Enterprise Business Objects: Design and Implementation of a Business Object Framework

Kai-Uwe Schafer

Department of Mathematics and Computing, Cork Institute of Technology, Cork, Ireland.

Follow this and additional works at: https://sword.cit.ie/allthe

Part of the Computer and Systems Architecture Commons, and the Databases and Information Systems Commons

Recommended Citation
Enterprise Business Objects

Design and Implementation of a Business Object Framework

Kai-Uwe Schafer

M.Sc. 1999
Enterprise Business Objects
Design and Implementation of a Business Object Framework

Kai-Uwe Schäfer, Dipl.-Inform. (FH)

Department of Mathematics and Computing
Cork Institute of Technology

Supervisors:
Prof. Dr. Dieter Dippel, Darmstadt
Mr. Jim O'Dwyer, Cork

Sponsor: S.E.S.A. GmbH, D-65760 Eschborn/Ts.

Submitted to the National Council for Educational Awards
for the Degree of
Master of Science

July 1999
For my parents.
Statement

I, Kai-Uwe Schäfer, a registered student of Cork Institute of Technology, do hereby certify that the material, which I now submit as my thesis is entirely my own work and has not been taken from the work of others, save and to the extend that such work has been cited and acknowledged within the text of my work.

Signed: [Redacted]

Date: 26.07.93
Acknowledgements

In the course of this project I have had the opportunity to spend half the time in Cork, in an environment and culture that is quite different to what I was used to. I have grown fond of Ireland and its people, and it is perfectly honest to say that my experiences here have taught me a lot and have also changed me a bit. Therefore, I first of all want to say thank you to my friends in Cork, especially Mick and Wesley, and all the guys at St. Fin Barre’s.

I sincerely thank Dr. Dieter Dippel and Mr. Jim O’Dwyer for all their guidance and encouragement throughout the course of this project; it was very much appreciated.

Many thanks also to the technical staff of the Department of Mathematics and Computing, Cork Institute of Technology, for their help and co-operation, and for the laughs we shared. Míle Buíochas daoibh!

To my esteemed colleagues at the Centre for Distributed Applications, University of Applied Sciences in Darmstadt, for the memorable hours of waffle, wit, and wisdom. I have enjoyed working with you and hope we will stay in contact!

To my sponsor, the S.E.S.A. GmbH, thank you very much for financing my living expenses during the last year. Your help makes research projects possible and enables students to fully concentrate on their subject.

Finally, I wish to thank my parents for their support and concern throughout the years.
Abstract

Software components representing business entities like customer or purchase order introduce a new way of Online Transaction Processing to business applications. Collaborating business objects allow to complete whole business processes as a single distributed transaction, instead of dividing it into queued steps, which sometimes even require user intervention. This is due to the fact that business objects contain both business data and logic and that they incorporate multiple databases from different vendors and different geographic locations in a single transaction.

Business objects cannot be used as stand-alone components, but require a framework of services that manage persistence, concurrent transactions, and state management. Business objects are placed in a component container that implements such a framework. The container transparently activates a business object when requested and deactivates it according to any garbage collection mechanism. It keeps transactional caches and synchronises concurrent access. It manages distributed transactions, which span several business objects, containers, and data sources.

Throughout this thesis, I introduce the concept of business objects and their containers, and discuss the technical issues of persistence, transactions, and concurrency. I compare different types of application servers and discuss their ability to host business objects. In the prototypical implementation of a business object framework I propose a design for a component container and present solutions for the state management and concurrent access to fine-grained business objects by multiple transactions.
# Table of Contents

1 **Introduction** .................................................................................................................. 1-1  
1.1 Enterprise Application Software Development ............................................................ 1-1  
1.1.1 Multi-Tier Distributed Applications ............................................................................. 1-2  
1.1.2 The Role of Java ........................................................................................................... 1-6  
1.1.3 The Role of CORBA .................................................................................................... 1-8  
1.1.4 Containers for Business Logic and Data ....................................................................... 1-9  
1.2 Structure of the Document ............................................................................................... 1-10  

2 **Business Object Components** ....................................................................................... 2-1  
2.1 Introduction ....................................................................................................................... 2-1  
2.2 OMG Standards ................................................................................................................ 2-4  
2.2.1 The Business Object Component Architecture (BOCA) ........................................... 2-5  
2.2.2 Interoperability Specification ..................................................................................... 2-6  
2.2.3 Definition of Business Objects .................................................................................. 2-8  
2.2.4 Object Relationships in the BOCA Environment ......................................................... 2-11  
2.3 Enterprise JavaBeans (EJB) ............................................................................................. 2-12  
2.3.1 EJB Design .................................................................................................................. 2-13  
2.3.2 Session Beans ............................................................................................................. 2-14  
2.3.3 Entity Beans ............................................................................................................... 2-14  
2.3.4 EJB Transactional Characteristics ............................................................................ 2-15  

