. Thus it is often required that personalized content is delivered according to the role or current activity of a user. It might also be possible that users by themselves can configure and compose a personal portal page by selecting components they are interested in.

3. Presentation: Portals are responsible for the presentation of embedded content and they are often used to establish new and consistent user interface (UI) standards for Web based applications within organizations. Sometimes support of different clients (like Web browsers or mobile phones) might be necessary, but more common is the need to support internationalization or to offer universal access, e.g. for visually handicapped users. Both areas of personalization and presentation are strongly related to each other so that an individually optimized UI can be presented.

4. Security: Users do not want to spend time for multiple logins which might be needed for every system accessible throughout a portal. This is especially true if there is the possibility to login to back-end systems with the same credentials used to login into the portal. It is the task of the portal infrastructure to offer Single Sign-On (SSO) capabilities which require that existing directories (e.g. a Domino directory, see page 6) are integrated into one security architecture.

**Business view**

The aim behind these four topics (and enterprise portals in general) is to increase the productivity of employees. A positive Return on Investment (RoI) should be achieved by enabling employees to speed up their work and therefore to decrease costs. As a byproduct of this, it is expected that the quality of decisions increases
because of better access of needed information. Unfortunately Dirk Stelzer (Stelzer et al. 2004, p. 23) claims, that it has not been possible to empirically verify that portals could increase efficiency of labor.

This is a remarkable statement and therefore has to be further analyzed. The same author states three pages earlier (Stelzer et al. 2004, p. 20), that process orientation and job enrichment are the reasons for an increasing amount of software systems, and that ‘the more heterogeneous these systems are, the less productive are the employees’.

Stefan Bohkmann and Heiko Stock (Stelzer et al. 2004, p. 122) seize this argumentation and state that in the past, enterprise portals often only offered basic features supporting the general flow of information in an enterprise. An extract of these basic features is presented in the following list:

- distribution of corporate news;
- publication of customer or enterprise magazines;
- white pages;
- index of branches;
- publication of companywide standards;
- calendar of events;
- information of available products; and
- organization charts.

Even though these features might be useful for an organization, their business value does not justify an investment in a complex enterprise portal. It is difficult to transform these ‘soft’ components to quantifiable values needed for the funding of a portal project. Stefan Bohkmann and Heiko Stock conclude, that an investment into a portal can only be justified if the solution targets the core processes of an enterprise (Stelzer et al. 2004, p. 123). Even though a portal does not necessarily reorganize business processes, it reorganizes the information flow and thus delivers a sustainable benefit for the organization (Stelzer et al. 2004, p. 129).

After both technical and business concepts behind enterprise portals have been presented, it can be concluded that from the technical side portals are about aggregation, presentation, personalization and security. The business side emphasis that
only if these functionalities are primarily used to improve core business processes, an investment into a portal infrastructure can be justified.

2.2.2. Multi-tier architecture

As a technical basis for the rest of this paper, multi-tier architectures are introduced in the context of the Java programming language. Java was invented by Sun Microsystems with a focus on interoperability, and its fundamental slogan of ‘Write Once, Run Anywhere’ ensures that Java applications ‘are far more portable than applications written using any comparable platform’ (Johnson 2003, p. 56). This support for portability is one reason why Java is intensively used in the area of enterprise applications, others can be found in object-oriented software techniques, stability of the execution environment and a large number of free and commercially available APIs (KBSt 2003, p. 60).

Java EE

Three different Java platforms exist, which are Standard, Micro and Enterprise Edition. While the Standard Edition (Java SE) contains libraries needed for general-purpose-use on desktop computers and servers, the Micro Edition (Java ME) is focused on mobile devices with limited capacities and the Enterprise Edition (Java EE) extends Java SE with special API specifications that are useful for the development of multi-tier applications.

This section focuses on Java EE since common portal servers like IBM WebSphere Portal, JBoss Portal, Liferay, Oracle Portal or SAP Enterprise Portal rely on this technology (Vlachakis et al. 2005, p. 34). Microsoft’s SharePoint Server is the only common portal that bases on the .NET Framework.

The concept of a tier has been already introduced in the context of the two-tier architecture of Lotus Notes (refer to page 4). Multi-tier architectures split systems in even more than two layers with the effect that the benefits from separation of concerns get more apparent. (Johnson 2003, p. 27) concludes, that ‘systems with three or more tiers have proven more scalable and flexible than client server systems, in which there is no middle tier.’ The purpose of this new introduced middle tier is to ‘represent an abstraction of the functionality [of an application]’ (Couch & Steinberg 2002, p. 5), so that the user interface is not any more dependent on the database implementation.
This principle of the separation of concerns, which is also important in the area of object-oriented programming (OOP), does not stop with the division of a system into multiple tiers. According to (Keogh 2002, p. 27), a tier itself can be divided into logical, reusable *components*. A component is a part of a tier and provides a service to a client. Components are managed by a *container* that provides them with system-level services (like threading, security or resource management). These entities are visualized in figure 2.6.

1. The *client tier* (or client-side presentation) is where users interact with a system and where data and a user interface are visualized. Java EE offers two containers for this tier: An applet container and an application client container. Applets are software components that run in the context of another program, e.g. a web browser. (Iyengar et al. 2005, p. 18)

2. The *presentation tier* (or server-side presentation) encapsulates the presentation logic of a Java EE application. This includes session management, receiving requests from clients, then generating responses. The Web container is associated with this tier and within it servlets and Java Server Pages
(JSPs) are executed. A servlet is designed to respond with dynamic content to client requests while a JSP is a technology that helps to dynamically generate HTML, XML or other types of documents. Other Java-based presentation tier technologies are template languages like WebMacro, Velocity or FreeMarker (Johnson 2003, p. 544ff).

3. The **business tier** contains the **business logic** of an application, i.e. ‘the code that fulfills the purpose of the application’ (Ball et al. 2006, p. 719). Business logic can be encapsulated in container-managed Enterprise Java Beans (EJBs), which are the Java EE components of this tier and which can not only be accessed from the presentation tier but also directly from the client tier. Their runtime is the EJB container and is ‘roughly equivalent to the ORB in CORBA [refer to page 7]. In EJB, however, the container is strictly a serverside entity. The client doesn’t need its own container to use EJB objects, but an EJB object needs to have a container in order to be exported for remote use’ (Flanagan et al. 2002, p. 177). Next to EJBs, also Plain Old Java Objects (POJOs) could be created for processing the business logic of an application (Fowler 2000).

4. The **integration tier** links a Java EE application to back-end resources. It is on this tier where Java EE applications interface with different technologies, e.g. ERP systems or databases. The needed functionality is offered by Java EE standard services like JDBC (refer to page 8), JMS\(^4\) or Java EE Connectors\(^5\). The application client container, Web container and EJB container offer JDBC and JMS services. Additionally the Web and EJB containers support Java EE Connectors.

5. The **resource tier** embodies back-end resources, e.g. ERP systems, Lotus Notes applications or databases.

**Application server**

A Java EE application server is a software engine that runs in its own Java Virtual Machine (JVM) and that offers the introduced containers and services needed for

---

\(^4\)Java Messaging Service (JMS) is a standard, vendor neutral API used to access enterprise message systems.

\(^5\)Java EE Connectors are a general architecture to connect to whole enterprise information systems.
Java enterprise applications (Sadtler et al. 2005, p. 24). It provides a runtime environment that allows components of the middle tier to integrate the front-end with resources of the back-end (see figure 2.6). Since application servers are implemented according to the Java EE specification, it is possible to deploy an application on different servers of different vendors. It is also possible to distribute an application across multiple application servers and computers. (Rymer 2005, p. 2) concludes: ‘The application server provides runtime services like access to the operating system, database, and network; administration and management; and scalability and reliability.’

Since an application server is related to the middle tier, it has to provide connectivity support to other servers like HTTP, LDAP (refer to page 6) or database servers. Figure 2.7 displays a possible configuration of an application server providing web and EJB containers. Some application servers (like Apache Tomcat) do not offer an EJB container.

The purpose of a HTTP server is to forward HTTP requests from clients to the application server and to return responses. It is possible to install the HTTP server together with the application server on the same machine or to install the HTTP server in its own (virtual) machine. The latter has the advantage that the HTTP server could be placed in a Demilitarized Zone (DMZ) while the application server would stay in the internal network.

A DMZ or ‘Perimeter Network’ provides a layer of security by preventing direct connections from the Internet to the internal network. On the edges of the DMZ are two firewall systems: The outer firewall between the Internet and the DMZ and the inner firewall between the DMZ and the internal network. If the HTTP server would be placed within these two firewalls, it could be accessed on port 80 from the Internet and then route the requests over its own port (e.g. 9080) into the internal network.
network and to its related application server. Like this the application server could be accessed from the Internet in a secure way.

### 2.2.3. Portal architecture

Basically it would be possible to implement the previously identified portal concepts (see figure 2.5) with a completely self-developed portal solution deployed into an application server. For this purpose, JSPs and servlets as well as more advanced Web frameworks could be used. Fortunately special (open-source or commercial) products exist that support portal development much more than general Web frameworks and that prevent reinventing the wheel over and over again.

At this point, the architecture of a portal is introduced. Its purpose is to offer standard functionalities that can be used from generic portal components. Up to now these functionalities were justified only with a focus on integrating existing data and systems: With the help of a portal, data and systems could be wrapped into a new, process oriented layer. Anyway, once a portal architecture is in place, it also can be used as a front-end technology for ‘normal’ multi-tier applications (Zörner 2006, p. 17). Seen like this, the portal architecture fits into the presentation tier (see figure 2.6) where it offers its presentation-centric features. These features and their related architecture are described in the following.

#### Basic components

*Portlets* are software components that extend a portal’s functionality and that can be seen as the main building blocks for portal applications. According to (Hepper
et al. 2005, p. 42), a ‘portlet is a Java-based Web component that processes requests from a portlet container and generates dynamic content. The content generated by a portlet is called a fragment, which is a piece of markup (e.g., HTML, XHTML, WML) adhering to certain rules. A fragment can be aggregated with other fragments to form a complete document, called the portal page.’

The way how a portlet is displayed on a page depends on its portlet mode and window state. On the one hand the portlet mode (Abdelnur & Hepper 2003, p. 35) enables a portlet to support different contexts of use. The obligatory view-mode is the most common mode since it is used for accessing the business functions of a portlet. The edit-mode enables administrators and users to set global or personal preferences and the help-mode displays general information about a portlet. On the other hand, the window state (Abdelnur & Hepper 2003, p. 39) determines the space a portlet can use to render its content. A portlet in the normal-state is displayed equivalent to others, in the maximized-state it will be the only visible portlet on a page and in the minimized-state only the title bar of the portlet will be seen (if existent).

A Portlet container provides a runtime environment for portlets, manages the lifecycles of all portlets within a portal and provides storage mechanisms for portlet preferences. Figure 2.8 displays a portlet container in the context of an overall portal architecture and shows how a request for a page is handled.

When a user accesses a portal page, the portal application queries its database for user related page data and according to this, determines the portlets of the requested page. Afterwards the portal application instructs the portlet container to call these portlets. The portlet container has access to the user’s preferences and calls relevant portlets via the portlet API. As a response, each portlet returns a markup fragment which the portlet container passes to the portal. It is the portal application that aggregates all markup fragments and composes the page returned to the user. (Hepper et al. 2005, p. 42)

**Portal server**

All the different pieces of the portal architecture are bundled within a portal server that technically leverages the functionalities of an application server (see page 16). Whereas an application server ‘is all about performance and scalability’, a portal server ‘is all about user experience (what we used to call UI)’ (Balaban 2006). The portal server is installed as an application of the application server; it is a collection
of servlets, JSPs and EJBs offering functionalities for UI aggregation (different user interfaces of different applications in one browser window), personalization (portal users can customize their own screen layout), Single Sign-On (SSO) and capabilities for front-end integration (data exchange between applications installed in a portal). These functionalities access additional resources like databases (e.g. to store personalization information) or LDAP servers (e.g. for user authentication). The portal server offers a portlet container that is implemented as a thin layer on top of the web container and that provides a runtime environment for portlets (Apache 2005).

**Portlet API**

The portlet Application Programming Interface (API) defines a contract between portlet container and portlet and offers the possibility to plug portlets into a portal application. The central part of this portlet API is the **Portlet** interface which every portlet has to implement directly or indirectly. This interface defines all of the portlet’s life-cycle methods. The **render()** method is e.g. such a life-cycle method that would generate a markup fragment.

Figure 2.10 shows a class diagram of a simple **HelloWorldPortlet** and it can be seen that the **Portlet** interface is implemented by **GenericPortlet**, an abstract adapter-class. In the case that the portlet container calls the **render()** method implemented in **GenericPortlet**, this method would set the window title of the portlet and call the **doDispatch()** method. Then, if the window state would not be ‘minimized’, the appropriate **doXXX()** method would be called depending on the
actual portlet mode.

The whole life-cycle of a portlet consists of three phases (Abdelnur & Hepper 2003, p. 21). At first the portlet is initialized and taken into service, then it responds to client requests and finally the portlet is taken out of service. ‘Client requests are triggered by URLs created by portlets. These URLs are called portlet URLs. Portlet URLs may be of two types, action URLs or render URLs.’ (Abdelnur & Hepper 2003, p. 24)

If a user clicks on an action URL, the portlet container sends an action request to the portlet where the action URL was clicked and calls its processAction() method. After this method has finished, the portlet container sends render requests to all non-cached portlets on the same page and calls their render() methods in no specific order. When a user clicks on a render URL (or when a new portal page is accessed), the portlet container calls all render() methods of all non-cached portlets on the page.

2.2.4. Development standards

The following section introduces relevant standards in the area of portlet development: The Java portlet specification (JSR-168) and Web Services for Remote Portlets (WSRP).
JSR-168

Before there was a standard portlet API, every portal product on the market supported its own custom API for portlet development. This meant for software developers that they had to develop different portlets if they wanted to reach a broad customer base and for organizations that they could not change their portal solution without rewriting their existent portlets.

The solution out of this ‘vendor lock-in’ (Zörner 2006, p. 19) was a common standard that describes how portal applications and portlets can interact with each other. In the Java world, new standards are defined via the Java Community Process (JCP). IBM and Sun together submitted the Java Specification Requests (JSR) 168 that targeted the design of a standard portlet API. The first version of the Java portlet specification was adapted in October 2003, one year later Apache Pluto 1.0.0 was released. Pluto is the reference implementation of the Java portlet specification.