3 **Persistence of Business Objects** ................................................................................... 3-1  
3.1 Comparison of Relational and Object DBMS .................................................................. 3-1  
3.1.1 Relational Database Management Systems (RDBMS) .............................................. 3-1  
3.1.2 Object Database Management Systems (ODBMS) ................................................... 3-2  
3.1.3 Object Relational Mapping ....................................................................................... 3-3  
3.2 Persistence of Java Objects ............................................................................................. 3-5  
3.2.1 Java Serialisation ......................................................................................................... 3-5  
3.2.2 Java Database Connectivity (JDBC) ......................................................................... 3-7  
3.2.3 Database Middleware Tools ...................................................................................... 3-9  
3.2.4 Persistence-capable Java Virtual Machine ................................................................. 3-10  
3.3 Persistence of CORBA Objects ....................................................................................... 3-11  
3.3.1 Non-Integrated vs. Integrated Solutions ................................................................... 3-12  
3.3.2 Object Database Adapter (ODA) ................................................................................. 3-13  
3.3.2.1 Skeleton Inheritance vs. Delegation (Tie) Approach ............................................... 3-13  
3.3.2.2 Architecture ......................................................................................................... 3-17
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.2.3 Object Reference Representation</td>
<td>3-18</td>
</tr>
<tr>
<td>3.3.2.4 Object Activation</td>
<td>3-20</td>
</tr>
<tr>
<td>3.3.2.5 Object Deactivation</td>
<td>3-22</td>
</tr>
<tr>
<td>3.3.2.6 Transaction Management</td>
<td>3-23</td>
</tr>
<tr>
<td>3.3.3 OMG Persistent State Service (PSS)</td>
<td>3-25</td>
</tr>
<tr>
<td>3.4 Design of a Persistency Layer</td>
<td>3-27</td>
</tr>
<tr>
<td>3.4.1 Architecture</td>
<td>3-28</td>
</tr>
<tr>
<td>3.4.2 Persistent Objects and Locking</td>
<td>3-30</td>
</tr>
<tr>
<td>3.4.3 Object Manager and Cache</td>
<td>3-32</td>
</tr>
<tr>
<td>3.4.4 Schema Mapping</td>
<td>3-34</td>
</tr>
<tr>
<td>3.4.5 Dealing with Relationships</td>
<td>3-35</td>
</tr>
<tr>
<td>4 Distributed Transactions</td>
<td>4-1</td>
</tr>
<tr>
<td>4.1 Introduction</td>
<td>4-2</td>
</tr>
<tr>
<td>4.1.1 ACID Properties</td>
<td>4-2</td>
</tr>
<tr>
<td>4.1.2 Standards</td>
<td>4-2</td>
</tr>
<tr>
<td>4.1.2.1 IBM LU6.2</td>
<td>4-2</td>
</tr>
<tr>
<td>4.1.2.2 ISO OSI-TP</td>
<td>4-3</td>
</tr>
<tr>
<td>4.1.2.3 X/Open DTP</td>
<td>4-3</td>
</tr>
<tr>
<td>4.1.3 Transaction Models</td>
<td>4-5</td>
</tr>
<tr>
<td>4.2 Transactions in Multi-Tier Environments</td>
<td>4-9</td>
</tr>
<tr>
<td>4.2.1 Simple vs. Distributed Transactions</td>
<td>4-9</td>
</tr>
<tr>
<td>4.2.2 TP Monitors</td>
<td>4-11</td>
</tr>
<tr>
<td>4.2.3 ORB Interfacing to a TP Monitor</td>
<td>4-13</td>
</tr>
<tr>
<td>4.2.4 Integrated CORBA Solution</td>
<td>4-14</td>
</tr>
<tr>
<td>4.2.5 Transactions with Business Components</td>
<td>4-15</td>
</tr>
<tr>
<td>4.2.6 Concurrency Control</td>
<td>4-16</td>
</tr>
<tr>
<td>4.3 OMG Object Transaction Service (OTS)</td>
<td>4-18</td>
</tr>
<tr>
<td>4.3.1 Introduction</td>
<td>4-18</td>
</tr>
<tr>
<td>4.3.2 Definitions</td>
<td>4-19</td>
</tr>
<tr>
<td>4.3.3 Service Architecture and Functionality</td>
<td>4-21</td>
</tr>
<tr>
<td>4.3.3.1 Typical Usage</td>
<td>4-22</td>
</tr>
<tr>
<td>4.3.3.2 Transaction Context</td>
<td>4-25</td>
</tr>
<tr>
<td>4.3.3.3 Synchronisation</td>
<td>4-25</td>
</tr>
<tr>
<td>4.3.3.4 Exceptions</td>
<td>4-26</td>
</tr>
<tr>
<td>4.3.3.5 Transaction Models</td>
<td>4-27</td>
</tr>
<tr>
<td>4.3.3.6 Transaction Integrity</td>
<td>4-28</td>
</tr>
<tr>
<td>4.3.4 Transaction Service Interfaces</td>
<td>4-28</td>
</tr>
<tr>
<td>5 Middleware Technologies</td>
<td>5-1</td>
</tr>
<tr>
<td>5.1 Traditional Technologies</td>
<td>5-2</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.1.1 Three-tier Architecture with TP Monitor</td>
<td>5-2</td>
</tr>
<tr>
<td>5.1.2 Three-tier with Message Oriented Middleware (MOM)</td>
<td>5-3</td>
</tr>
<tr>
<td>5.1.3 Three-tier with an Object Request Broker (ORB) Architecture</td>
<td>5-3</td>
</tr>
<tr>
<td>5.1.4 Three-tier with an Object DBMS</td>
<td>5-5</td>
</tr>
<tr>
<td>5.2 Application Server</td>
<td>5-6</td>
</tr>
<tr>
<td>5.2.1 Introduction</td>
<td>5-6</td>
</tr>
<tr>
<td>5.2.2 Web Information Server</td>
<td>5-10</td>
</tr>
<tr>
<td>5.2.3 Component Server</td>
<td>5-14</td>
</tr>
<tr>
<td>5.2.4 Object Transaction Manager (OTM)</td>
<td>5-16</td>
</tr>
<tr>
<td>5.3 Where do we go from here?</td>
<td>5-18</td>
</tr>
<tr>
<td>6 Implementation of a Business Object Framework</td>
<td>6-1</td>
</tr>
<tr>
<td>6.1 Introduction</td>
<td>6-1</td>
</tr>
<tr>
<td>6.2 Design</td>
<td>6-3</td>
</tr>
<tr>
<td>6.3 Object Model</td>
<td>6-5</td>
</tr>
<tr>
<td>6.3.1 Main Program</td>
<td>6-7</td>
</tr>
<tr>
<td>6.3.2 Configuration</td>
<td>6-7</td>
</tr>
<tr>
<td>6.3.2.1 Initialisation</td>
<td>6-8</td>
</tr>
<tr>
<td>6.3.2.2 Configuration Information: Transaction Policy</td>
<td>6-10</td>
</tr>
<tr>
<td>6.3.2.3 Configuration Information: Marshalling information</td>
<td>6-12</td>
</tr>
<tr>
<td>6.3.2.4 Configuration Information: Initial Business Object State</td>
<td>6-13</td>
</tr>
<tr>
<td>6.3.3 Resource</td>
<td>6-14</td>
</tr>
<tr>
<td>6.3.3.1 Integration of Business Logic</td>
<td>6-14</td>
</tr>
<tr>
<td>6.3.3.2 BResource Interface</td>
<td>6-18</td>
</tr>
<tr>
<td>6.3.3.3 The Role of the Resource Object during an OTS Transaction</td>
<td>6-20</td>
</tr>
<tr>
<td>6.3.3.4 Transaction Termination</td>
<td>6-23</td>
</tr>
<tr>
<td>6.3.3.5 Future Extensions</td>
<td>6-26</td>
</tr>
<tr>
<td>6.3.4 Business Object</td>
<td>6-26</td>
</tr>
<tr>
<td>6.3.4.1 BusinessObjectTIE Interface</td>
<td>6-28</td>
</tr>
<tr>
<td>6.3.4.2 De-Marshalling Requests</td>
<td>6-30</td>
</tr>
<tr>
<td>6.3.4.3 Transaction Policies</td>
<td>6-31</td>
</tr>
<tr>
<td>6.3.4.4 Locks</td>
<td>6-37</td>
</tr>
<tr>
<td>6.3.4.5 BusinessObject::delete()</td>
<td>6-39</td>
</tr>
<tr>
<td>6.3.4.6 Relationships</td>
<td>6-41</td>
</tr>
<tr>
<td>6.3.4.7 Future Extensions</td>
<td>6-42</td>
</tr>
<tr>
<td>6.3.5 State management</td>
<td>6-43</td>
</tr>
<tr>
<td>6.3.5.1 Activation</td>
<td>6-44</td>
</tr>
<tr>
<td>6.3.5.2 Deactivation</td>
<td>6-44</td>
</tr>
<tr>
<td>6.3.5.3 Future Extensions</td>
<td>6-45</td>
</tr>
<tr>
<td>6.3.6 Persistent State and Types</td>
<td>6-46</td>
</tr>
<tr>
<td>6.3.6.1 Persistent Type Hierarchy</td>
<td>6-46</td>
</tr>
</tbody>
</table>
Table of Contents

6.3.6.2 Access to Persistent Attributes 6-47
6.3.6.3 Future extensions 6-48

6.3.7 Type Manager 6-48
6.3.7.1 Design 6-48
6.3.7.2 Factory Methods 6-49
6.3.7.3 Future Extensions 6-51

6.3.8 DataStore 6-52
6.3.8.1 DataStore Interfaces 6-52
6.3.8.2 Future extensions 6-53

7 Conclusion........................................................................................................7-1

7.1 Review 7-1
7.2 Future Developments 7-2
7.3 Example Applications for Business Objects 7-3
7.4 Vision 7-4

A Appendix ........................................................................................................ A-1
A - 1 Literature A-1
A - 2 Abbreviations A-6
A - 3 Glossary A-8
A - 4 IDL Descriptions of BOF Components A-11
  A - 4.1 Business Object Framework by Kai-Uwe Schäfer A-11
    A - 4.1.1 datatypes.idl A-11
    A - 4.1.2 exception.idl A-11
    A - 4.1.3 bof.idl A-12
  A - 4.2 OMG Object Transaction Service (OTS) A-15
A - 5 Class Diagram for BOF A-20
A - 6 Source Code on CD A-21
Grammar, learning, glosses plain,
Even philosophy is vain;
Arithmetics and letters all
In Heaven’s hall God shall disdain.¹

An unknown scholar in seventh century Ireland.

1 Introduction

1.1 Enterprise Application Software Development

Being on our way to the information society of the next century, the Information Technology (IT) industry is facing higher demands for presenting and manipulating data, especially business data. New software and hardware architectures - pushed by the success of the Internet - lead to what is known as the third wave of enterprise application software development: *multi-tier distributed applications*.

During the first wave (up to the 70s) monolithic applications were built on mainframe machines. Data was very tightly integrated into the program which made it very difficult to model and reuse corporate information. With the commercial viability of Database Management Systems in the early 80s, it was possible to create data repositories that were shared by multiple programs and users. Simple two-tier client/server applications made data accessible over a network (early 80s to early 90s). Later, the increasing complexity of applications and the adoption of graphical user interfaces (GUI) lead to intelligent (thick) clients running on thousands of PC's, all trying to access a central data repository. This development gave rise to a number of problems with regard to management, scalability and maintenance of the software as well as maintenance of the database schema. The third wave of enterprise application software development established new architectures for distributed, component-based applications by means of which the IT industry now tries to solve these problems and additionally introduces new services to the customer. There are three basic architectures being developed for enterprise-wide object reuse: the Distributed Component Object Model (DCOM) from Microsoft, Enterprise JavaBeans (EJB) from Sun...
1.1.1 Multi-Tier Distributed Applications

What is called a multi-tier system actually consists of three logical parts. There's a thin GUI client talking to a middle-tier application server running centralised business logic, itself talking to a traditional DBMS server. There may be several interacting application servers on the middle-tier, each connected to a different DBMS server.

The thin client constitutes a single, logical tier of a multi-tier application whose processing is limited to the user-interface. A true thin client only does the processing related to displaying information in a user-friendly format by means of complex and highly interactive user interfaces and also the gathering of user input. The source of the data and any interpretation of that data must be handled at some other level.

The middle-tier is the playground for at least four different types of objects, which combine to form a whole application system. These are (a) business objects, (b) technology objects, (c) application objects and (d) service objects.

(a) Business objects are fine-grained objects, which represent concrete entities in the real world, like a customer, a purchase order, a product, or a billing business process. They are developed independently of their later use and are thought to serve as reusable components for many different application domains. The next chapter gives a more detailed description of this type of object, when introducing the Business Object Facility (BOF) specified by the Object Management Group (OMG), and the Enterprise JavaBeans (EJB) component model specified by Sun Microsystems.
(b) **Technology objects** represent a programming or technology concept, and thus are the building blocks of applications and implemented business objects. They are the components of the information system and application environment. Examples of technology objects include GUI components like windows and push buttons, programming constructs like string or container classes, object request broker services and databases.

(c) **Application objects** are programs, which present information and manage interaction with human users and process information. They are solutions to specific business problems and coordinate the interaction in a specific set of business objects in order to perform a specific task. Examples of application objects include order entry, quarterly report and reservation. Physically, application objects are assembled from business objects and technology objects, glued together with program code. Using client-server terms, application objects can be viewed as clients of business objects. The EJB session objects can be seen as application objects.

(d) **Service objects** don’t have any state at all. They are used for special services like calculation or printing. Their internal behavior relies on input parameters only, i.e., they do not store any state information between two client requests. In the EJB model, session objects would offer this type of service.

The third tier is also called the database tier or data storage layer and represents the database servers as they are known from the traditional client/server architecture. This layer is often divided into a transaction processing piece and the physical database. Examples for the database tier include a single database or the combination of a database and a database middleware product (e.g., JDBC middleware like Symantec’s dbAnywhere or I-Kinetic’s Databroker, or a transaction manager).
This general architecture (Figure 1-1) is now establishing itself as the dominant enterprise application software architecture for the late 90s and early 21st century. It has some important advantages, of which reuse and scalability are to be regarded as the most important:

- **Scalability.** Object-based applications can be distributed across multiple servers, allocating compute-intensive objects to servers built for the purpose, leaving the client machines to mainly manage the user interface. *Load balancing* can be achieved by replicating certain application objects. This allows for a more efficient use of resources and reaction to changing client load.

- **Reuse of business logic.** Reuse has become a buzzword, but that doesn't diminish its validity. An excellent way to deploy new applications faster is to do less work by reusing the work of others. Business objects, coupling methods and data, allow both code and data to be reused. Additionally, reused classes are more trustworthy than new ones because they have been proven in other applications.
Not only software components (source/binary) but also actual object instances in memory can be reused by two or more applications. Imagine, for example, an automatic teller machine withdrawing money from your bank account, which is represented by a CORBA object running on one of the bank's servers. Meanwhile, your spouse visits the bank's web site and retrieves information from that exact same CORBA account object. As soon as you've withdrawn money from the CORBA object, your spouse will see that withdrawal reflected in the browser. Being able to see changes immediately is just one benefit of instance reuse.

Furthermore the possibility of wrapping a legacy application by means of an object server broadens the benefit of reuse.

- **Object encapsulation.** In this new three-tiered client/server paradigm clients do not talk directly to database servers by means of interfaces to data but instead they communicate via application interfaces with objects that encapsulate data. Applications can now be designed independently from the underlying data model. The gap between an object model and a relational data model, also called the *impedance mismatch*¹, is hidden behind object interfaces.

- **Reliability.** After a system crash or during a network problem, replicated server objects can replace their temporarily unavailable counterparts.

- **Performance.** By using special hardware for selected components of the application the performance of the whole system can be improved. Network load can be minimised by distributing the business objects (and their respective databases) to the departments in which they are most frequently used.

¹ [HEU92] Page 125.
• **Maintenance.** The division of an application into many autonomous components increases the modularity and simplifies the localisation of errors. Application components can be replaced much easier by newer versions with additional features. Change does not become free; it becomes more manageable. Architecture becomes more transparent because the systems reflect the business.

Especially in an environment where Java clients are held on a centralised server, it takes very little effort, even to replace the client components. On the other hand the collaboration of those autonomous components requires a much higher amount of administration.

• **Open architecture.** Applications assembled of distributed components are open to being enlarged or manipulated according to new requirements. They can adapt quickly to structural changes of the enterprise or changes of the underlying database model.

• **Reduced complexity.** Enabling the assembly of applications from reusable business logic components reduces the complexity of application development.

• **Cross-platform support.** Business objects can interact with other objects on any other platform provided that they all talk the same language, i.e., a standard protocol like the Internet Inter-ORB Protocol (IIOP).