All of the JSR-168 topics relevant for this paper have been already introduced in the section about the portal architecture (see page 18), which is based on JSR-168. Despite of this, it should be noticed that the portal architecture introduced on figure 2.8 contains ‘portal extensions’. These extensions are dependent on the portal product and should normally not exist if JSR-168 would have covered all needed functionalities. Unfortunately areas like inter-portlet communication are (not yet) addressed by the standard and therefore portal vendors have developed their own solutions. Anyway, the Java portlet API 2.0 is under development by the JSR-286 expert group.

WSRP

Web Services for Remote Portlets (WSRP) is a standard defined by OASIS (Organization for the Advancement of Structured Information Standards) that was accepted in August 2003 (Kropp et al. 2003). While applications can use Web services to expose a programmatic interface (see page 9), they can use WSRP to expose a presentation layer that can be plugged into other applications (without extra programming effort on the client side). Similar to ‘normal’ Web services, WSRP-coupled systems exchange XML defined SOAP messages. The difference is that these SOAP messages are not structured according to a custom WSDL definition, but according to a standard WSRP definition.

For a complete list of topics addressed by JSR-168, refer to (Sun 2003, p. 2).
In comparison to JSR-168, the aim of WSRP is to make it possible for local portlets to become ‘pluggable visual components to the World Wide Web’ (Hepper et al. 2005, p. 224). Like this, applications can be shared across network boundaries, between different organizations or different technologies (e.g. Java and the .NET framework).

Figure 2.11 shows the differences between function- and presentation-oriented Web services consumed by a portal. The upper part presents a custom developed portlet that calls a local proxy (an interface to a Web service) which in turn invokes the remote service operation. On the remote server, the Web service function executes and the result is returned to the client.

The lower part of the figure presents the same scenario but for a presentation-oriented Web service. In this case, there is no need for a custom developed portlet since WSRP defines a common contract to all WSRP services. This is the reason why a generic proxy portlet can be used that communicates directly with the Web service of the remote application. From the perspective of the consumer, there is no need to develop service-specific code. Changes to the application logic or future changes of the WSRP specification are directly reflected with no need for local changes (despite of an update of the generic proxy portlet).
3. Concept and Discussion

This is the main chapter of this paper. In the beginning, a strategy is identified for the integration of Lotus Notes databases into a portal. Based on this, requirements for a generic application architecture are derived that address both the Lotus Notes and the portlet application. Then this architecture is designed according to these requirements and its different layers are analyzed in detail.

3.1. Integration strategy

The foundation of the architecture is made in this section since the architecture design is strongly related to the used integration strategy. This paper defines integration as an aggregation of data (and if possible functionality) of a Notes application into an enterprise portal. An integration strategy is an approach how to integrate a Domino application into a portal. This strategy is dependent on both the level of integration and the integration technology (see figure 3.1).

At the beginning, portlet patterns and their related technologies are introduced and then evaluated from the technical and business perspective. As a result, an integration strategy is identified and summarized with a decision tree that helps to answer the question: ‘Which technology should I use to integrate my Notes application into a portal?’

3.1.1. Patterns and technologies

‘Patterns are about communicating problems and solutions. Simply put, patterns enable us to document a known recurring problem and its solution in a particular context, and to communicate this knowledge to others. One of the key elements in the previous statement is the word recurring, since the goal of the pattern is to foster conceptual reuse over time.’ (Alur et al. 2003, p. 9)
Patterns in the field of software engineering were popularized in the book ‘Design Patterns: Elements of Reusable Object Oriented Software’ (Gamma et al. 1995) and since then have become a common decision topic in software development teams. Thus, it is not astonishing that patterns can be found also in the area of enterprise portals. (Tulisalo et al. 2003, p. 19) identifies so called ‘portlet patterns’ and introduces them as follows: ‘Portlets can be as simple as data display windows into existing applications, or as advanced as a replacement for complex workflow applications. They also can have a different level of integration with the portal. Portlet patterns will help to classify what type of integration level should be used; this information is relevant when deciding what integration technique to use.’

In the following, the different portlet patterns are described at a glance and used as a classification scheme for the technologies which are available to integrate Domino applications. This part is mainly based on (Tulisalo et al. 2003, p. 19ff) and (Bergland et al. 2005, p. 42ff).

**Link pattern**

The most simple form of integration is to use a portlet that addresses a Notes application via a HTML link. With the help of the URL element the application could be triggered from the UI of the portal but would launch outside of the portal (Tulisalo et al. 2003, p. 19). It either could launch in a Notes client locally installed on the user’s computer or in its own web browser window (if it is a web-enabled application). A portal might use this solution to offer role-based or personalized bookmarks.

**Display pattern**

This pattern specifies a portlet that displays data from Notes applications. Furthermore, if users would want to interact with this data or use related functionalities,
the link pattern could be used to launch the native Notes application. The following list presents different possibilities how this pattern can be technically applied.

- One way is to use the *inline frame* (IFrame) element of HTML. This approach is very close to the link pattern since it is just a special way of how to link to a Notes application. It also uses a standard HTML element, but in comparison to the link pattern an IFrame can only address a web-enabled application because it directly embeds the application within a portlet. It is the task of the Domino server to render the embedded HTML page. (Tulisalo et al. 2003, p. 43)

- *Clipping* (or screen scraping) means that a subset from a HTML page of a Lotus Notes application is extracted and then embedded within the markup of a portlet (Tulisalo et al. 2003, p. 43). If links within this markup should act portal friendly so that selecting a link does not result in a new browser window, a portlet has to act like a reverse proxy for a connected Domino application and modify links via regular expressions.

- A portlet could also use HTTP to access *XML-formatted data* stored in Domino Views or Documents (see page 9). This data can be processed in the presentation tier where it could be rendered with the help of an XSL Transformations (XSLT) style-sheet. Another option would be to use this data as an input for the business tier.

**Integrated pattern**

This pattern describes portlets that 1) are responsible for the rendering of data and 2) allow a user to (indirectly) interact with a Domino application via a portal interface. The aim behind this pattern is to make lightweight and commonly used functionalities accessible in a portal. The full range of interaction possibilities is only available via the Notes client or a dedicated Web browser interface.

- A portlet application could leverage the *Domino Java API* to access, manipulate and create data and functionalities of a Notes application. The classes of this API use CORBA (see page 7) to access Domino Object Model (DOM) back-end classes on a Domino server. (Tulisalo et al. 2003, p. 247)
• *Domino JSP tag libraries* is a technology that uses Domino’s Java API and encapsulates it in simple to use XML tags. This approach supports access to Notes applications directly from JSP pages (refer to page 16).

• Data residing in a Domino server also could be accessed from a portlet application via *JDBC* (introduced on page 8).

• *Web services* are another option (see page 9 and figure 2.11).

• *Portlet Builders* are proprietary tools that ‘provide the capability to rapidly create portlets that can access and manipulate Domino applications (...)’ (Bergland et al. 2005, p. 591). It depends on the tooling which of the above listed technologies are used.

**Migrated pattern**

A migrated portlet offers the full functionality of a Notes application via a portal interface. An approach like this would usually result in a redesign of the application (Tulisalo et al. 2003, p. 21) and might include a switch from Notes to a different back-end system.

### 3.1.2. Technical view

Up to now it was described that the deeper the integration level, the more data and functionality of the Notes application is accessible directly in the portal interface. Seen like this, the link pattern describes the most *shallow* and the integrated pattern describes the *deepest* integration level. For the user experience this means that deep integration levels ‘mask’ the back-end system so that a user does not recognize used resources. Thus, the overall look and feel of a portal is more consistent as its UI depends mainly on globally defined themes and skins. Additionally, a deep integration increases the amount of portal features that can be leveraged by the integrated application, e.g. caching, SSO or inter-portlet communication.

On the other hand, shallow integration levels, as described by the link and view pattern, offer a more quick and easy way to integrate already Web-enabled Notes application into a portal. Like this, Lotus Notes RichText, i.e. rich formatted data created with the Lotus Notes client, is rendered natively as HTML by the Domino server.
Figure 3.2.: Common portal topology

This and the next section analyze the different shallow and deep integration levels from the technical and the business view. This section is focused on integration technologies.

**HTML link**

Even though a link is the most easiest integration possibility, it has to be verified that the linked Domino resource can be directly accessed by the portal user. In a common portal topology\(^1\) (see figure 3.2) both portal and Domino servers are placed in the internal network for security reasons (refer to page 17). Thus, it should be noticed that a web-enabled Domino application can only be accessed from the Internet via a reverse proxy.

**HTML IFrame**

An IFrame is technically very similar to a HTML link and therefore the integrated resource also needs to be accessible from all possible locations of portal users. The main advantage of an IFrame is that it displays the Notes application within the context of a portal and relies at the same time on Domino’s HTTP task for rendering.

\(^1\)Topology, in the context of this paper, refers to the mapping of software components to available computer hardware.
This is an important feature because ‘the only way to convert RichText into HTML is to use the Domino HTTP task.’ (Tulisalo et al. 2003, p. 266)

A limitation of this solution is that UI requirements of a portlet differ from those of a normal web site since a portlet is only one part of a whole page. This might result in the development of a new portal-related UI for the integrated Notes application. Furthermore, IFrames that contain applets will not work correctly as for every request of a portal page the applet instance is stopped and a new one is built (Tulisalo et al. 2003, p. 74). According to (Hepper et al. 2005, p. 95), IFrames ‘undermine the whole portlet principle as the portlet API is just tunneled. Thus IFrames should only be used for very special cases, like surfacing legacy applications.’ These concerns are also expressed in the JSR-168 specification as it explicitly forbids IFrames because they ‘might impact content generated by other portlets or may even break the entire portal page’. (Abdelnur & Hepper 2003, p. 113)

**Clipping portlet**

Portlets that use screen scraping try to overcome the disadvantages of IFrames by offering a functionality that ‘clips’ a part of a web-enabled Domino application and pastes it into a portlet. Like this, the data itself would still be rendered by a Domino server but a part of the generated HTML would be integrated into the markup generated by the portlet.

Unfortunately clipping does not always work as intended since header information gets lost so that JavaScript does not execute properly (Tulisalo et al. 2003, p. 80). Even if nearly a complete HTML site would be clipped, other problems like links to other parts of the application or broken images remain. Broken images are caused because Domino addresses them with relative URLs absolute to the Domino server (Tulisalo et al. 2003, p. 267). This could be solved by modifying Domino’s HTML code with the help of regular expressions. In this case all requests should be channeled through the portal server where the HTML code would be parsed. Then the portlet would act like a reverse proxy to the Domino server and could also offer caching functionalities (Bergland et al. 2005, p. 123). This would have the side-effect that a portal user does not have to have anymore direct access to a Domino server.

IBM provides a Domino Application Portlet (DAP) that offers a ‘proxied’ IFrame and can be downloaded free of charge from IBMs Workplace Solutions Catalog. It is a generic portlet that uses custom rules to solve both URL and image problems.
Rules have to be configured individually for every addressed Domino application. This portlet does until now not support JSR-168 and thus can only be deployed on an IBM WebSphere Portal (Delahunty et al. 2004).

Links, IFrames and clipping portlets are the only ways to access HTML-formatted Lotus Notes RichText because they rely on Domino’s native rendering capabilities. Unfortunately, there is no proper JSR-168 compliant solution addressing this problem and therefore ‘rich text integration into portlets is a complex and potentially challenging procedure’ (Tulisalo et al. 2003, p. 269).

**XML**

Basically, there are two possibilities how data stored in Notes applications can be accessed via XML. The first possibility is to design *data-oriented* Views or Forms with a hardcoded XML structure (Tulisalo et al. 2002, p. 749ff). Like this, all data types (despite of RichText and file attachments) could be accessed via HTTP. This approach has the benefit that Notes applications can easily expose their data according to a custom defined XML interface and without huge development effort. On the negative side the amount of supported data types is limited.

The second possibility leverages Notes’ native support of *document-oriented* DXL (see page 9). This makes it possible to retrieve a DXL document of a standard web-enabled View just by adding the URL parameter ‘ReadViewEntries’ to the request. Since Views cannot contain RichText, this solution is still limited but it is possible to develop a custom Agent that could return a DXL document of a requested Notes Document over HTTP (Balasubramanian 2005). Like this you can get a handle to DXL defined RichText and to Base64-encoded attachments.

One disadvantage of this second solution is that a Domino Web Agent does not perform well for high volume requests (Nielsen et al. 2001, p. 34). The cause is that Agents (unlike servlets, see page 16) have to be loaded and unloaded for every invocation and do not stay in memory. Additionally, it has to be considered that IBM does not offer any standard XSLTs which could be used to transform DXL into other formats. If HTML is needed, then a custom XSLT has to be developed which might be a complex undertaking related to the various RichText elements.

**Domino Java API**

The main asset of the Domino Java API is that it offers the most flexible possibilities in comparison to all other integration technologies. With the help of this API fully
customizable portlets can be developed ‘which are very fast and scalable in accessing Domino’ (Tulisalo et al. 2003, p. 226). In the area of RichText it is possible to export single Notes Items or whole Notes Documents as DXL. Developers familiar with LotusScript can use their existing knowledge since the same back-end Domino Object Model (DOM) is used. Furthermore, all benefits of the Java programming language can be leveraged (see page 14).

On the negative side it should be noticed that there needs to be an IIOP connection between portal and Domino server requiring its own port. Also the same limitations related to DXL and big compound documents are valid for the Domino Java API. (Tulisalo et al. 2003, p. 274) additionally states, that ‘DIIOP sessions to Domino servers require non-trivial amounts of time to create and destroy. Often, in high-throughput scenarios, portlets must wait for a session object to become available, creating a bottleneck in the flow of the application.’ This bottleneck can be disposed with the help of object pooling so that many portlets can share the same resources.

**Domino JSP tag libraries**

The Domino JSP tag libraries rely on Domino’s Java API and thus the same limitations apply. But in comparison to the Domino Java API, which supports data access from the integration tier, tag libraries propagate data access from the presentation tier to simplify application development. This approach is a ‘poor design choice’ (Johnson 2003, p. 276) as the presentation tier is the logical architectural layer concerned with presenting a Web interface and not for executing business logic or data access. The Domino tag libraries compromise the advantages of a tired architecture (see page 14) since changes in the back-end system should never break a web interface.