**1.1.2 The Role of Java**

Java-based applications combine many of the benefits of Internet technology and incorporate the strength of traditional client-server applications. They can feature rich user interface elements like windows, toolbars and drag-and-drop facilities independently of any platform. Unlike the installation and maintenance problems of client/server applications Java-based applications distribute easily in medium to high bandwidth networks (Intranet and Extranet) to all
platforms which support the necessary Java virtual machine (VM) and which can be upgraded on a centralised server. There may be a significant amount of network traffic in the initial download of a Java application, but thereafter bandwidth requirements are kept to a minimum during database access and update operations.

As SunSoft's Java Development Kit and Java development tools become mature and more efficient, Java is increasingly introduced in business projects, even playing the role of highly scalable servers. The platform-neutral component architecture *JavaBeans* is the ideal choice for developing or assembling network-aware solutions for heterogeneous hardware and operating system environments - within the enterprise or across the Internet.

*JavaBeans* is a component model for visual construction of reusable components for the Java platform. *Enterprise JavaBeans* extends *JavaBeans* to middle-tier/server-side business applications. The extensions that *Enterprise JavaBeans* add to *JavaBeans* include support for transactions, state management, security, and meta-information. As a platform-independent Java API, *EJB* components use IIOP as the underlying protocol, and they can also access existing transaction processing systems. In the *EJB* programming model, so-called "entity objects" take the role of business objects. Due to its open architecture and the features mentioned above, *EJB* application server are the best frameworks for business objects at the moment. Chapter 2 describes this technology in more detail.

The business object framework presented in this thesis is implemented in Java, but does not conform to the *EJB* specification, because its design is older than the first release of *EJB*. Nevertheless, the idea and the technology behind it turn out to be very much the same.
1.1.3 The Role of CORBA

Standards are more important for distributed objects than for any other technology in any other industry. Objects from one company must be able to communicate and cooperate with objects from other companies.

Roger Sessions
Object Persistence

Roger Sessions’ statement perfectly points out the most important goal of the OMG. Since 1990 the Object Management Group has constantly been developing standards for inter-object communication that should enable a market for software components, which seamlessly work together, regardless of the programming language they are written in or the company which sells them. The Common Object Request Broker Architecture (CORBA), the Internet Inter-ORB Protocol (IIOP), the Object Management Architecture (OMA), and finally the specifications around the Business Object Facility (BOF) have been specified to meet this goal. Besides, there are specifications for additional services that can be reused by any CORBA component, e.g., naming, security, transaction, event, and many other (15 services are now specified), and that add to a rich application infrastructure.

The final goal of an open market for software components has not been reached yet. This is mainly because the CORBA specification has not been complete enough to enable seamless interoperability and portability among different ORB products. An interoperable directory service and the Portable Object Adapter (POA) will considerably improve the situation. The greatest expectations, though, lie in the combination of CORBA and Java. Software components that are implemented based on these two technologies are not only easily portable among different ORBs, but, what is more, they are also portable among any operating system platform, which supports a Java Virtual Machine. This expected high portability has lead to the development of many CORBA-based EJB application server

[SES96]
products since the end of 1998. Today, EJB servers seem to be the preferred technology for business components. Microsoft and the OMG both have their own architecture standard for business components, but Microsoft is limited to its Windows NT operating system, and the OMG is still struggling with their time-consuming specification process. For the implementation of the business object framework, the open standard CORBA and the portable programming language Java happened to be the best choice. The first submissions to the Common Business Object (CBO) specifications served as a guideline for the design of the required interfaces. Chapter 2 describes these specifications in more detail.

1.1.4 Containers for Business Logic and Data

Business software components are usually embedded in a rich infrastructure of basic services. By nature they require sophisticated and scalable transaction, session, and state management, which adds significant complexity to today’s applications. Instead of having to write all this low-level code themselves, application developers can now rely on services offered by so-called “containers” or “component servers” that host the business components. This increases reuse, portability, and maintainability. For instance, not each single component has to implement the database access itself, but rather uses a database component with a standardised, database independent interface, or even relies on transparent persistency offered by the container. This is especially important for standard business components that are sold to several companies with completely different database environments. Besides, a separate, optimised database component can increase scalability by pooling database connections and caching data.

Up to now, the most common use of software components has been in the area of user interfaces or as stand-alone components connected by an Object Request Broker (ORB). The business components (business objects) described in this thesis are different in that they
cannot exist without a "host" or "container", which offers them persistence, transactions, state management, and other services. The number of active instances of such a component is hundreds or thousands, rather than one or two. This is because these business entities represent actual database objects. The concept of business objects is explained in chapter two.

The new concept of business components that "live" in standardised containers leads to a paradigm shift from custom crafted enterprise systems to component-based enterprise systems. These component-based enterprise systems will be assembled from plug and play components using entirely new tools. With the support of appropriate tools and components, business system developers can then concentrate on the business solution without being concerned about issues of distributed processing, database design and sharing of objects.

The prototype implementation of a business object container is presented in chapter six. It is based on the research done in the areas of transaction, persistence and ORB technology (mainly concurrency and state management).

1.2 Structure of the Document

1) Introduction

This chapter gives an overview of current software development technology in the domain of business applications. This includes a multi-tier architecture, Java and CORBA technologies, and the notion of a container for business components.

2) Business Object Components

In order to be "pluggable" into different server environments and to be interoperable between independently developed business objects, the business components have to be
capable of describing themselves (in Java called "introspection"). In addition, a standard syntax and semantics of interfaces both of components and containers have to be specified. This will lead to a greater simplicity in building, using and deploying business objects. Currently, the most complete specification is the EJB specification by Sun Microsystems. The OMG is still working on an equivalent for CORBA components. The two standards are delineated in chapter two.

3) Persistence of Business Objects

From the viewpoint of the application developer, a business object represents a number of values in one or more databases. Its state persists between different method calls and is shared among multiple concurrent client requests. In fact, this is the key advantage of keeping business logic on the middle-tier. Chapter three describes the peculiarities of persistence in Java and CORBA and suggests a way of managing persistence of distributed objects transparently. Furthermore, the architecture of a persistency layer is introduced. A separate application layer for persistence decouples the business object from specific database systems and schemas and thus increases portability and maintainability of business components.

4) Distributed Transactions

The business logic is mostly processed within transaction boundaries defined by a client application. Also, a single transaction often covers different distributed business components, enabling a whole business workflow to be processed on the middle-tier. These transactions require a separate transaction service to perform the two-phase commit protocol on all partaking business components. Chapter four introduces the basic concepts of transactions in general and distributed transactions in particular. It describes the OMG Object Transaction Service (OTS), which constitutes an open standard for distributed transactions on software
components. Inprise's OTS implementation is part of the Business Object Framework that has been implemented for this thesis.

5) Middleware Technologies
Business object containers deal with persistence and transactions, concurrent access (multi-threading), security (encryption, authentication), scalability (load balancing), integration of web technology, and integration of legacy systems. These services have long been features of transaction processing (TP) monitor products. Today, vendors of TP monitors, CORBA runtime systems, Web server systems, database systems, and application servers are building or adapting products that function as full-featured component servers, often called "Object Transaction Managers" (OTM). Chapter five introduces various products in this area.

6) Design and Implementation of a Business Object Framework
Chapter six finally describes the design and implementation of what has been introduced as a "container" for business objects. It is a framework of services that manage transparent persistence, distributed transactions, life cycle, and concurrent access. Experiences gained during the research of persistence and transaction mechanisms, as well as other middleware concepts, found their way into the present implementation. The CORBA interfaces are based on submissions to some specifications for OMG Business Object Components.

7) Conclusion
The last chapter reviews the design of the business object framework. It describes what has been achieved in this thesis and displays a vision of future developments in the software market. It presents example applications for enterprise business objects and their impact on the users and the IT industry.
2 Business Object Components

If objects are to be assembled, they must be compatible with one another. This is rarely a problem when writing a single program because all the objects are written in the same language, run on the same machine, and use the same operating system. But building entire information systems out of objects is a quite different matter. Objects have to interact with each other even if they are written in different languages and run on different hardware and software platforms.

David Taylor

Business Engineering with Object Technology'

2.1 Introduction

Global business competition and increasing complexity of business processes has created an environment of continuous business structure change. Short life cycles, rapid solution delivery and product customisation demand a high flexibility to adequately support business change. The challenge is to provide rapid implementation of accurately specified business requirements.

"Mechanisms for achieving these goals are likely to utilise concepts of componentisation, model-based specifications, and end-user solution composition. Such composition may be achieved through 'business objects' as self-contained and independently-developed 'application components' which can be used in different combinations at different times."

The development of specifications for business object components aims at the realisation of a long-held vision of a marketplace of interoperable, self-describing business objects. Several companies are making significant investments to meet this goal, including Oracle's redesign of its application suite around Enterprise JavaBeans (EJB) and IBM's San Francisco project. This chapter introduces two approaches of providing a standardised infrastructure for

1 [TAY95]
2 [BUS98a] Page 11.
business components: the OMG Business Object Facility and Sun’s Enterprise JavaBeans. The OMG has not been very successful in producing a widely accepted specification. It looks as if the OMG would rather adapt the EJB solution to CORBA components than provide an own solution (nevertheless, the present implementation of a business object server is based on ideas from early proposals to OMG RFPs, because the first EJB specification came out some time after the design phase). Sun has managed to create an environment where an industry for third-party tools and platforms has begun to flourish. This is the reason for the great success of EJB. In contrast, Microsoft’s competing technology is not only confined to the Windows NT platform, but there are also very few viable development tools besides Microsoft’s own developer studio products and the Microsoft Transaction Server.

**Business Objects are not DBMS Tables**

Business objects may, at first, seem not much more than a representation of a table in a relational DBMS, since tables also represent business information. In some simple cases there may be a correspondence between a business object and a database table. But in most cases, the business object will implement rules and processes beyond the capability of a DBMS. They may combine multiple tables, or manage information that is not even stored in a DBMS (like online stock price quotations). Business objects represent multiple tables, processes and rules at a higher level than the DBMS table.

**A New Concept of “Application”**

Business objects encapsulate the information and rules associated with that object and its relationships to other objects. Some business objects may “wrap” existing legacy applications. Other objects may be implemented using workflow tools or 4GLs. Each functionality, legacy application or database will be encapsulated in business objects. With a system composed of a set of co-operative business objects, the outmoded concept of monolithic applications becomes irrelevant. Instead, an information system is now composed of semi-autonomous
but co-operative business objects which can be more easily changed. This type of component assembly and reuse has been recognised as a better way to build information systems. An application, in terms of business objects, becomes a set of co-operative business objects combined to facilitate business processes.

**General goals of business object components**

**Business Goals:**

- The ability of business systems to adapt rapidly to changing business requirements.
- A reduction in cost and increase in the effectiveness of information systems.
- An open marketplace for business components.
- Mainstream use of distributed object technology for commercial data processing.

**Usability Goals:**

- Application development through component assembly and reuse.
- Simplicity in the development, deployment, change, and the use of business objects for application users and developers.
- Implementation of frameworks of business objects as “substitutable business components” of an information system.
- Enable high-level visual tools.

**Technical Goals:**

- Interoperability of independently developed business objects.
- Isolation of infrastructure and business objects from tool or presentation technologies (Multi-tier architecture).
- Isolation of technology from business logic.
- Transactional integrity across distributed business objects.
- Direct coupling between a business object and its defining meta-object (reflective system).
2.2 OMG Standards

The Object Management Group is still working on a specification for business objects, which constitutes a technological infrastructure to support "plug and play" business application components. The infrastructure must have the semantics to support common and enterprise specific business objects. The so-called Business Object Facility is based on the following definition:

"The infrastructure (application architecture, services, etc...) required to support business objects operating as co-operative application components in a distributed object environment."

The OMG's Business Object Domain Task Force (BODTF) requested two proposals for a Business Object Facility: the Business Object Component Architecture (BOCA), which defines a meta-model for business objects, and the Business Object Interoperability Framework, which is a standard for interoperability, component assembly, open systems, and ease of use.

![Diagram showing the Business Object Facility and its components]

Figure 2-1. The Business Object Facility provides a "higher level of abstraction" as a layer between business objects and the underlying technology. See [BUS98a].

The Business Object Facility represents the information system semantics required to support business objects. It provides an abstraction level in the form of extended application semantics and a technical support framework, to support business objects and business object specifications (see figure above). Common Business Objects represent the business semantics that may be found in any enterprise, such as the concept of a “purchase”. Financial, Manufacturing, and Other Business Objects represent common objects of a domain, such as manufacturing.

2.2.1 The Business Object Component Architecture (BOCA)

The fundamental unification point for business objects is the application architecture represented by the BOCA Meta-Model. The BOCA Meta-Model describes the constructs and types that are used to build a business object system. Without the BOCA, CORBA components provide a way to “snap together” implementations, but no business application architecture to snap them into.

![Diagram](image)

Figure 2-2. Elements that are part of a business objects solution.

The above diagram illustrates the pieces that must “fit together” to provide a coherent solution for business objects. These elements are:
• **Business Object Component Architecture (BOCA).** The BOCA Meta-Model describes constructs and types that are used to build a business object system.

• **Specification Language.** The Component Definition Language (CDL) offers a simple way to define publishable business object standards.

• **IDL Mapping.** The IDL mapping specifies the form and content of business object interfaces based on the meta-model.

• **Unified Modelling Language (UML).** BOCA is aligned as a UML extension. Given a UML design, the BOCA provides a way to express and realise the design as CORBA based distributed business objects.

• **Interoperability Framework.** This specification defines interfaces and protocols for the technical interoperability of business objects.

• **CORBA + IDL.** The CORBA Meta-Model, ORB and IDL are the basis on which the BOCA and framework is built.

• **CORBA Services.** Supporting the framework are the library of CORBA services used by business objects in well defined ways.

• **CORBA Components.** When CORBA-Components come on-line, the IDL mapping can be extended to utilise CORBA components, achieving true “plug and play”, an absolute necessity for the component marketplace to flourish.

### 2.2.2 Interoperability Specification

This section gives an overview of the nature of business objects and the collaborating frameworks in which they “live”. Figure 2-3 illustrates the fundamental aspects of business object interoperability. The diagram depicts two different business object implementations or “frameworks” which could be provided by different vendors, executing on different platforms and written in different languages. Each framework contains one unique instance of an object type manager. In general, business objects are transactional and persistent. The
business objects and type managers support a number of interfaces and protocols by which they interoperate. These interfaces and protocols are listed in the middle of the diagram. Business object frameworks also rely on several CORBA services shown below.

![Diagram showing interoperability by interfaces, protocols and shared services.](image)

The following list shows the main features, which provide the technical foundation for business object interoperability:

- **Object Identity.** A business object, and the entity it represents, has a unique existence and identity. Consequently, business objects are never passed by value. Business objects may be moved and the move operation will relocate the unique identity.