According to Rod Johnson (Johnson 2003, p. 276f), there are various disadvantages related to this technology.

- **Error handling**: If a problem (e.g. a communication failure with the Domino server) occurs, the system is committed to render one particular view. At best there is a generic error page presented. At worst, the buffer was flushed before the error was encountered and a broken page is the result.

- **Exposition of business logic**: It is not possible to expose the business logic contained in a JSP page. This might be needed if a portal should serve also
non-web clients or web service clients.

- Unit testing: Since there is no business interface of a JSP page, unit testing cannot be performed.

- DXL: It is not possible to access Notes objects as DXL and to transform them with an XSLT stylesheet; thus, it is not possible to access simple structured data in RichText Items.

The conclusion is that data access in productive systems should never be performed from JSP pages since they are view components. If there is any place for data access from JSP pages using tag libraries, it is in trivial systems or prototypes (Ball et al. 2006, p. 168).

**Lotus Domino driver for JDBC**

In general, JDBC is a commonly used technology for data access in Java EE environments. This is due to the fact that most Java EE applications access data from relational databases; therefore, mapping between object model of the application and the RDBMS is required. This mapping is not trivial because of the impedance mismatch (Johnson 2003, p. 255) between OOP and the relational model in database technology.

Furthermore, the ‘JDBC API is relatively low-level, and using it is error-prone and requires an unacceptable volume of code (…)’ (Johnson 2003, p. 311). This is the reason why applications should rely on a higher level language like SQLJ or should use an Object-Relational (O/R) mapping framework like Hibernate.

After these requirements to the JDBC API have been introduced, it is hard to justify its use since the Domino Java API offers a possibility to directly access the DOM. The Lotus Domino driver for JDBC artificially converts Domino’s object model into a table model, which in turn has to be (manually) remapped to a new object model probably very similar to the original one. Another limitation is that via JDBC only data, but not functionality, of Domino databases can be accessed and that it is also not possible to get a handle to RichText (or its DXL representation).

**Web service**

Also in regard to web services, the Domino Java API should be favored. The reason is that web services require the development of custom web service providers in
the Domino application. Another reason is the amount of data exchanged between portal and Domino server. As already introduced (refer to page 7), web services exchange XML defined SOAP messages while CORBA’s ORBs exchange shorter, binary message formats.

On the other hand, the strength of web services is that their ‘transport protocols run over HTTP and are more firewall friendly’ than IIOP (Johnson 2003, p. 35). Therefore, if portal and Domino servers can only communicate over HTTP this technology should be considered. Also if a Domino application should be integrated into a portal that is based on a non-Java technology, e.g. the .NET framework, the use of web services might be attractive.

Portlet builder

Different portlet builders, like CONET Knowledge Director or IBM WebSphere Portlet Factory are available to support rapid development of portlets that integrate custom Notes applications. This paper is of the opinion that instead of proprietary and potentially complex (O’Donnell 2006) tools some of the previous introduced technologies should be favored because they better support sound design principles, flexible applications, and do not lead to a vendor lock-in.

Instead of portlet builders, open-source frameworks like Spring should be used to simplify Java EE and portlet development.

3.1.3. Business view

After different integration technologies have been discussed from the technical view, this section analyses the different integration levels, i.e. portlet patterns, from the business view.

Portlet patterns have been previously classified in shallow and deep integration levels (see page 27). This classification scheme is sized here again but under different labels: A shallow integration realized with the link or view pattern refers to an application-oriented portal integration, while a deep integration through integrated or migrated pattern refers to a data-oriented integration.

Application-oriented portal integration

According to Thorsten Gurzki (Stelzer et al. 2004, p. 37f), application-oriented portal integration embeds or links a Web interface of a back-end application into
a portal page. This approach usually makes it necessary that the integrated application gets a new, adjusted UI fitting to the overall layout and design of the surrounding portal. Since such a back-end system then is providing its own portal interface, the portlet application directly correlates to the back-end system: The back-end system basically is the portlet application.

This approach is limited as the majority of features offered by a portal architecture is bypassed (e.g. inter-portlet communication, personalization). The only task of application developers is to tune the UI of the native application and then their job is done. It has to be questioned if this kind of integration delivers a value for an organization and can justify a portal project. As previously introduced (see page 12), portal systems should target core business processes so that they can deliver a measurable return on investment (ROI). Since one single process can be related to a various amount of different applications (Stelzer et al. 2004, p. 126), employees still have to interact with every individual application and search for the functionality needed in the actual context. Used like this, the portal can only enhance the accessibility to different, still independent systems.

**Data-oriented portal integration**

In comparison to application-oriented portal integration, data-oriented integration can help if one portlet applications should have access to different back-end systems (Stelzer et al. 2004, p. 37f). This can be realized by connecting enterprise
information systems (EIS) in the resource tier to a portlet application placed in the middle tier (see figure 3.3). Thus, portlet applications can interact with different data pools and then it is possible to build a new process layer between user and back-end system.

Business processes are often divided into application-related chunks due to the fact that processes leverage more than one data pool. The result of this is a technology-caused process fragmentation which could be prevented if a process oriented middle-layer is used. Sebastian Grimm states (Stelzer et al. 2004, p. 286), that such a layer delivers crucial benefit for an organization and a portal project because the portal is brought into line with business processes and shields back-end systems from users. Employees only retrieve those information needed in the context of the current process while related data is aggregated from different and independent data pools.

It can be concluded that processes that span different back-end systems can be simplified with the help of data-oriented integration in the business tier. Application-oriented portal integration fails to solve this problem and therefore will be replaced in the long term by a data-oriented approach. (Stelzer et al. 2004, p. 292)

3.1.4. Conclusion

In this section, the results of previously evaluated integration technologies and integration levels are summarized and merged into an overall strategy. It is explained 1) why the integration strategy strongly depends on the structure of data within a Notes Document, 2) with which technology already existing Documents should be integrated and 3) how applications could be designed so that they support a deep and data-oriented portal integration.

Structured and unstructured data

One result of the evaluation of integration technologies was that it is challenging to integrate RichText into a portal due to technical constraints (see page 30). This is the reason why in the following the structure of data stored in Notes applications is further analyzed.

Structured data is an enforced formation of primitive datatypes (like plain text, numbers or dates). Notes databases can store structured data in 1) primitive Items of Note objects or in 2) View indices. Opposed to this, unstructured data has no datatype definition and is stored without structure at an atomic level. The RichText
Item is the only Item that can store such unstructured data. It can contain simple or rich formatted text, special UI elements like buttons or tabbed tables, application logic, embedded graphics, attached files and Object Linking and Embedding (OLE) objects. Thus, a RichText Item has the potential to store very different kinds of data at the same time.

Notes’ semi-structured compound Documents are containers for structured and unstructured data (Fischer et al. 2002, p. 253). This means that these Documents can contain both structured data in primitive Items and unstructured data in RichText Items.

Additionally, this paper introduces simple Documents. Simple documents are special semi-structured compound Documents that store only structured or ‘simple unstructured’ data. Defined by usage, a simple Document could contain all types of structured data. It even could hold RichText Items, but only if containing simple formatted text. In regard to attachments, a file should be ‘outsourced’ into a RichText Item of another Document and then referenced. This should keep the size of a simple Document low.

In the following part, structured and unstructured data (stored in semi-structured compound Documents, simple Documents or View indices) are brought in line with previously evaluated integration technologies (page 27ff) and integration levels (page 33ff).

Integration of unstructured data

Generally spoken, unstructured data is intended for publishing. This kind of data can only be integrated into a portal if the Domino HTTP task is leveraged and a shallow, application-oriented integration approach is chosen. This is the only possibility to generically render the content of RichText Items as HTML (see page 30).

Additionally to the structure of data, the size of a Notes document has to be considered. Documents might be so big that they cannot be programmatically handled with high-performance from a remote portal. If structured data within heavyweight compound documents needs to be accessed, this problem could be bypassed by storing the data in lightweight Notes Views. But problematic are nicely formatted texts spiced with file attachments or embedded images. Also this problem can be solved sufficiently only by leveraging Domino’s native rendering capabilities.

Thus, the first choice to integrate unstructured and potentially huge-sized com-
pound Documents into a portal should be IFrames. Although officially forbidden for JSR-168 portlets (see page 28), an IFrame can be used as a window to a Domino web application. Like this, a portal user can view the unstructured data stored in Notes applications without leaving the portal environment.

It was mentioned that this shallow integration approach means that a Notes application has to offer a customized web interface and that the related Domino server needs to be directly accessible for portal users (refer to figure 3.2). Furthermore, a general limitation of Notes is that it does not support the possibility to edit unstructured data created with the Notes client via a Web interface or vice versa.

The business view emphasizes that such an application-oriented portal integration fails to create business value if simply an existing Notes application is taken and embedded within an IFrame. But if unstructured data stored in Lotus Notes is related to a business process and therefore needs to be displayed within a portal, business value is delivered even if in general the data-oriented approach should be preferred (see page 35). This is due to the fact that structured and function-oriented data of a cross-system business process needs further processing in the business tier, where it can be handled together with data from other systems. In comparison, unstructured data is intended for publishing and therefore can be directly published into the client tier.

**Integration of structured data**

Structured data can be deeply integrated into a portal. If read access is sufficient, then data could be accessed with the help of XML as described on page 30. In the case that 1) read and write access are needed and that 2) the size of related Documents is low (e.g. by using simple documents), the Domino Java API and CORBA should be utilized because this offers the possibility to directly work with the DOM. It has been explained why the use of the Domino JSP Tag Libraries is a ‘poor design choice’ (page 31) and that JDBC demands elaborate object mappings (page 32). Web services are an alternative technology to the Domino Java API and should be considered if a Java EE portal server is not used or if portal and Domino server can only communicate over HTTP (page 32).

From the business perspective, this deep integration of structured and function-oriented data should be the main driver of a portal project since the portal is brought into line with cross-system business processes and can shield individual back-end systems from users.
Summary

In the end of this section it can be summarized that a portal project should focus on cross-system business processes. In this case, an enterprise portal can provide a process-oriented layer that consolidates different back-end systems; therefore, it can improve productivity and quality of labour.

Figure 3.4 displays the integration possibilities for Lotus Notes in this scenario and visualizes two different integration approaches. Structured data stored in Notes databases and related to the function-oriented part of a cross-system business process should be deeply integrated into a portal so that data of all related back-end systems can be handled together in the business tier. Unstructured data stored in Notes is intended for publishing and therefore can be shallowly (and preferably context related) integrated into a portal. It was noticed that unstructured data can only be displayed in a portal interface and requires for editing a rich Lotus Notes client.

In respect of introduced portlet patterns (see page 25), the integrated pattern is the one that will deliver the most beneficial results. This pattern allows to deeply integrate commonly used and process related functionalities while more rarely used functionalities (e.g. for administration purposes) still remain within the Notes application. Also the view pattern might be useful if process related and unstructured information should be displayed within a portal. The migrated pattern should be avoided since re-development is a ‘cost-killer’ (Hepper et al. 2005, p. 221).
An example of a reasonable portal project could be the process of hiring a new employee. Triggered by the human resources department, personal data and accounts of new employees need to be created within different systems of an organization. A portal could be the interface to this process and manage the different subsystems in the back-end. From the Notes perspective, this would mean that a new user is created in the Domino directory and maybe also in other organizational databases or Notes applications. Next to this integration of structured data, a Notes application could as well be used to display help-documents that contain RichText Items and that assist users of the portlet application.

If a new Notes application is developed from scratch to be integrated into a portal, it should be designed so that it enforces the structuring of data as much as possible. Lightweight simple documents should be preferred instead of heavyweight documents. A file attachment should be stored in its own container document and then referenced. These guidelines are implemented in a sample application which will be presented in the ‘implementation’ chapter of this diploma thesis. But beforehand, the identified integration strategy is used to design a generic architecture of an application that integrates structured and unstructured data into an enterprise portal.
3.2. Architecture

‘A software architecture is a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and non-functional properties of a software system. The software architecture of a system is an artifact. It is the result of the software design activity.’ (Buschmann et al. 1996, p. 384)

This section of the diploma thesis is about the design of a cross-system software architecture used to integrate document-centric content stored in Lotus Notes into a Java based and process driven enterprise portal. The architecture is build on the integration strategy which has been previously identified and addresses both portlet and Notes subsystems.

In the beginning of this section, functional and non-functional requirements are discussed, and then the architecture is designed according to them. After the architecture has been explained at a glance, its different layers are analyzed in detail.

3.2.1. Requirements

Requirements of the application architecture can be separated in functional and non-functional requirements. Functional requirements define the functionality and features the architecture should provide while non-functional requirements describe its properties.

Functional

The functional requirements can be directly derived from the already identified integration strategy (see page 24ff). The main goal is to deeply integrate structured or function-oriented data into a JSR-168 compliant portal while unstructured data intended for display should be shallowly integrated. The subsystems of the application have to address this task in the following way.

1. Subsystem: Notes application
   - Since structured data can be deeply integrated into a portal, the Notes application needs to enforce structuring as much as possible. Data to be
deeply integrated should be available in back-end Views or simple Documents and it should be possible to view and edit this data despite of the subsystem which has been used for its creation. If new applications are designed, file attachments should be stored in own container documents and then referenced within related simple documents. Like this, data processing and handling is speeded up, not only for the remote portlet application but also for users of the rich Notes client.

- Often a Notes Document is related to other Notes Documents and has to be seen in their context. A relationship like this is displayed and managed in Notes Views that either are flat- or tree-structured. The Notes application should support a service that uses the Domino Java API and frequently reads tree-structured Views, caches their data and makes it available within a suitable Java data structure. Flat structured Views do not need such a service since they sufficiently can be accessed with the help of the DOM.

2. Subsystem: Services
To make the architecture modular and better maintainable, services should be used to offer generic functionalities for all portlet applications that integrate Notes into a portal.

- Object pooling needs to be supported so that many users or portlets, which use the Domino Java API, can share the same resources (see page 31). By implementing a pool as a portal-wide service, all portlet applications running in the same portal server can share these bottleneck resources.

- The architecture should offer a navigation tree service that accesses Notes back-end Views and provides their structure and data to portlet applications.

3. Subsystem: Portlet application

- The integrated pattern should be used for a function- and data-oriented integration of structured information. It has to be possible that the portlet application can read from or write to the Notes application no matter by which subsystem the original data has been created. File up- and downloads need to be supported and images stored in a Notes database
should be displayed in a portlet application. The technology constraint is that either XML (and HTTP) or the Domino Java API (and CORBA) should be leveraged.

- For the integration of unstructured information in RichText items, the view pattern should be applied and then realized with HTML IFrames. Like this, it is possible to display (but not to edit) unstructured information within a portal. There might be different solutions to store unstructured information via a Web interface, but they all neglect the rich editing capabilities of the Notes client and do not incorporate unstructured data already existing in a Notes database. Thus, unstructured data should only be displayed in a portal and edited with the help of the Notes client.

- Notes’ security model (e.g. Access Control List (ACL) and Reader Fields) has to be leveraged in an efficient way. Otherwise, security related to the portlet application might be handled in higher tiers with the features provided by Java EE and portal servers.

Non-functional

The following non-functional requirements are not unique to the architecture designed in this chapter but apply to all enterprise applications. They are based on (Johnson 2003, p. 16f).

Enterprise software is important to an organization and therefore should be robust. Users expect a reliable and bug free system that also executes with decent performance. Scalability can support this since it addresses the potential of an application to deal with increased load.

A cross-system portal application is a complex software and thus object-oriented (OO) design principles should be applied. These principles encourage the use of proven design patterns and can help to avoid unnecessary complexities. Complexity should be kept low because it increases the costs throughout the software life-cycle; on the other hand, it also needs to be ensured that requirements are not simplistic or naive. Another benefit of OO design is that it promotes the reuse of existing code. This leads to a clean code-base and fosters the use of already existing features of a portal or Domino server.

The highest costs during a software life-cycle are caused by maintenance. Thus, it is important that maintainability is already considered when a system and its archi-
tecture are designed. Next to maintainability, also extensibility is largely dependent on a clean design. It should be ensured that each component of the application has a clear responsibility, and that maintenance is not affected by tightly coupled components.

Testing has an important role in the development process and the architecture should be designed so that it eases testing. This is especially true for the portlet application. It should be possible to test huge parts of the application independently from the portal server. This speeds up the development process since deployment to a portal server is time consuming and might be error-prone.

3.2.2. Overview

In this section an overview of the developed architecture is given and its basic concept is explained.

Java EE provides many architectural choices and offers different component types (like servlets, portlets, JSPs, EJBs) that have to be considered. Additionally, portal and application servers provide a variety of vendor-dependent services. This paper cannot offer a generic architecture that would fit in all possible scenarios. As already introduced (see page 38), the portlet application should be a process related layer in the middle between users and different back-end systems. Such a cross-system process application would not only interact with Lotus Notes, but maybe also with relational databases or enterprise systems like SAP. Therefore, the entire architecture is not only dependent on Notes but should include also other (yet unknown) systems.

(Johnson 2003, p. 28ff) identifies four main Java EE architectures and labels them as follows:

1. Non-distributed architectures
   a) Web application with business component interfaces
   b) Web application that accesses local EJBs

2. Distributed architectures
   a) Distributed application with remote EJBs
   b) Web application exposing web service interface

The key point is that these four architectures differ only in the manner that they implement and access business logic. The architectural parts that are concerned
with the integration of Notes applications can be generically applied to all four architectures. To make the architecture discussion more concrete, the ‘web application with business component interfaces’-architecture is used throughout this paper since it is ‘a simple, performant architecture that meets the requirements of many projects. Although it does not use EJB, it still delivers a clean separation between business objects and web-tier components. A layer of business component interfaces exposes all the application’s business logic to web-tier components.’ (Johnson 2003, p. 41)

Such an architecture is displayed in figure 3.5 and its main idea is to merge already introduced two-tier architectures (see figure 2.1) and multi-tier architectures (see figure 2.6) into a new, mixed architecture. This is necessary since both a deep and a shallow oriented integration has to be supported. Structured information stored in Notes databases is deeply integrated and therefore a multi-tier architecture is used. On the other hand, a two-tier architecture had to be chosen since there might be the need to display unstructured information via IIFrames.

Figure 3.5.: Integration architecture
The different layers of this architecture are analyzed in the remaining sections of this chapter.

### 3.2.3. Resource tier

This section addresses the architecture of the Lotus Notes application and analyses how data to be integrated into a portal should be prepared and handled.

**Structured data**

From the architectural point, Notes Views are the main design elements that should be used to build clean data interfaces between business and resource tier. Views have the benefit that they are cached by the Domino server and that only a defined selection of Documents and structured data is exposed. On the negative side, the size of Notes Views should be kept low for performance reasons.

Structured data in a Notes database is either stored as primitive datatypes in Documents or in View indices. If read access is sufficient, then data-oriented (see page 9) XML Views or Forms can be used to embed this data in a hard-coded XML structure accessible over HTTP. This approach would only require the development of additional View or Form design elements and has the benefit that standard solutions for caching can be used since HTTP is the transport protocol.

In the case that also write access is needed, the Domino Java API should be used to access data stored in View indices and to get a handle to complete Notes Documents. The most efficient way to retrieve Documents is the method `getDocumentByUNID()` of the `Database` class (Nielsen et al. 2000, p. 76); thus, it is recommended to use Domino for providing a selection of Documents within a View and then to access View data and universal identities (UNIDs) of related Documents. The UNIDs can in turn be used to get and edit existing Documents. A limitation is that the Document size has a huge impact on Document handling since a complete Document is always transferred to a remote portal if `getDocumentByUNID()` is used. Thus, simple documents should be preferred and the portlet application should cache data to avoid remote calls to Domino.

**Unstructured data**

Unstructured data intended for publishing and stored in RichText Items of a Notes Document should be shallowly integrated into a portal. For this purpose HTML
Views and Forms can be developed that are comparable to XML Views and Forms. The difference is that the Domino HTTP task automatically transforms unstructured data contained in RichText Items into HTML.

If simple unstructured data should be deeply integrated into a portal, simple Documents and DXL have to be utilized as displayed in figure 3.6. The left side shows the process of editing data in RichText Items within a Notes client. When the Postsave event is triggered, a temporary Document should be created and all RichText Items that need to be converted into HTML should be copied into it. Then a DXL Exporter can be used to create a DXL representation of this temporary Document which can be transformed into a HTML snippet with the help of a custom XSLT. The produced HTML snippet can finally be stored in its own Item. Like this, the portal can directly access and display this HTML without any need for further processing.

In the case that simple structured data should also be edited in the portal, a similar procedure can be applied. A modified HTML snippet can be transformed into DXL with the help of a second XSLT and then a temporary Notes Document can be created by using the DXL Importer. The relevant Items of this Document can afterwards be copied to the original Notes document. This makes it possible to edit simple structured data both in the portal and with the Notes client.

**Contextual information**

In general, contextual information is a collection of environmental parameters used to describe the situation of an entity. Related to Notes, contextual information is about Document relationships. Views offer mechanisms to express Document
relationships because Documents can be ordered in one or two dimensions. Flat structured Views can sufficiently be handled by standard functionality of the Domino Java API but tree-structured Views might require more elaborate calculations, e.g. if all descendants or siblings of a document are requested.

Figure 3.7 addresses this problem and displays the design which is leveraged by the navigation tree service described later in this chapter. A front-end View displays Documents in a tree structure to users of the Notes client. Users have the possibility to individually position single Documents at any requested place in the tree. A second, flat-structured View contains the same documents and is hidden from users. This View has four columns: The first column is a sorted one and contains OrderLevels of Documents. An OrderLevel is a string that expresses a Document’s position in the View tree. The second column stores the numbers of a Document’s ancestors, the next column holds Document titles and the last column displays the UNIDs. This data can be leveraged by the navigation tree service to build a Java tree representation.

If users do not need to change the position of individual Documents in the View tree, the back-end View can be skipped and the navigation tree service could directly access a front-end View. Anyway, it is recommended to integrate only data in back-end Views, because then a change of the UI of the Notes application would not brake the integration tier.

3.2.4. Integration tier

In this section, the Data Access Object (DAO) pattern is introduced together with portal-wide services that offer generic functionalities for deeply integrated and Domino related portlet applications.
**DAO pattern**

The *DAO pattern* is the most important pattern related to the integration tier. It is a special case of the *strategy pattern* (Gamma et al. 1995) and its motivation is to separate resource-tier-related code from the business tier and to improve modularity and reusability. With the help of the DAO pattern, all tiers above the integration tier are made independent from back-end systems because these systems are hidden behind a uniform API.

According to (Alur et al. 2003, p. 463), a DAO is implemented as a *stateless object* and does *not cache* any results from query executions; therefore, threading or concurrency issues are prevented. On the one hand, session state should be held in the web tier since *HttpSession* replication between application servers is efficient and reliable (Johnson 2003, p. 208). On the other, data should be cached as close to the client as possible, especially in distributed applications.

Figure 3.8 presents how DAOs can be applied to decouple business logic from the Domino Java API. An ordinary interface creates an abstraction layer for clients in the business tier. Only the gray marked area is dependent on the Domino Java API and could be replaced by web services or even by JDBC if once a different database system is requested. DAOs that encapsulate the Domino Java API should not expose any DOM objects or exceptions of the *lotus.domino.* package to clients outside the integration tier.
Figure 3.9.: Domino session pooling based on (Alur et al. 2003, fig. 4.24)

**Domino session pool service**

Due to the fact that IIOP sessions to Domino servers require non-trivial amounts of time for creation and destruction (see page 31), a session pool is needed that helps to avoid this bottleneck by sharing resources between portlet applications and maybe also between users. Such a pool can not only improve performance but also scalability because the total number of sessions is limited and they might be exhausted far sooner than desired.

At this point of the architecture design, it has to be decided if and how Domino’s security model can be used by the portlet application. If the security of the Notes application should be used directly, then every user needs personal sessions so that ACL, Reader and Author Fields work properly. In this case every user should have an own session pool so that sessions could be shared at least between the different Domino related portlet applications of an user. This possibility allows secure and quick access to Notes Databases but lavishly uses sessions. Personal sessions to a Domino server are only then returned when a user explicitly logs out of a portal or if the sessions time out. Consequently, a situation might be possible that new IIOP connections are refused while existing connections are barely used.

It can be concluded that in general, personal Notes sessions should be avoided and only used for Domino applications that heavily rely on Reader Fields. If possible, Notes security features should be handled indirectly in the portlet application via a technical user that can connect to and interact with a Domino server so that sessions can be efficiently pooled. Security for the portlet subsystem should be handled in higher tiers and not directly by the Notes application in the resource tier.
Figure 3.9 visualizes the difference between session consumption in a non-pooled and pooled scenario. When a pool is managing sessions, it instantiates them and borrows them to DAOs. Before a DAO returns a Notes session to the pool, the used database object needs to be recycled. This is related to the fact that ‘Java has no knowledge of the heavyweight back-end Domino objects, only the lightweight Java objects representing them. Garbage collection has no effect on Domino objects unless you first explicitly recycle them’ (Tulisalo et al. 2003, p. 257). By recycling database objects the size of sessions is kept low; sessions can be kept alive and then can be borrowed again. Thus, the time consuming creation and destruction of Notes sessions is prevented and they are better utilized.

A Notes session pool should have the ability to grow and shrink automatically related to the amount of user-requested sessions. This feature (and many more) is supported by the `GenericObjectPool` class of the Jakarta Commons object-pooling API. This class can be leveraged if a Domino session pool for a technical user should be developed. In the case that multiple and user-related session pools should be provided, then the `GenericKeyedObjectPool` class of this API is needed (Bergland et al. 2005, p. 407). The Jakarta Commons pool API has been used for the development of a generic service providing Domino sessions for portlet applications and its implementation is explained in the next chapter of this paper. The related interface can be seen in listing 3.1.

Listing 3.1: Interface of session pool service

```java
1 package com.linde.sessionpool.service.domino.core;
2 import lotus.domino.Session;
3 public interface ISessionPoolService {
4     public void destroy();
5     public boolean isServiceOpen();
6     public SessionPoolServiceStatus getServiceStatus();
7     public Session getDominoSession();
8     public void returnDominoSession(Session dominoSession);
9 }
```

**DOMINO NAVIGATION TREE SERVICE**

A client of the Domino session pool service is the Domino navigation tree service which has the task to integrate contextual information of Notes documents into a portal. It addresses back-end Views of a Notes application (see figure 3.7), reads the data they contain and then builds a Java tree out of it. This tree is cached and
therefore does not have to be created for every request. A timer should be considered that frequently updates the forest of navigation trees handled by this service.

From the implementation point of view, the `DefaultMutableTreeNode` class of the `javax.swing.tree` package can be used to build a tree that contains custom user objects. These objects should be simple Java beans that among other things contain the UNIDs of related documents and like this provide a convenient way for the usage of the `getDocumentByUNID()` method. This approach is used for the implementation of the NavTree service introduced in the last chapter of this paper. The interface of this service is displayed in listing 3.2.

```
package com.linde.pike.service.navtree.bus.core;
import javax.swing.tree.DefaultMutableTreeNode;
public interface INavTreeService {
    public void init();
    public boolean isServiceOpen();
    public NavTreeServiceStatus getServiceStatus();
    public DefaultMutableTreeNode getNavTree(String databaseId);
}
```

### 3.2.5. Business tier

While the purpose of the integration tier was to decouple the business tier from details of data access, the purpose of the business tier is to decouple the user interface from details of business logic through an abstraction layer. (Johnson 2003, p. 252) concludes, that it ‘should be possible to change an application’s persistent strategy without rewriting the business logic, in the same way as we should be able to change an application’s user interface without affecting business logic.’

For this reason, business interfaces should be used that define the contract for the middle tier in ordinary Java interfaces. EJBs are just an implementation strategy and if they are not used then the implementation of the business interface will be Plain Old Java Objects (POJOs) running in a Java EE web container.

As already mentioned (see page 43), this thesis cannot offer a generic solution for the business tier because this tier is dependent on concrete application requirements. For its implementation it needs to be decided if a collocated or distributed architecture is chosen and if and how EJBs are used. This all has no influence on the way how a Domino application is integrated into a portal and thus is out of
3.2.6. Presentation tier

The presentation tier encapsulates presentation logic of an application. Its main pattern is the Model-View-Controller (MVC) pattern whose implementation is described in this section. Additionally, it is explained how files and images (e.g. stored in Notes RichText Items) can be integrated into the UI of a portlet.