- **Meta-Objects.** Each business object has a corresponding meta-object, which represents its type and features. *Features* are state declarations (persistent state), attributes, relationships, operations, signal events and apply statements. The values of the type and feature parameters become attributes in the meta-objects that represent each type, giving it "introspective" capability.
• **Managers.** In OOP languages a "static" scope is often used that contains constructors, attributes and operations that effect the entire type, not just one instance. The concept of static scope does not exist in a distributed system, therefore the BOCA introduces *type manager objects* for each single type. Type managers are CORBA objects that contain constructors, attributes and operations that may effect the entire type, not just one instance. One of the jobs of managers is to be the *instance factory."

• **Event Model.** Business objects allow other objects to register in events produced by attributes, operations, relationships and user-defined signals. This allows components to have dependencies on other components that were not anticipated when they were implemented.

• **Business System Domain.** A business system domain (BSD) combines several business object types belonging to a particular business domain. The scope of a BSD is defined by the practical size of a consistent, integrated model as well as the scale that can be effectively managed for both development and operation. It is identified by a unique domain manager object, which provides BSD identity information and a reference to the BSD Naming Service.

### 2.2.3 Definition of Business Objects

A business object is a specialised CORBA object, which is identifiable, transactional and persistent, and has attributes, states, relationships, operations, exceptions and event mechanisms. See [BUS98b] and [SIM94].

• **Identifiable.** As a business object is just a normal CORBA object, its identity is represented by a CORBA object reference. Because of its unique existence a business object is never passed by value. It can be moved, but this will cause the unique identity to be relocated. It can also be copied, but the copy must have a new unique identity. One
could think of an implementation that supports replication of business objects for improved performance but this has to be transparent for the developer and other business objects, assuring that there is only one recoverable state for an object.

• **Transactional.** In general, business objects are accessed in a transactional context. This requires concurrency control mechanisms to maintain the integrity of the system state. Business object frameworks hide the details of transactional mechanisms from the developer. This means that the transactional context is passed transparently simplifying the way methods are invoked.

• **Persistent.** Most business objects require their state to be made persistent. Persistence should be transparent to business object developers and operations related to persistence are therefore not visible in the interface of a business object. Requested objects are automatically activated (brought into memory), updated with persistent state out of a database and deactivated (garbage collected) as required by performance considerations and operations being performed.

• **Attributes.** These are associated, transient or persistent data elements, which are accessible through get- or set-methods ("accessor" methods) of defined forms. These methods are expected to incorporate related business object functionality like transaction handling, events, exceptions or integrity mechanisms.

• **States.** A business object can have defined states. The formal state of an object is expressed by the current values of one or more state variables. A formal state variable is implemented as an attribute with enumerated values.
• **Relationships.** Generally, business objects have relationships with other objects. Relationships may be one-to-one or one-to-many, and they may be bi-directional or one-way. Relationships are encapsulated and managed so that their integrity is assured, i.e., when one object removes the relationship, the other object is updated accordingly. Relationships only occur between business objects within a business system domain.

• **Operations.** Like other CORBA objects, business objects perform operations defined in their IDL interface. Operations are expected to be performed in a transactional context.

• **Events.** Each business object can provide notification of selected events on request. This allows a dependency between two independently developed objects without any special programming. Events may also be requested from all instances of a type and its sub-types by requesting notification from the respective type manager. Selected events may be directed to an event service.

Business objects have three types of events: (1) intrinsic events exist for changes of state, (2) implicit events occur due to changes of attributes or relationships, or when operations are invoked, completed, or result in failure, and (3) programmed events are those which are declared in the interface specifications and generated by business logic.

Business objects are shared within heterogeneous frameworks. They interoperate across different platforms, operating systems, databases and languages. Some computational services (e.g., naming, events, transactions, and persistence) are integrated into business object frameworks to relieve the business object developer of a number of technical complexities. This will reduce the cost and risk of developing distributed systems.

Business objects are developed independently of their later usage and fit into different contexts. These multi-purpose components will create a market in common business objects.
They will allow solutions to be delivered more quickly and with a high quality, because the repeated use of the same components will justify investment in a robust design and a quality implementation.

2.2.4 Object Relationships in the BOCA Environment

According to David Taylor\textsuperscript{1}, objects relate to each other in one of the three basic ways:

1) \textit{Specialisation}. Classes inherit properties of superclasses and specialise the super-type.

2) \textit{Collaboration}. Objects interact with each other by sending messages and requesting services.

3) \textit{Composition}. Objects are constructed out of other objects.

In the BOCA context, composite objects have a slightly different meaning, they cannot be independent business objects with a unique identity.\textsuperscript{2} Thus, relationships between business objects refer to the term \textit{collaboration}.

Relationships may be one-to-one or one-to-many. Many-to-many relationships are not allowed in the BOCA environment but they can easily be represented as two one-to-many relationships. Only binary relationships are allowed. N-ary relationships can always be implemented as binary relationships.

Furthermore, relationships may be one-way or bi-directional. One-way relationships can only have cardinality one. Bi-directional relationships are typically "managed" so that references remain consistent. If two Business Objects are in a \textit{managed, bi-directional relationship}, when one object removes the relationship (e.g., when being deleted), the other object is updated.

\textsuperscript{1} [TAY95] Page 75.

\textsuperscript{2} Composition is roughly the same as aggregation in object modeling terms. However, aggregations would be implemented as relationships between independent objects, which have their own (persistent) identity.
accordingly. If one object is made persistent, all dependent objects are recursively made persistent as well (*persistence by reachability*).

Relationships between objects can exist across different address spaces, i.e., different business object frameworks within a business system domain (BSD), and across different heterogeneous databases. Relationships between Business Objects should not cross BSD boundaries (such relationships would imply that the two domains comprise a single, consistent model).

Relationships between business objects are encapsulated and managed so that their integrity is assured. Access to relationships is provided through accessor methods. Access to members of one-to-many relationships is provided by iterators, which complement the accessor methods and maintain the integrity of the relationship.

### 2.3 Enterprise JavaBeans (EJB)

The EJB architecture is a server-side component model for the Java platform. It has the same purpose as the OMG standards, i.e., to simplify implementation, configuration and deployment of business components by relieving the developer from writing "plumbing" code. For example, the enterprise developer no longer needs to write code that handles transactional behaviour, security, connection pooling, or threading, because the architecture delegates this task to the server vendor. In addition, EJB components are portable to any Java platform.

---

1 [HEU92]
2.3.1 EJB Design

The EJB design relieves the developer as much as possible from any system-level programming and allows him/her to focus solely on writing business logic. This is accomplished by turning features that ordinarily would need to be hand-coded into simple *declarative properties* of the enterprise Beans. Certain runtime characteristics like security and transactions do not have to be implemented in code, but are represented as “flags” on the Bean. These characteristics are set at deployment time and interpreted by the container the EJBs are “living” in.

**EJB Containers**

An EJB server must implement EJB *containers*. EJB components are assigned to containers when they are deployed. Once deployed, the container is responsible for managing the EJB object life cycle, co-ordinating transactions, and accomplishing some other system-level functionality. This is done transparently to the client by intercepting client requests and delegating work to the EJB object. In the case of entity Beans, multiple clients may access a single entity object concurrently; it is the container’s responsibility to synchronise access, using a transaction service.