The two design goals of the presentation tier are that it should be as clean and thin as possible (Johnson 2003, p. 447). A clean presentation layer separates control flow and invocation of business objects from presentation handled by view components (such as JSP pages). This has the benefit that on the one hand the presentation of the application can be changed without interfering with business object handling and on the other that the development process can be split up between Java developers and markup authors.

An attribute of a thin presentation tier is that it uses just as much Java code as needed to invoke the business logic encapsulated in the business tier. This is useful since testing the UI is more difficult then testing ordinary Java objects which do not require a web container. If the presentation tier of an application also would contain business logic, this logic could only be tested through the UI. Additionally, thin presentation layers enhance code reuse and can better support different interfaces, e.g. needed if the application also should be accessed via web services.

While a thin presentation tier comes along with the use of a business tier, a clean presentation tier can be accomplished by using the MVC pattern.

MVC pattern

The origins of the MVC pattern can be found in the Smalltalk programming language where it was used for UI development (Wolff 2006, p. 192). The same pattern also can be applied in the area of portlet development where it is used to handle requests to a portlet application (see figure 3.10). The controller has the task to implement the processing of a special request and does not contain any code related to screen presentation. When the controller has executed a request, it creates a model which is a data structure containing the result. In a next step, the controller turns over control to the view that renders the data of the model in a suitable way.

The benefit of the MVC pattern is that presentation tier logic is unitized. The
controller parses requests and implements related functionalities while rendering is encapsulated within the view. The purpose of the model is to transport the result of the controller to the view. Like this, the tasks of the presentation tier are clearly separated and the tier is kept clean.

The sequence diagram of figure 3.11 shows how the MVC pattern can be implemented in a portlet application. This approach is described by (Alur et al. 2003, p. 276) where it is labeled ‘Service to Worker’. A typical portlet application often contains multiple portlets. Each portlet has its own standard controller which is a controller of controllers. This standard controller offers a single point of entry to a portlet and its task is to select one of multiple custom request controllers, e.g. dependent on portlet mode or request parameter. The chosen request controller then uses the Command pattern to encapsulate an HTTP specific request to a subsystem as ordinary Java objects. The controller interacts with the business interface of the application and returns the result (the model) to the controller portlet. It is the job of the controller portlet to pass the model to the selected View where the markup fragment is generated which is returned to the portal.

There is no need to newly implement a portlet MVC framework since many implementations are available as open source (e.g. the Spring Portlet MVC framework).

**Integration of files and images**

Even though not explicitly stated in the JSR-168 specification (Abdelnur & Hepper 2003, p. 46), portlets are in general only able to handle data of type ‘text’. This is due to the facts that 1) the portlet provides a fragment which has to be rendered inside an overall portal page and that 2) the portlet API does not provide a second
channel through which binary content could be delivered. A workaround for this problem is to include a non-portlet URL in the markup generated by a portlet. As displayed in figure 3.12, this URL can be used to trigger an ordinary servlet included within the portlet application. After the portlet has returned its markup fragment, the servlet is called automatically (for images) or manually (for file downloads) and will return the requested resource.

The limitation of this approach is that the security model of the portal is bypassed; thus, the servlet needs to be manually included into the security concept by sharing data between PortletSession and HttpSession. The JSR-268 expert group, responsible for the Java portlet specification V2.0, addresses this problem by providing a ResourceServingPortlet interface that declares a serveResource() method (Hepper 2006, p. 67). This will support a second channel for delivering binary content to the user and also can be used for the development of Asynchronous Javascript and XML (AJAX) portlets. The final release of the JSR-268 specification is scheduled for May 2007.
Figure 3.12.: Portlet file and image handling
4. Implementation

The purpose of this chapter is to turn parts of the discussed architecture into action and to present a sample application which has been developed in line with this diploma thesis.

4.1. Foundation

The first section lays the foundation for the sample application. Not only its requirements are presented but also the related portal infrastructure and leveraged application framework are briefly introduced.

4.1.1. Requirements

The main result of the defined integration strategy was that structured and function-oriented data stored in Notes should be deeply integrated while unstructured and display-oriented data should be shallowly integrated into a portal (see figure 3.4). Additionally to this, an area in-between was identified, i.e. ‘simple unstructured data’ stored in simple documents. The sample application focuses on this area since its implementation is most challenging and covers the integration of structured data as well.

The requirements are for a basic Content Management System (CMS) that is realistic enough to present the architectural concept of a deep integration of data stored in simple documents. (Lohr & Deppe 2001, p. 4) defines content as a composition of data, structure and layout. A CMS supports decentralized processing of content and underpins the content life-cycle form creation to archiving.

The developed CMS is limited to the functional requirements displayed in figure 4.1. The actual content is created in the Notes subsystem which consists of two databases. The article database is the front-end to content creators since it offers required functionalities to create and handle articles that can contain simple formatted text, images and file attachments. From the technical view, simple formatted
text is transformed to HTML via DXL (see figure 3.6) and attached images and files are automatically outsourced into an attachment database. This separation between lightweight and heavyweight files into one article and one (or more) attachment databases has the benefit that replication procedures can be smoothed and ACLs specifically edited (Nielsen et al. 2000, p. 8).

The second subsystem of the application consists of two services that support the integration of Notes data into a portal. A Notes session pool provides IIOP sessions for the portlet application and grows and shrinks related to the actual load. The session pool takes itself out of service if Domino is unavailable and returns if Domino is up and running again. The navigation tree service is a client of the session pool service and accesses a View of the article database that contains article relationships (refer to figure 3.7). With this information the navigation tree service creates a cached Java tree which is consumed by the portlet application.

The portlet application uses both portlet services and offers three different portlets. The content portlet surfaces the content stored in the Notes application, the navigation portlet presents a subset of the navigation tree and the navigation path portlet displays the full path from the root of the site to the actual article position. The three portlets can communicate with each other so that interactions with one of the two navigation portlets are reflected by the others.
4.1.2. Portal infrastructure

The sample application should be deployed on the infrastructure displayed in figure 4.2. Two logically grouped WebSphere Application Servers (WAS) are installed on one computer (node) and managed by a node agent. The first application server is named ‘server1’ and hosts the WAS admin console, which is a Java EE application for the administration of all application servers in one node. The second application server ‘WebSphere_Portal’ hosts the actual WebSphere Portal Server (WPS). The benefit of these two WAS is that the WAS admin console does not compete with the portal server for resources (like CPU and memory) because ‘server1’ will only be started if the admin console is needed. Additionally, a DB2 database is installed on the portal computer which mainly is used to store user related data, e.g. needed for personalization.

The portal is connected to a Lotus Domino LDAP server so that Domino users are able to login to the portal by using their existing HTTP passwords. A second Domino server is available that hosts the Notes subsystem of the sample application.

The last server related to the portal infrastructure is an HTTP server on which a WAS plugin is installed. This plugin directs incoming port 80 requests via port 9080 to the WAS and like this enables users to access the portal from the Internet in a secure manner.
4.1.3. Spring framework

‘The main benefit of adapting an existing framework is the same as that in adopting Java EE itself: it enables an organization’s development team to focus its effort on developing the required product, rather then concerning itself with the underlying infrastructure’ (Johnson 2003, p. 168).

The Java application framework Spring 2.0 has been chosen for the development of the sample application since it is widely popular in the Java community and offers the following benefits (Wolff 2006, p. 1f):

1. Spring provides a simplified, unified and unitized API-layer that covers many Java SE APIs, Java EE APIs and open source frameworks. These various APIs follow different concepts and are therefore altogether clumsy to handle if implemented directly.

2. The creation of object relationships in OO applications might be challenging, especially if business logic is implemented as a service. Spring addresses this area by supporting Dependency Injection (DI). This makes it possible to inject objects into other objects according to a XML based configuration. Thus, objects are made independent of their environment since they do not directly interact with it. Objects can be used in different environments (e.g. in a Java EE server or in a standard Java SE runtime) so that e.g. unit testing can be smoothly applied. Another benefit is that the overall configuration of an application is eased due to Spring’s XML configuration.

3. Spring supports Aspect Oriented Programming (AOP). In general, cross-cutting issues related to tracing or security are spread throughout the code. AOP helps to centralize these issues and like this keeps the code of an application clean.

4. Various other frameworks are based on Spring, e.g. Spring Rich Client or the Acegi security framework. In the area of JSR-168 portlet development, Spring offers a portlet MVC framework. This framework is based on Spring’s Web MVC framework (used for the development of standard web applications) and can be combined with Spring WebFlow which helps to manage the page flow of web applications.
4.2. Sample application

This section gives an overview of the sample application. The three subsystems are described and illustrated with the help of models and screenshots.

4.2.1. Notes application

With respect to the Notes subsystem, this part focuses on the resource tier and explains how simple formatted text and attachments are handled and how articles can be two-dimensionally ordered in a navigation tree.

Overview

The Notes application consists of an article and an attachment database. The article database is the front-end to the application since it offers an administration and a publishing area while the attachment database is just a passive repository for article-related files or image attachments.

The administration area of the article database provides three different functionalities. At first, a central reference to an appropriate attachment database can be created so that there is no need to select this database manually each time a file is up or downloaded. The second functionality is related to the navigational structure of published articles. It is possible to restrict the depth of the navigation tree so that too deep document levels can be prevented. The last setting is about the definition of templates which control the rendering of articles (see figure 3.7). Template documents can be created that store custom XSLT stylesheets. It is possible to test these stylesheets directly within the Notes application by dynamically applying them to already existing articles.

Next to this administration area, there is the publishing area intended to be used by content creators. Within this part of the application, articles can be created and related files or images can be uploaded into the attachment database.

Simple formatted text

Figure 4.3 displays an Article-Form and its basic configuration settings. The already introduced templates defined in the administration area can be selected and according to them, specified Items will be transformed into a HTML snippet when
the Postsave event of the Form is triggered. The choice between different templates enables content creators to influence the generated HTML.

Articles are simple documents (see page 36) since they need to be deeply integrated into a portal; thus, a structure has to be provided that on the one hand ensures that this kind of integration is possible and that a consistent layout is maintained. On the other hand, this structure also should give authors enough flexibility in the area of content creation. These requirements were realized with the help of paragraph sections. It is possible to choose the number of paragraphs and like this to adjust the structure of the article to its content. Every paragraph after the first one has its own header and body field. In the created HTML markup, the data in these fields is marked with special <span> tags so that its formatting can be influenced by the portal via Cascading Style Sheets (CSS). CSS offer a possibility to describe the presentation of markup documents.
Figure 4.4.: Relationship between article and file attachments

**File attachments**

The size of simple documents should be kept low; thus, files should not be contained but referenced. This request has been realized in the sample application as visualized in figure 4.4. Only text-based information is stored in article documents and heavyweight files are outsourced into the attachment database. Article documents link to so called *file master documents* which in turn link to *file documents* that store the actual files. File masters are needed so that different versions of a document can be managed.

General benefits of this architecture are that it decreases network traffic and database size in comparison to ‘classic’ Notes applications while it speeds up remote and local Document handling. This is due to the fact that in a distributed environment, users have the possibility only to replicate the lightweight article database while the heavyweight attachment database stays on the server and is only accessed if needed. Even if both databases would be locally replicated, the amount of network traffic could be reduced since a change in the navigational structure or in the text-based content of an article does not result in a new replication of a heavyweight file attachment because its wrapper-document is not affected.

Another benefit (though not implemented in the sample application) would be to prevent redundant storage of file attachments as single files can be referenced from various articles at the same time. Also in the area of ACL controlled security it might be beneficial if multiple file databases are used to support multiple user groups.

Figure 4.5 shows how the functionality of up- and downloading attached files is seen from the perspective of a content creator. Four actions are available related
to file handling: A new file or a new version of an existing file can be created, a file can be downloaded and opened directly in its related program or a file can be removed, i.e. archived. While it is possible to download attached files via the portlet application, attached pictures are directly displayed within the portlet application. Therefore, documents that store pictures additionally contain information about the width and height of each picture.

**Contextual information**

The navigation tree View (refer to figure 4.6) contains contextual information about articles. The topmost article in this View is a *Home* Document which is the root of the navigation tree; thus, the position of this special article is fixed. All other articles can be moved (together with their children) to different positions with the help of related View Actions. Similar to the previously introduced features, also this functionality is encapsulated within an own script library that also controls the validity of requested movements. Therefore, it is e.g. not possible to exceed the maximum depth of the navigation tree specified in the administration area.

The limitation of this approach is that the movement of an article influences other articles that are recursively edited in the background. This might increase the amount of replicated (but lightweight) articles and implies also a risk for replication conflicts.
If a new article should be created, a parent article needs to be selected and then the View Action ‘Create Article’ can be triggered. This will create a new article as the last child of its parent. The position of this article can be adjusted with the help of related View Actions.

Additionally to this front-end View, there is also an associated back-end View a described in figure 3.7. This View is flat-structured but contains all needed data so that the navigation tree service of the portal can build a suitable Java navigation tree.

### 4.2.2. Services for integration of Domino

This part of the paper is about the Domino session and the navigation tree service. Both services are used by the portlet subsystem to interact with the Notes application. Since these services should be available for every application deployed in a portal, they could be realized with special WebSphere Portal components called *portlet services* (Dheap & Chen 2005, p. 4). Like session beans, portlet services are thread-save remote components but managed by a portlet container. Portlet services follow a singleton-based programming model (Hepper & Liesche 2005) and can be easily deployed without configuring deployment-descriptors. Portlets can access these services via JNDI.

---

**Figure 4.6.:** Screenshot of the front-end navigation tree View

- Move selected Document and its children to different position in View tree.
Java EE compliant stateless session beans are not recommended since they cannot be used as singletons. This is due to the fact that session beans are managed by the EJB container. Furthermore, the EJB specification prevents the use of read-write static variables and synchronization (Johnson 2003, p. 220). Singletons are required since only one session pool and one navigation tree per View should be provided.