**Application Models**

There are two fundamental models for building enterprise applications. In the first model, the client begins a *session* with an object that acts like an application, executing a unit of work, possibly including multiple database transactions and requests to other distributed components. In the second model, the client accesses an object that represents an *entity* in a database, in this thesis called a “business object”.

---

1 For more details, read [SUN99].
2.3.2 Session Beans

A session Bean is an object that represents a transient conversation with a client, and it executes database read and write operations for the client. These database access operations can be in the context of a transaction. An instance of a session Bean is private to the client that created it and is not shared among multiple clients. It contains the state of the conversation in transient fields. This implies that if either the server or the client crashes, the session Bean is gone and a previously started transaction has to be rolled back.

Session Beans may be stateful or stateless. Stateful session beans have a unique identity that is assigned by the container at create time. They contain client-specific conversational state. Stateless session Beans do not have conversational state. All EJB objects of the same stateless session Bean have the same object identity. The term “stateless” signifies that an instance has no state for a specific client. However, the instance variables of the instance can contain the state across client-invoked method calls. Examples of such states include an open database connection and an object reference to another EJB object.

2.3.3 Entity Beans

An entity Bean represents data in a database, and logic to act on that data. In a relational database context each entity Bean consists of rows in one or more database tables. Entity Beans are transactional, and are long-lived. As long as the data remains in the database, the entity Bean exists. This model is typically used in object-oriented databases and introduced to the RDBMS world by object-to-relational mapping tools.
Unlike session Beans, entity Beans must be assigned a unique identity when they are created and provide a mechanism that enables a client to locate the desired Bean instance based on its key (primary key). Multiple clients may access a single entity object concurrently; it is the container’s responsibility to synchronise access, using a transaction service.

**Persistence of Entity Objects**

There are two models of persistence in entity EJBs. An EJB can manage its own persistence, in which case the EJB developer has to write the database access logic himself. Alternatively, the developer can delegate the management of the Bean’s persistence to the container. Using container management insulates the EJB class from needing to know about the data source in which the entity is stored. It also reduces the coding work for EJB developers, who then can completely avoid writing data retrieval code.

**2.3.4 EJB Transactional Characteristics**

EJB support flat transactions, modelled after the OMG Object Transaction Service (OTS). EJBs participating in a transaction can atomically update data in multiple databases that may be distributed across multiple sites. The EJB container bears the responsibility for managing transactions. The transaction context is transparently passed to any other resources that are invoked as part of the transaction. There are six valid transaction attributes that can be assigned to an EJB:

- **TX_NOT_SUPPORTED** indicates that the EJB should not participate in a transaction. The invocation of any method of the EJB should occur outside the scope of a transaction.
- **TX_BEAN_MANAGED** allows the client to control transaction demarcation manually. Only EJBs with this attribute can be manually associated with a transaction.
• **TX_REQUIRED** indicates that any invocation of a method on the object must be associated with a transaction. If there is no transaction active, the container will start a new one before delegating the method call to an EJB.

• **TX_SUPPORTS** does not demand a transaction context from the client. If there is a transaction in progress, the EJB will participate; if not, the container will invoke the method without a transaction.

• **TXQUIRES_NEW** always results in a new transaction to be started before delegating the invocation to the EJB method. If there is already a transaction in progress, it suspends when the new transaction starts and resumes when the new transaction completes.

• **TX_MANDATORY** requires the client to start a transaction before sending a request to the EJB. If a client attempts to invoke such an EJB outside of the scope of a transaction, the container will throw a TransactionRequired exception to the client.

Besides the transaction type, the EJB provider also has to define the transaction isolation level. The available isolation levels correspond to the isolation levels defined by JDBC: **TX_READ_COMMITTED**, **TX_READ_UNCOMMITTED**, **TX_REPEATABLE_READ**, and **TX_SERIALIZABLE**.
3 Persistence of Business Objects

Business Objects are real-world entities like customer, invoice or account. They are the key entities in a business process and, requiring reliability and durability of business activities, they have to be made persistent in a database system. This chapter deals with various issues around persistence including database management systems, database middleware software and distributed component technology. Both, the increasing influence of Java as the preferred programming language for server environments, and the fact that the business object framework developed as part of this research project is based on some Java-specific concepts, demand a section about Java on its own.

3.1 Comparison of Relational and Object DBMS

While most business applications today are developed in an object-oriented programming technology, most of the business data those applications need to access lives in a very different universe – a relational database. In such environments the introduction of object-oriented development creates a fundamental mismatch between the programming model (objects) and the way in which data is stored (relational tables). This problem is sometimes referred to as the impedance mismatch between application objects and relational data. Nevertheless these two technologies coexist beside each other, both confirming their significance for business applications. The following three sections give a short description of different approaches to making objects persistent.

3.1.1 Relational Database Management Systems (RDBMS)

RDBMS are currently the dominant database structure in most organisations. They are compliant to industry standards and they support open systems using SQL. Relational
Persistence

database systems are mature in robustness, performance, security and reliability. They even support special hardware to enhance scalability. Regardless of this maturity RDBMS are no longer sufficient for new types of Internet applications. A new order of magnitude of client numbers require the installation of some kind of highly scalable database middleware that manages a data cache and replicates server processes that run business logic.

RDBMS also lack of a data model for complex objects including images, documents, video, audio, animation and composite objects. These types of objects are much better addressed by object-oriented databases. The most important drawback, however, is the impedance mismatch between relational data models and object models.

3.1.2 Object Database Management Systems (ODBMS)

ODBMS effectively manage complex objects and relationships. They provide an efficient way of navigating through large object structures, which is hardly possible with an RDBMS. The object model can directly be mapped to the database, thus providing an easier way of programming an application, enhancing flexibility, productivity and modularity. However, object databases can be harder to maintain because a change in data objects can require major changes in the application. A better solution would be a separate persistence layer that maps the database schema to application objects (see section 3.4). This also applies to RDBMS.

One major drawback of ODBMS is the lack of support for an industry-standard Object Query Language (OQL). Instead of adopting the ODMG specification of the OQL most ODBMS vendors stick to their own proprietary protocols and APIs. Moreover, ODBMS are not as mature as RDBMS yet. They lack of robustness and scalability in multi-user environments. They are less suitable for extensive query processing which is the main type of application in business systems. Instead, ODBMS are the preferred systems for applications
where navigation through objects is needed, e.g. graphic design (CAD), or knowledge representation. Even when object databases become reliable and tested technology, relational databases will always be better suited for certain types of data. So there is an ongoing need for a connection between relational and object data. This leads to object-relational mapping.

### 3.1.3 Object Relational Mapping

To resolve the impedance mismatch, a mapping between the application objects and the relational data must be established. Creating an object model from an existing relational database schema is often referred to as reverse engineering. Conversely, creating a relational schema from an existing object model is often referred to as forward engineering. In practice, over the life-cycle of an application, forward and reverse engineering need to be combined in an iterative engineering process to maintain the relationship between the object and its relational data representations.

Mainly since 1997 a number of tools have emerged which can be used to establish such object-relational mapping processes. They often consist of a schema capture component (a utility to read a relational database schema and record it in a format that can be used by a schema mapper) and a schema mapper component (a tool to establish and maintain mappings between the object model and one or more relational database schemas). These object-relational (OR-) mapping solutions provide further important features like caching, event management, high scalability, CASE tool integration and integration of component architectures (CORBA, DCOM, EJB, RMI). There are also some OR-mapping tools without support for distributed objects or components that can be used to simplify implementation of business objects on the middle-tier (Table 3-1).