In regard to the sample application, the initial decision was to implement the Notes session pool and navigation tree builder by leveraging WebSphere’s portlet services together with Spring as displayed in figure 4.7. The upper part of the figure presents a Spring managed application while the lower part shows a WebSphere portlet service through which the Spring application can be accessed. Designed like this, the functionality of the service is encapsulated within the Spring application and is not directly bound to a WebSphere Portal specific API. This approach works fine as long as the portlet application is implemented without Spring. If a Spring managed portlet application tries to consume a Spring managed portlet service, a BeanDefinitionStoreException is thrown probably due to the fact that different classloaders are used for portlet application and portlet services (Leau 2006). Thus, it was decided not to offer portal-wide services in the sample application but to develop pure Spring based services that support the integration between portlet and Notes application independently from the runtime environment. This makes it possible to use these services more flexibly (since they are not bound to a portal server) but has the disadvantage that every portlet application will use its own instance of a service; therefore, Notes resources cannot be shared across multiple portlet applications.

Domino session service

In the following the Notes session service is described. At first its classes are introduced (see figure 4.7) and then the related object net is presented which is wired by Spring.

The ISessionPoolService interface (refer to listing 3.1 on page 50) has an important role for the Notes session service since it exposes the API of the service and thus is directly used by its clients. This interface is implemented by the SessionPoolService class which contains the actual session pool. It is the task of the SessionPoolFactory to leverage a given Jakarta Commons GenericObjectPool and to provide functionalities so that this pool can manage Notes sessions. The SessionPoolFactory has access to Spring’s ApplicationContext and therefore can
send an event to the `SessionPoolService` if Domino is unavailable. In this case the `SessionPoolService` takes itself out of service, closes the session pool and uses a controller class to start a timer. This timer periodically calls a `SocketChecker` that tests as long as the Domino server is up and running again. The timer itself is provided by Quartz\(^1\), an open source job scheduling system. Additionally, there are two convenience classes `PoolConfig` and `AccessConfig` into which all configuration information is injected from associated property files.

As a complement to the class diagram, figure 4.8 presents a graph of related Spring beans which all are implemented as singletons. When the application is in-

---
\(^1\)Since Quartz starts unmanaged threads, IBM recommends CommonJ WorkManager for the use in a productive WebSphere environment (Alcott et al. 2006). Spring also supports this scheduling package but this requires more configuration effort.
stintiated, the \texttt{init()} method of the \texttt{sessionPoolServiceDomino} bean is called and like this the session pool is initialized. Spring injects two beans into the \texttt{sessionPoolServiceDomino}. There is on the one hand a bean which triggers Quartz related classes and that checks the availability of Domino in the case that the connection between portal and Domino server is broken. On the other hand a factory is injected that can create session pools. This factory itself uses two beans that contain configuration information about the session pool and access information of the Domino server. The latter bean is also used by the \texttt{socketCheckerDomino} bean whose \texttt{socketCheck()} method is frequently called if the service cannot connect to Domino. Finally there is a \texttt{propertyPlaceholderConfigurer} that reads the three configuration files of this service and injects their properties into appropriate Spring beans.
Navigation tree service

The class diagram of the developed navigation tree service is displayed in figure 4.9. Similar to the session pool service, also this service has an interface that exposes an API which is directly used by its clients. This INavTreeService interface is presented in more detail in listing 3.2 on page 51 and is implemented by the NavTreeService class that holds a Map (or forest) of multiple navigation trees. Every accessed Notes View has its own navigation tree that consists of DefaultMutableTreeNode objects of the javax.swing.tree package. Like this, each published article within the Notes article database is represented by a node in the navigation tree. A node contains a user object with information e.g. about the title of an article or its UNID. It is the NavTreeService class that creates such a navigation forest with the help of the INavTreeFactory interface. The NavTreeFactoryDomino class implements this interface and offers two public methods that can be used to either create a complete navigation forest or a single navigation tree. This class is a client of the previously introduced session pool service and gets its configuration data from a NavTreeFactoryDominoConfig Java bean. Additionally, a Quartz scheduler frequently updates the navigation forest provided by this application.

There are two packages related to this service. The ...
...bus.core package contains the basic business logic of the service while the ...
...db package is used for database related functionalities. Since the NavTreeService class only depends on the INavTreeFactory
interface, it is possible to switch database-related functionalities without impacting the classes of the ....bus.core package (DAO pattern).

Figure 4.10 shows the wiring of service related Spring beans which are all instantiated as singletons. The pikeNavTreeService bean is the heart of the service since it is an instance of the NavTreeService class. Its init() method is called during application instantiation and like this the service is initialized. The pikeNavTreeService gets a pikeNavTreeFactoryDomino bean injected which it uses to create a navigation forest that contains all required navigation trees. Three beans are in turn injected into this factory. One bean contains information about related Notes Views, one bean represents the session pool service and one bean is a helper that supports the handling of navigation trees. The upper part of the figure is related to Quartz which is used to periodically call the updateNavForest() method of the pikeNavTreeService bean.

4.2.3. Portlet application

The last part of this section addresses the portlet application. It starts with an introduction of the structure of this application, moves on to the related Spring configuration and finally explains how inter-portlet communication and user interface have been developed.
Structure

The three tiers of the portlet application are visualized in figure 4.11. At the bottom of this figure there is the integration tier that offers its functionality via two interfaces. The INavTreeDao interface is either used to get a complete navigation tree or to receive a single node of such a tree (by passing order and document level of the node). According to the DAO pattern, a service specific NavTreeDaPikeService class implements this interface and directly interacts with the navigation tree service. In addition to the INavTreeDao interface, the IArticleDao interface provides functionalities to read an article or an article attachment. This interface is implemented by the ArticleDaoDomino class which leverages the Domino session pool and then directly accesses the Notes databases in the resource tier.

On top of the integration tier there is the business tier. It is in this tier where a deep integration of a Notes application into a portal pays off since here it is possible to aggregate data of different systems and to build a process oriented layer (see page 35). However, the sample application has a single Notes application in its back-end and thus only works with data from this source.
Similar to the integration tier also the business tier exposes its functionality through two interfaces. The IReadOnlyManager interface is implemented by the NavigationManager class which uses two factories to build a sub-navigation or a navigation path out of a navigation tree. The other interface is the IReadOnlyManager interface that in this state of expansion directly relates to the IReadOnlyDao interface.

The highest tier of the portlet application is the presentation tier which contains three portlets and one servlet. The different portlets are used to display navigation and articles while the servlet delivers binary content, i.e. attached pictures and files. The sample application uses Spring’s portlet MVC framework. Three instances of Spring’s generic dispatcher portlet are leveraged that offer three points of entry to the application and that cooperate with the three controllers displayed in figure 4.11. These controllers extent Spring’s AbstractController class and overwrite its portal-related life-cycle methods. NavViewController and NavpathViewController rely on the navigation related interface of the business tier while ArticleViewController and the servlet interact with the business tier through the IReadOnlyManager interface.

**Spring wiring**

This part explains how Spring addresses the structure of the portlet application and produces a fully configured system ready for use.

Figure 4.12 shows the four configuration files of the sample application. At first there is the main configuration contained in the applicationContext.xml. Within this file the Spring beans of the integration and business tier are defined and wired together. Additionally, this file contains beans that portlets of the presentation tier have in common. Beans constricted to a single portlet are configured in three separate and portlet-related application contexts. These files are children of the applicationContext.xml; thus, they have access to all beans defined in the main configuration.

Spring reads linked property files and injects their values into the pikeConfigHttp and pikeConfigDomino beans. While the first one contains keys and values of used HTTP parameters and attributes, the latter one holds the names of design elements of the connected Domino application. Spring injects the pikeConfigHttp bean into the two factories responsible for the creation of sub-navigation and navigation path. These factories in turn are injected into the navigationManager bean. This bean furthermore has access to the navigation tree service because the main configuration file imports the pikeServiceNavtree.xml in which the pikeNavTreeService bean is defined. The pikeServiceNavtree.xml file itself imports the sessionpoolServiceDomino.xml; therefore, also the bean that represents the Domino sessionpool service is available in the main configuration of the portlet application.
A special bean is the viewResolver that relies on the InternalResourceViewResolver class of the Spring MVC framework. This bean decouples controllers from views as requested by the MVC pattern. Like this, portlet controllers can select a view by name without any further knowledge of its implementation. Then the viewResolver takes over and resolves this name of the view to find the instance associated with that name at runtime (Wolff 2006, p. 215). In the sample application the view resolver is configured to support cached JSP Standard Template Library (JSTL) views that can be found in the /WEB-INF/jsp/ directory.

While the same view resolver is used in all three portlet contexts, every single portlet leverages an ownportletModeHandlerMapping bean. Thus, there are three of these beans that are all based on Spring's PortletModeHandlerMapping class. A handler mapping is related to each generic dispatcher portlet and configures the mapping of an incoming
portlet request to an appropriate handler, i.e. a request controller (see figure 3.11). In general, a mapping can depend on the portlet mode or on request parameters but it is also possible to write custom handler mappings which can be related to ‘anything available to the portlet request’ (Johnson et al. 2006, p. 323). The three handler mappings of the sample application simply map a view-mode request of the related portlet to a single controller that processes this request and that returns a model and a name of a view that should generate the markup fragment.

**Inter-portlet communication**

Inter-portlet communication describes the possibility of portlets to interact with each other by sharing data. As already mentioned on page 22, the JSR-168 standard does not address this area sufficiently. The only possibility for JSR-168 compliant portlets to share data is to declare application-wide session attributes. This approach is very basic since a portlet can only process a received message in its render phase but not in the recommended action phase. Therefore, the sample application uses a WebSphere Portal specific solution for inter-portlet communication that overcomes these limitations.

According to (Rodriguez et al. 2005, p. 801ff), a WebSphere portal provides a property broker that manages data exchange between portlets and that is implemented as a portlet service. This enables JSR-168 portlets that run in a WebSphere Portal to communicate through wires with each other. A wire defines a communication channel for a message
that contains data. In the case that one portlet should send a message to another portlet, these two portlets have to expose their capabilities to produce and consume messages to the property broker. Portlets provide this information in WSDL defined interfaces that are processed at deployment time. The contained information is persisted in the portal datastore. After portlet deployment, the wiring tool (refer to figure 4.13) can be used to create specific wires between portlets. If now a user triggers a custom action in the source portlet, an action filter determines e.g. according to an in the WSDL file defined request attribute that the `processAction()` method of the target portlet has to be called and that the specified request attribute has to be submitted from source to target portlet.

The three portlets of the sample application should communicate with each other since the content displayed in all portlets needs to be updated if a user navigates to a new article. The data exchanged between portlets should be as little and simple as possible since data has to be serialized and de-serialized (which is time consuming). Thus; the two portlets responsible for the navigation only submit an article ID to the article portlet since this is the required information so that a different article can be displayed (see figure 4.14). Between each other, the navigation portlets transfer Java beans that contain the order and document level of a selected article.

**User interface**

The main layout of the portal is determined by an overall portal *theme* and by various portlet related *skins*. Themes represent the look and feel of a portal and define colors, images and fonts. In addition to this, skins describe ‘the visual appearance of the area surrounding an individual portlet. Each portlet can have its own skin. The skins that are available for use with a portlet are defined by the portal theme that is associated with the page.’ (Rodriguez et al. 2005, p. 20)
Figure 4.15.: Screenshot of portlet application

Next to these pre-defined markup fragments produced by themes and skins, the sample application creates its own markup fragments with the help of three different JSTL views. Each view belongs to a portlet and is selected during request processing by the associated request controller. The views of the two navigation related portlets take the passed model and render it as displayed in figure 4.15. The view related to the article portlet not only presents the HTML snippet provided by the Notes application but also incorporates binary content by using the developed ArticleAttachmentHandlerServlet as already presented in the sequence diagram of figure 3.12.
5. Outlook

This chapter of the diploma thesis presents possibilities how the architecture and the sample application could be enhanced and extended.

5.1. Architecture

The discussed architecture covers both a deep and a shallow portal integration of data stored in Notes databases. The architecture is based on 1) an integration strategy and on 2) proven concepts used to design enterprise systems. It is natural that the chosen integration strategy will change over time due to new technologies and advancements in architectural concepts, patterns and related frameworks.

Resource tier

The resource tier of the architecture could be extended in the area of deep integration of function-oriented data. If such structured data has to be integrated into a portal, a relational database could be placed in-between Lotus Domino and portal server. The task of this additional database is to be a data broker by storing Notes data redundantly in tables and by making this data available for the portlet application. Every row of such a table represents data of one Notes Document; rows and Documents are synchronized with each other. This approach makes it possible to use the sophisticated connectivity support that application servers offer for relational databases. Thus, in high-throughput scenarios data interaction can be accelerated while at the same time existing data is available also via the Notes client. This approach also has an impact for the DAOs of the integration tier since the Domino Java API needs to be replaced by JDBC.

One possibility how this solution could be implemented are DB2 Access Views. According to (Monson et al. 2006, p. 10), these Views are background (i.e. non-UI) design elements that contain data stored in Notes Items for primitive data types. This data is redundantly stored in a table of a relational DB2 database. DB2 Access Views can be queried via SQL and are automatically synchronized with their Notes Document counterparts. Figure 5.1 presents this architecture and extends the architecture previously displayed in figure 3.5.
Presentation tier

Most technology changes are expected in the presentation tier since this tier encapsulates the relatively young portal technology. The next enhancement of this technology is the adaption of the Java Portlet Specification V2.0 which targets inter-portlet communication, AJAX and the delivery of binary content. This specification should be released in May 2007\(^1\) but will not change the overall architecture designed in this diploma thesis since it has been ensured that changes of one tier do not effect other tiers.

\(^1\)It will take further time until this new standard is supported by commonly used portal servers.
5.2. Sample application

The sample application is not aimed to be a full-fledged Content Management System (CMS) but shows an extract of architectural concepts that have been identified in this paper. In the following, possibilities are presented how the different subsystems of the sample application could be enhanced.

Notes application

- The standard XSLT template that transforms DXL into HTML snippets needs to be extended since the version leveraged by the sample application only handles plain or bold text. The Notes application has been designed so that template changes can be easily applied. Custom XSLT stylesheets are stored in ordinary Notes Documents that can be accessed in the administration area of the article database.

- A preview feature is needed so that content authors can check how Documents will display in the portal. For this purpose existing article-related HTML snippets could be used and embedded locally from the Notes client into a static HTML page that leverages the same CSS as the portlet application.

Services

- Session pool and navigation tree could be implemented as portal-wide services like originally planned. Thus, Domino resources could be more efficiently used since they could be shared between portlet applications. Portal-wide services can be implemented with the help of WebSphere portlet services (see page 65). If these services should be implemented with the Spring framework, it might be possible to access relevant bean factories via JNDI and with the help of Spring’s BeanFactoryLocator.

- The developed Notes session pool service could be extended so that each portal user could use a personal session pool. This feature might be needed if the Domino application relies on Reader Fields. It can be implemented with the help of the GenericKeyedObjectPool class of the Jakarta Commons object-pooling API. Anyway, it has been mentioned that in general personal session pools should be avoided since they use Notes resources lavishly (refer to page 49).

Portlet application

- In the sample application caching is only applied for the JSTL Views and based on a standard feature of Spring’s Portlet MVC framework. One possibility how caching could be extended is to access HTML snippets of articles, e.g. via a WebSphere
Application Server (WAS) specific feature called *command cache* (Dheap & Chen 2005, p. 18f). In addition to this, file and picture attachments could be copied to the file system of the portal server to speed up accessibility and to reduce network traffic between Domino and portal server (Bergland et al. 2005, p. 576).

- The sample application uses a navigation and a navigation-path portlet that provide an application related sub-navigation. This sub-navigation is not integrated into the overall portal navigation tree. WebSphere Portal V5.1 and V6.0 offer a special Service Provider Interface (SPI) that allows the integration of an application related sub-navigation into the portal navigation. The relevant classes are included in the package `com.ibm.wps.portlet.menu` but can be applied only if the IBM Portlet API is used. In case that a future version of WebSphere Portal will also support this feature for JSR-168 compliant portlets, the overall portal navigation could be extended with data provided by the developed navigation tree service. This would mean that the two navigation portlets could be removed from the sample application.

- WebSphere Portal V5.1 and V6.0 do not support the creation of user-friendly URLs which could be used to directly point to a specific article from outside of the portal. Even though it is possible to get a handle to URLs with WebSphere’s `PortalStateManagerService`, the portlet application cannot access a custom URL parameter that would contain necessary information to display a selected article. This feature might be supported in a future version of WebSphere Portal and could then be implemented.
6. Summary

The objective of this diploma thesis was to design a generic architecture that integrates document-centric content stored in Notes databases into a Java based and process driven enterprise portal.

The motivation of this paper was related to the fact that data tends to be immobile since it cannot be moved easily from one into another system. Therefore, it is beneficial to keep data in its original place and to access and interact with it remotely. Enterprise portals assist in implementing this approach; they offer a unified knowledge desktop that consolidates the view of various data pools. Unfortunately Java EE application servers offer connectivity support only for highly structured data persisted in relational databases; thus, Notes with its semi-structured compound documents is a bit exotic in this field. The paper targeted this niche and presented an architecture that answered the question ‘What is the best way to integrate my Notes application into a portal?’

The second chapter introduced relevant basics. The first section of this chapter gave an overview of two-tier architectures, the Lotus Notes client and the Domino server. It was explained which technologies can be used to remotely access Notes databases. The second section brought enterprise portals into the picture and started with both a technical and business introduction of portal concepts. From the technical view enterprise portals address aggregation, personalization, presentation and security. The business side emphasized that only if these functionalities are primarily used to improve the core and cross-system business processes of an organization, an investment into a portal infrastructure can be justified. Afterwards the discussion turned to multi-tier architectures and described them in the context of Java EE and application servers. Based on this, the architecture of a portal was introduced with a focus on portal components and the portlet API. The chapter closed with an overview of JSR-168 and WSRP development standards.

The third chapter was the main chapter of this paper. In the beginning a strategy was identified that described how Lotus Notes databases could be integrated into a portal. This strategy was build upon a technical evaluation of different integration technologies and a business evaluation of different integration levels, i.e. portlet patterns. The result of this evaluation was that two different integration possibilities were identified. Structured data, stored in Notes databases and related to the function-oriented part of business processes,
should be deeply integrated into a portal so that data of different back-end systems can be handled together in the business tier. In this scenario an enterprise portal can provide a process-oriented layer that consolidates access to back-end systems; therefore, productivity and quality of labor can be improved. If read access is sufficient, then Notes XML Views and Forms should be used for such a deep portal integration of data. For write access the Domino Java API would be the first choice. In comparison to this, unstructured data stored in Notes’ RichText Items is primarily intended for publishing and therefore can be shallowly (and preferably process related) integrated into a portal by using IFrames. It was mentioned that this unstructured data can only be displayed (and not edited) in a portal interface.

The second part of the main chapter was based on the identified integration strategy and discussed a related architecture. After functional and non-functional requirements were presented, an overview of this architecture and its tiers was given. The main idea was to merge two-tier architectures and multi-tier architectures into a new, mixed architecture. This was necessary since both a deep and a shallow oriented integration had to be supported. Structured or ‘simple unstructured’ information stored in Notes databases could be deeply integrated into a portal and therefore a multi-tier architecture was chosen. On the other hand, a two-tier architecture was used since there might be the need to display unstructured information via IFrames. The rest of the main chapter analyzed the different tiers of the architecture in more detail. According to the resource tier, it was described how data (to be integrated) should be prepared by the Notes application. For the integration tier, the DAO pattern was used to encapsulate resource tier dependent APIs from higher tiers; a Notes session pool and a navigation tree service were designed. The section of the presentation tier addressed an implementation of the MVC pattern and showed how binary content could be integrated into the user interface of a portal.

The purpose of the fourth chapter was to develop a sample application that turned parts of the discussed architecture into action. In the beginning, functional requirements of a basic Content Management System (CMS) were introduced. This CMS was then composed of three different subsystems: A Notes subsystem with two databases, a service subsystem that provided a Notes session pool and a navigation tree builder, and a portlet subsystem with three portlets. The rest of the chapter described the implementation of this sample application. Services and portlet application were developed with the help of the Spring application framework, while inter-portlet communication was implemented with a proprietary solution offered by the leveraged WebSphere Portal. Additionally the portal infrastructure was presented in which the sample application was deployed.

Finally an outlook was given how the developed architecture and the sample application could be extended in the future.
A. Artifacts of Sample Application

A.1. XSLT Stylesheet used by Notes subsystem

An XSL Transformer uses this stylesheet to create a HTML representation of data stored in multiple Notes Items. The paper describes this process and visualizes it in the left part of figure 3.6 on page 46.

Listing A.1: Standard XSLT of article database

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
    xmlns:dxl='http://www.lotus.com/dxl'>
    <xsl:output method="html" indent="yes"/>

    <!-- variable PRODUCTIVE [used for hidewhen of empty paragraphs] -->
    <xsl:variable name="numParagraphs" select="number(dxl:document/dxl:item[@name='txt_NumParagraphs ']/dxl:textlist/dxl:text)" />

    <!-- variable TESTING [used for hidewhen of empty paragraphs] -->
    <xsl:variable name="numParagraphs" select="number(dxl:document/dxl:item[@name='txt_NumParagraphs ']/dxl:text)" />  -->

    <!-- key for every par-element whose value is the id of the related pardef-element [used for lists] -->
    <xsl:key name="par" match="dxl:par" use="generate-id(preceding::dxl:pardef[1])" />

    <!-- root element -->
    <xsl:template match="/">
        <HTML><BODY>
            <xsl:comment>START notes content</xsl:comment>

            <!-- paragraph 0 -->
            <SPAN class="article_paragraph_text">
                <xsl:apply-templates select="dxl:document/dxl:item[@name='rtf_Body_0 ']/dxl:textlist/dxl:text" />
            </SPAN>

            <!-- paragraph 1 -->
            <xsl:if test="$numParagraphs &gt; 1">
                <SPAN class="article_paragraph_header">
                    <xsl:apply-templates select="dxl:document/dxl:item[@name='rtf_Body_1 ']/dxl:textlist/dxl:text" />
                </SPAN>
            </xsl:if>

        </BODY></HTML>
    </xsl:template>
</xsl:stylesheet>
```
A.2. Spring configuration of services

A.2.1. Properties of session pool

This properties file configures the developed Notes session pool. The pool is based on the `GenericObjectPool` class of the Jakarta Commons object-pooling API.

Listing A.2: `sessionpoolservicedomino-pool.properties`

```properties
sessionpoolservicedomino.pool.maxActive=200
#
# cap on the total number of active instances from the pool. When non-positive,
# there is no limit to the number of objects that may be active at one time.
# default: 8
# recommended: 200

sessionpoolservicedomino.pool.maxIdle=50
#
# cap on the number of "sleeping" instances in the pool. When negative, there is no
# limit to the number of objects that may be idle at one time.
# default: 0
# recommended: 50

sessionpoolservicedomino.pool.maxWait=2000
#
# maximum amount of time (in millis) the borrowObject() method should block before
# throwing an exception when the pool is exhausted and the "when exhausted"
# action is WHEN_EXHAUSTED_BLOCK.
# default: -1L
# recommended: 2000

sessionpoolservicedomino.pool.minEvictableIdleTimeMillis=10000
#
# recycle sessions if > X millis idle
# default: 1800000 (30 min)
# recommended: 240000 (4 * 60 * 1000 millis = 4 min)

sessionpoolservicedomino.pool.minIdle=5
#
# minimum number of "sleeping" instances in the pool before before the evictor
# thread (if active) spawns new objects.
# default: 0
# recommended: 5

sessionpoolservicedomino.pool.numTestsPerEvictionRun=100
#
# number of objects to examine per run in the idle object evictor
# default: 3
# recommended: 100

sessionpoolservicedomino.pool.softMinEvictableIdleTimeMillis=-1
#
# default: -1L

sessionpoolservicedomino.pool.testOnBorrow=true
sessionpoolservicedomino.pool.testOnReturn=true
sessionpoolservicedomino.pool.testWhileIdle=true
#
# test object when borrowed, returned or idle
# default: false
```

86
A.2.2. Domino session pool service

Figure 4.8 on page 67 presents the Spring wiring of the session pool service. This is the related XML configuration.

Listing A.3: sessionpoolServiceDomino.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
                         http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="sessionPoolServiceDomino" class="com.linde.sessionpool.service.domino.core.SessionPoolService"
       init-method="init" destroy-method="destroy">
    <constructor-arg ref="sessionPoolFactoryDomino"/>
    <constructor-arg ref="socketCheckerControllerDomino"/>
  </bean>

  <bean id="sessionPoolFactoryDomino" class="com.linde.sessionpool.service.domino.core.SessionPoolFactory">
    <constructor-arg ref="accessConfigDomino"/>
    <constructor-arg type="int" value="${sessionpoolservicedomino.checker.interval}"/>
  </bean>

  <bean id="socketCheckerDomino" class="com.linde.sessionpool.service.domino.core.SocketChecker">
    <constructor-arg ref="accessConfigDomino"/>
  </bean>

  <bean id="socketCheckerControllerDomino" class="com.linde.sessionpool.service.domino.core.SocketChecker">
    <constructor-arg ref="accessConfigDomino"/>
  </bean>
</beans>
```
<bean id="socketCheckerSchedulerDomino"
    class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers">
        <list>
            <ref local="socketCheckerJobTriggerDomino"/>
        </list>
    </property>
</bean>

<bean id="socketCheckerJobTriggerDomino"
    class="org.springframework.scheduling.quartz.SimpleTriggerBean">
    <property name="jobDetail" ref="socketCheckerJobDomino"/>
    <property name="startDelay" value="${sessionpoolservicedomino.checker.interval}"/>
    <property name="repeatInterval" value="${sessionpoolservicedomino.checker.interval}"/>
</bean>

<bean id="socketCheckerJobDomino"
    class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
    <property name="targetObject" ref="socketCheckerDomino"/>
    <property name="targetMethod" value="socketCheck"/>
</bean>

<!-- ~~~~~~~~~~~~~~~~~~~~ Properties ~~~~~~~~~~~~~~~~~~~~ -->

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="ignoreUnresolvablePlaceholders" value="true"/>
    <property name="locations">
        <list>
            <value>classpath:sessionpoolservicedomino-pool.properties</value>
            <value>classpath:sessionpoolservicedomino-access.properties</value>
            <value>classpath:sessionpoolservicedomino-checker.properties</value>
        </list>
    </property>
</bean>

<bean id="accessConfigDomino"
    class="com.linde.sessionpool.service.domino.core.AccessConfig">
    <property name="ipAdress" value="${sessionpoolservicedomino.access.ipAdress}"/>
</bean>
A.2.3. Domino navigation tree service

Figure 4.10 on page 69 presents the Spring wiring of the navigation tree service. This is the related XML configuration.

Listing A.4: pikeServiceNavtree.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd"/>
```
<bean id="pikeNavTreeService"
   class="com.lin.de.pike.service.navtree.bus.core.NavTreeService"
   init-method="init">
   <constructor-arg ref="pikeNavTreeFactoryDomino"/>
</bean>

<bean id="pikeNavTreeFactoryDomino"
   class="com.lin.de.pike.service.navtree.db.NavTreeFactoryDomino">
   <constructor-arg ref="pikeNavTreeFactoryDominoConfig"/>
   <constructor-arg ref="pikeNavTreeHelper"/>
   <constructor-arg ref="sessionPoolServiceDomino"/>
</bean>

<bean id="pikeNavTreeFactoryDominoConfig"
   class="com.lin.de.pike.service.navtree.db.NavTreeFactoryDominoConfig"
   init-method="init"/>

<bean id="pikeNavTreeHelper" class="com.lin.de.pike.service.navtree.bus.core.NavTreeHelper"/>

<!-- ~~~~~~~~~~~ SessionPoolService via Spring ~~~~~~~~~~~ -->
<import resource="classpath:sessionpoolServiceDomino.xml"/>

<!-- ~~~~ SessionPoolService via JNDI (WebSphere portletservice) ~~~~
<jndiObjectFactoryBean>
   <property name="jndiName" value="portletservice/com.lin.de.
   sessionpool.service.domino.portlet.ISessionPoolPortletService"/>
</property>
</jndiObjectFactoryBean>
</bean>

<!-- ~~~~~~~~~~~~~~~~~~~~ Quartz ~~~~~~~~~~~~~~~~~~~~ -->
<!-- Job starts automatically with periodical updates of the navigation
   trees when application is started -->
<bean id="pikeNavForestUpdaterJob" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
   <property name="targetObject" ref="pikeNavTreeService"/>
   <property name="targetMethod" value="updateNavForest"/>
</bean>

<bean id="pikeNavForestUpdaterJobTrigger" class="org.springframework.scheduling.quartz.SimpleTriggerBean">
   <property name="jobDetail" ref="pikeNavForestUpdaterJob"/>
   <property name="jobDetail" ref="pikeNavForestUpdaterJob"/>
</bean>
<property name="repeatInterval" value="${pikeservicenavtree.updater.interval}"/>

</bean>

<bean id="pikeNavForestUpdaterSchedulerFactoryBean" class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
  <property name="autoStartup" value="true"/>
  <property name="triggers">
    <list>
      <ref local="pikeNavForestUpdaterJobTrigger"/>
    </list>
  </property>
</bean>

<beans>

<!-- ~~~~~~~~~~~~~~~~~~~~ Properties ~~~~~~~~~~~~~~~~~~~~ -->

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <!-- allows multiple PropertyPlaceholderConfigurers -->
  <property name="ignoreUnresolvablePlaceholders" value="true"/>
  <property name="locations">
    <list>
      <value>classpath:pikeservicenavtree-updater.properties</value>
    </list>
  </property>
</bean>

</beans>
A.3. Spring configuration of portlet application

Figure 4.12 on page 72 presents the Spring wiring of the portlet application. These are the related XML configuration files.

A.3.1. Main configuration

Listing A.5: applicationContext.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee
        spring-jee-2.0.xsd">

    <!-- ############ Default Presentation Layer ############ -->
    <!-- ~~~~~~~~~~~~ Default View Resolver ~~~~~~~~~~~~ -->
    <bean id="viewResolver" class="org.springframework.web.servlet.view.
        InternalResourceViewResolver">
        <property name="cache" value="false" />
        <property name="viewClass" value="org.springframework.web.servlet.view.
            JstlView" />
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>

    <!-- ############ Business Layer ############ -->
    <!-- ~~~~~~~~~~~~ NavigationManager ~~~~~~~~~~~~ -->
    <bean id="navigationManager" class="com.linde.pike.bus.NavigationManager">
        <constructor-arg ref="navigationFactory"/>
        <constructor-arg ref="navigationPathFactory"/>
        <constructor-arg ref="navTreeDaoPikeService"/>
    </bean>

    <!-- ~~~~~~~~~~~~ Navigation Factory ~~~~~~~~~~~~ -->
        NavigationFactory">
        <constructor-arg ref="pikeConfigHttp"/>
    </bean>

    <!-- ~~~~~~~~~~~~ Navigation Path Factory ~~~~~~~~~~~~ -->
    <bean id="navigationPathFactory" class="com.linde.pike.bus.
        NavigationPathFactory">
        <constructor-arg ref="pikeConfigHttp"/>
    </bean>

    <!-- ~~~~~~~~~~~~ ArticleManager ~~~~~~~~~~~~ -->
```

92
<bean id="articleManager" class="com.linde.pike.bus.ArticleManager">
    <constructor-arg ref="articleDaoDomino"/>
</bean>

<!-- Database Layer -->
<!-- Nav Tree DAO (Domino) -->
- pikeNavTreeHelper via pikeServiceNavtree.xml
- pikeNavTreeService via pikeServiceNavtree.xml

<bean id="navTreeDaoPikeService" class="com.linde.pike.db.NavTreeDaoPikeService">
    <constructor-arg ref="pikeNavTreeHelper"/>
    <constructor-arg ref="pikeNavTreeService"/>
</bean>

<!-- Article DAO (Domino) -->
- sessionPoolServiceDomino via pikeServiceNavtree.xml

<bean id="articleDaoDomino" class="com.linde.pike.db.ArticleDaoDomino">
    <constructor-arg ref="pikeConfigDomino"/>
    <constructor-arg ref="sessionPoolServiceDomino"/>
</bean>

<!-- PikeNavTreeService + SessionPoolServiceDomino via Spring -->
<import resource="classpath:pikeServiceNavtree.xml"/>

<!-- PikeNavTreeService via JNDI (WebSphere Portlet Service) -->
<bean id="pikeNavTreeServiceHome" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="portletservice/com.linde.pike.service.navtree.bus.portlet.INavTreePortletService"/>
    <property name="resourceRef" value="false"/>
</bean>

<bean id="pikeNavTreeService" class="com.linde.pike.service.navtree.bus.portlet.NavTreePortletServiceFactory">
    <constructor-arg index="0" ref="pikeNavTreeServiceHome"/>
</bean>

<bean id="pikeNavTreeHelper" class="com.linde.pike.service.navtree.bus.core.NavTreeHelper"/>

<!-- Utilities -->
<!-- Properties -->
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <list>
            <value>classpath:pike-http.properties</value>
            <value>classpath:pike-domino.properties</value>
        </list>
    </property>
</bean>
<bean id="pikeConfigHttp" class="com.linde.pike.config.ConfigHttp">
    <property name="reqParamActionnameKey" value="${pike.http.request.param.actionname.key}/">
    <property name="reqParamActionnameValueUpdateNavi" value="${pike.http.request.param.actionname.value.updateNavi}"/>
    <property name="reqParamOrderlevelKey" value="${pike.http.request.param.orderlevel.key}"/>
    <property name="reqParamDoclevelKey" value="${pike.http.request.param.doclevel.key}"/>
    <property name="reqParamUnidKey" value="${pike.http.request.param.unid.key}"/>
    <property name="reqAttrArticlePosFromNavKey" value="${pike.http.request.attrib.articleposition.from.navigation.key}"/>
    <property name="reqAttrArticlePosFromNavpathKey" value="${pike.http.request.attrib.articleposition.from.navigationpath.key}"/>
    <property name="reqAttrArticleIdKey" value="${pike.http.request.attrib.articleid.key}"/>
    <property name="sesAttrArticlePosKey" value="${pike.http.session.attrib.articleposition.key}"/>
    <property name="sesAttrArticleIdKey" value="${pike.http.session.attrib.articleid.key}"/>
</bean>

<bean id="pikeConfigDomino" class="com.linde.pike.config.ConfigDomino">
    <property name="dbArticle" value="${pike.domino.db.article}"/>
    <property name="dbAttachment" value="${pike.domino.db.attachment}"/>
    <property name="viewArticle" value="${pike.domino.view.article}"/>
    <property name="formFile" value="${pike.domino.form.file}"/>
    <property name="formPic" value="${pike.domino.form.pic}"/>
    <property name="itemForm" value="${pike.domino.item.form}"/>
    <property name="itemOrderLevel" value="${pike.domino.item.orderlevel}"/>
    <property name="itemDocLevel" value="${pike.domino.item.doclevel}"/>
    <property name="itemTitle" value="${pike.domino.item.title}"/>
    <property name="itemHomeHeader" value="${pike.domino.item.homeheader}"/>
    <property name="itemBody" value="${pike.domino.item.body}"/>
    <property name="itemFilesNumber" value="${pike.domino.item.filesnumber}"/>
    <property name="itemFilesUnid" value="${pike.domino.item.filesunid}"/>
    <property name="itemFilesLabel" value="${pike.domino.item.fileslabel}"/>
    <property name="itemFilesSize" value="${pike.domino.item.filessize}"/>
    <property name="itemPicsNumber" value="${pike.domino.item.picsnumber}"/>
    <property name="itemPicsUnid" value="${pike.domino.item.picsunid}"/>
    <property name="itemPicsLabel" value="${pike.domino.item.picslabel}"/>
</bean>
A.3.2. Navigation portlet

Listing A.6: navigation.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>

<!-- ~~~~~~~~~~~~~~~~~~~~ View Mode ~~~~~~~~~~~~~~~~~~~~ -->

<bean id="navViewController" class="com.linde.pike.portlet.nav.NavViewController">
  <property name="navigationManager" ref="navigationManager"/>
  <property name="pikeConfigHttp" ref="pikeConfigHttp"/>
</bean>

<!-- ~~~~~~~~~~~~~~~~~~~~ Handler Mappings ~~~~~~~~~~~~~~~~~~~~ -->

<bean class="org.springframework.web.portlet.handler.PortletModeHandlerMapping">
  <property name="portletModeMap">
    <map>
    <entry key="view">
      <ref bean="navViewController" />
    </entry>
    </map>
  </property>
</bean>

</beans>
```

A.3.3. Navigationpath portlet

Listing A.7: navigationPath.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>

<!-- ~~~~~~~~~~~~~~~~~~~~ View Mode ~~~~~~~~~~~~~~~~~~~~ -->

<bean id="navpathViewController" class="com.linde.pike.portlet.navpath.NavpathViewController">
  <property name="navigationManager" ref="navigationManager"/>
  <property name="pikeConfigHttp" ref="pikeConfigHttp"/>
</bean>

</beans>
```
<bean class="org.springframework.web.portlet.handler.PortletModeHandlerMapping">
  <property name="portletModeMap">
    <map>
      <entry key="view">
        <ref bean="articleViewController" />
      </entry>
    </map>
  </property>
</bean>

A.3.4. Article portlet

Listing A.8: article.xml

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <!-- ---------------- View Mode ---------------- -->
  <bean id="articleViewController" class="com.linde.pike.portlet.article.ArticleViewController">
    <property name="articleManager" ref="articleManager" />
    <property name="pikeConfigHttp" ref="pikeConfigHttp" />
  </bean>
  <!-- ---------------- Handler Mappings ---------------- -->
  <bean class="org.springframework.web.portlet.handler.PortletModeHandlerMapping">
    <property name="portletModeMap">
      <map>
        <entry key="view">
          <ref bean="articleViewController" />
        </entry>
      </map>
    </property>
  </bean>
</beans>
A.4. Configuration of inter-portlet communication

Page 73f describes WebSphere Portal's support for inter-portlet communication. These are related WSDL files that configure data exchange between portlets of the sample application as presented in figure 4.14.

A.4.1. Navigation portlet

Listing A.9: navigation.wsdl

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="CooperativeSender_Service"
  targetNamespace="http://cooperativetest"
  xmlns="http://schemas.xmlsoap.org/wsdl/
  xmlns:portlet="http://www.ibm.com/wps/c2a"
  xmlns:tns="http://cooperativetest"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <types>
    <xsd:schema targetNamespace="http://cooperativetest">
      <xsd:complexType name="Navigation_ArticlePosition">
        <xsd:sequence>
          <xsd:element name="orderLevel" type="xsd:string" minOccurs="0" maxOccurs="1" />
          <xsd:element name="docLevel" type="xsd:int" minOccurs="0" maxOccurs="1" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:schema>
  </types>
  <message name="Navigation_Receive">
    <part name="ArticlePosition_Input" type="tns:NavigationPath_ArticlePosition" />
  </message>
</definitions>
```
A.4.2. Navigation path portlet

Listing A.10: navigationPath.wsdl

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="CooperativeSender_Service"
    targetNamespace="http://cooperativetest"
    xmlns="http://schemas.xmlsoap.org/wsdl/
    xmlns:portlet="http://www.ibm.com/wps/c2a"
    xmlns:tns="http://cooperativetest"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```xml
```
<types>
  <xsd:schema targetNamespace="http://cooperativetest">
    <xsd:complexType name="Navigation_ArticlePosition">
      <xsd:sequence>
        <xsd:element name="orderLevel" type="xsd:string" minOccurs="0" maxOccurs="1" />
        <xsd:element name="docLevel" type="xsd:int" minOccurs="0" maxOccurs="1"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="NavigationPath_ArticlePosition">
      <xsd:sequence>
        <xsd:element name="orderLevel" type="xsd:string" minOccurs="0" maxOccurs="1" />
        <xsd:element name="docLevel" type="xsd:int" minOccurs="0" maxOccurs="1"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:simpleType name="ArticleId">
      <xsd:restriction base="xsd:string"></xsd:restriction>
    </xsd:simpleType>
  </xsd:schema>
</types>

<message name="NavigationPath_Receive">
  <part name="ArticlePosition_Input" type="tns:Navigation_ArticlePosition" />
</message>

<message name="NavigationPath_Send">
  <part name="ArticlePosition_Output" type="tns:NavigationPath_ArticlePosition" />
  <part name="ArticleId_Output" type="tns:ArticleId" />
</message>

<portType name="NavigationPath_Service">
  <operation name="NavigationPath_Operation">
    <input message="tns:NavigationPath_Receive" />
    <output message="tns:NavigationPath_Send" />
  </operation>
</portType>

<binding name="NavigationPath_Binding" type="tns:NavigationPath_Service">
  <portlet:binding />
  <operation name="NavigationPath_Operation">
    <portlet:action name="UpdateNavigation"
actionNameParameter="ACTION_NAME" type="standard"
caption="UpdateNavigation" description="UpdateNavigation"/>
</input>

</input>

<portlet:param name="ARTICLE_ID" partname="ArticleId_Output" boundTo="request-attribute" caption="output.ArticleId" />
</output>
</operation>
</binding>
</definitions>

A.4.3. Article portlet

Listing A.11: article.wsdl

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="CooperativeSender_Service"
targetNamespace="http://cooperativetest"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:portlet="http://www.ibm.com/wps/c2a"
xmlns:tns="http://cooperativetest"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <types>
    <xsd:schema targetNamespace="http://cooperativetest">
      <xsd:simpleType name="ArticleId">
        <xsd:restriction base="xsd:string"></xsd:restriction>
      </xsd:simpleType>
    </xsd:schema>
  </types>
  <message name="Article_Receive">
    <part name="ArticleId_Input" type="tns:ArticleId" />
  </message>
  <portType name="Article_Service">
    <operation name="Article_Operation">
      <input message="tns:Article_Receive" />
    </operation>
  </portType>
</definitions>
```
<operation>
</operation>
</portType>

<binding name="Article_Binding"
type="tns:Article_Service">
<portlet:binding />
<operation name="Article_Operation">
  <portlet:action name="UpdateNavigation"
    actionNameParameter="ACTION_NAME" type="standard"
    caption="UpdateNavigation" description=""/>
  <input>
    <portlet:param name="ARTICLE_ID" partname="ArticleId_Input"
      boundTo="request-attribute"
      caption="input.ArticleId" />
  </input>
</operation>
</binding>
</definitions>
Bibliography


IBM (2002), ‘Lotus Notes 6.5.1 Help’.

IBM (2004), ‘Servlet and JSP Development with IBM WebSphere Studio v5.1.1’. IBM WebSphere Training and Technical Enablement.


Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Lappeenranta, den 14.01.2007

(Jens Heipmann)