---

1 Relational database management systems lose navigation information during the normalisation process and are therefore not suitable for applications that require navigational access. See also [LAU96] p. 61 and [HEU92] pp. 110, 118, 89.
<table>
<thead>
<tr>
<th>Title</th>
<th>Components</th>
<th>Lang*</th>
<th>DOM**</th>
</tr>
</thead>
<tbody>
<tr>
<td>CocoBase Enterprise (Thought, Inc.)</td>
<td>- JavaBeans components for database access</td>
<td>Java</td>
<td>CORBA</td>
</tr>
<tr>
<td></td>
<td>- application server</td>
<td></td>
<td>RMI</td>
</tr>
<tr>
<td></td>
<td>- UML object modeling tool Together/J</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JavaBlend (Sun Microsystems)</td>
<td>- ODMG-conform API</td>
<td>Java</td>
<td>CORBA</td>
</tr>
<tr>
<td></td>
<td>- integration with object modeling tool JavaPlan</td>
<td></td>
<td>RMI</td>
</tr>
<tr>
<td>J-Database Exchange (JDX) (Software Tree, Inc.)</td>
<td>- API</td>
<td>Java</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>- forward and reverse engineering tools</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JdbcStore (LPC)</td>
<td>- workbench to define mappings</td>
<td>Java</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>- runtime environment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ontos* Integrator (Ontos, Inc.)</td>
<td>- tool for design and generation of data components</td>
<td>Java</td>
<td>VB</td>
</tr>
<tr>
<td></td>
<td>- data component server for NT</td>
<td></td>
<td>C++</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DCOM</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(MTS)</td>
</tr>
<tr>
<td>Relational Object Framework</td>
<td>- tool maintains mappings between JavaBeans and database definitions</td>
<td>Java</td>
<td>EJB</td>
</tr>
<tr>
<td>JavaBeans Edition (Watershed Technologies)</td>
<td>- runtime classes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtleware Database Connectivity</td>
<td>- API similar to ODMG-93</td>
<td>C++</td>
<td>-</td>
</tr>
<tr>
<td>(Subtle Software, Inc.)</td>
<td>- forward and reverse engineering tools</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOPLink for Java 2.0 (The Object People)</td>
<td>- Java class libraries</td>
<td>Java</td>
<td>EJB</td>
</tr>
<tr>
<td></td>
<td>- visual mapping editor</td>
<td></td>
<td>Smallt.</td>
</tr>
<tr>
<td>Visual BSF (Objectmatter, Inc.)</td>
<td>- ODMG 2.0-based API</td>
<td>Java</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>- visual mapping tool</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Supported language (Java/C++/Smalltalk)  
** Integration into any Distributed Object Model (CORBA/RMI/DCOM/EJB)

There are basically two approaches to defining how relational data maps to application objects: the table-type approach and the object modelling approach. With the first, each row in a table represents an object instance, and each column in the table corresponds to an object attribute. This literal translation between relational data representation and an application object is simple but offers little flexibility. By contrast, the object modelling approach represents a relational database schema in a true object-oriented model with all possible relationships between the objects and independent of the underlying data format. This enables a transparent integration with one or more physical databases and increases the maintainability by means of a separate schema mapping layer.

The advantages of OR mapping toolkits are:

- Providing database independence and portability by means of a common API.
- Wrapping SQL functionality in an object-oriented API.
- If the public interface of the class libraries is separated from the data access layer, data mappings can be modified in response to changes to the relational database structure without requiring application recompilation.
- Some tools generate code from an ODL\(^1\)-like description or even from the repository of an integrated object modelling tool.
- Some tools maintain persistent relationships between objects whose data reside in different databases.
- Some tools deal with object caching and transaction management.
- Some tools deal with component technologies and serve as a persistent container for components (e.g. EJB).

### 3.2 Persistence of Java Objects

This section deals with issues associated with the persistence of Java programming language objects. Serialisation, JDBC, Reflexion and Virtual Machine are Java-specific concepts for persistence that help establishing Java in server-side application components.

#### 3.2.1 Java Serialisation

Typically object serialisation is used to store copies of objects in a file, or to copy the state of objects to code running in another Java virtual machine. Serialisation enables the flattening of objects into a stream of bytes that, when later read, will recreate objects equivalent to those

\(^1\) Object Definition Language. Enables language-neutral definition of data structures for ODBMS. Specified by the ODMG-93 group.
that were written to the stream. Object serialisation is part of the Java Developer Kit (JDK) since release 1.1.

Serialisability of a class is enabled by the class implementing the java.io.Serializable interface. The programmer does not have to write special code to allow a class to be serialised but it is possible to customise this process. The default mechanism or the programmer himself implements writeObject and readObject methods that write (read) each field of the object.

The following example creates a new Employee object and writes it to a file output stream:

```java
Employee e = new Employee("John Smith", 99, 75000, null);
FileOutputStream f = new FileOutputStream("tmp");
ObjectOutput s = new ObjectOutputStream(f);
s.writeObject(e);
s.flush();
```

Reading an object from a stream is equally straight forward:

```java
// De-serialise an Employee from a file.
FileInputStream in = new FileInputStream("tmp");
ObjectInputStream s = new ObjectInputStream(in);
Employee e = (Employee)s.readObject();
```

Serialisation provides a simple yet extensible mechanism for storing objects persistently. The Java object type and safety properties are maintained in the serialised form. Serialisation should be sufficient for applications that operate on small amounts of persistent data and where reliable storage is not an absolute requirement.

Because Serialisation has to read/write entire graphs of objects at a time, it works best for a small number of objects. When the byte stream is a couple of megabytes in size, one may find that storing objects via Serialisation is too slow, especially if the application is doing frequent
Persistence

 updates that need to be saved. Another drawback is the lack of undo or abort of changes to objects. Finally, Serialisation does not provide reliable object storage. If the system or application crashes during the `writeObject` call, the contents of the file will be lost. To protect against application or system failures and to ensure that persistent objects are not destroyed, the persistent file has to be copied before each change is saved.

In conclusion, Serialisation is not the optimal choice for applications that have to manage megabytes of persistent objects, that are frequently updating those objects, or that want to ensure the changes are reliably saved in persistent storage.

3.2.2 Java Database Connectivity (JDBC)

Modelled after X/Open's SQL CLI (Client Level Interface) and Microsoft's ODBC abstractions, Java database connectivity (JDBC) aims to provide a database connectivity mechanism that is independent of the underlying database management system (DBMS). The JDBC standard, defined by JavaSoft (a division of Sun Microsystems), defines classes that represent constructs such as database connections, SQL statements, and result sets. With JDBC, a Java program can execute SQL statements and process results. To become JDBC-compliant, drivers need to support at least the ANSI SQL-2 entry-level API, which gives third-party tool vendors and applications enough flexibility for database access.

There are essentially four different types of JDBC drivers defined by Sun Microsystems. The following are descriptions of each:

1) The type 1 driver is a JDBC-ODBC bridge which provides JDBC access to most ODBC drivers. This allows off-the-shelf ODBC drivers to be used, enabling JDBC to leverage the database connectivity provided by the existing array of native ODBC drivers. The JDBC-ODBC bridge is being offered by JavaSoft as part of the JDBC package. One
drawback however in this approach is that the database driver must be loaded and configured on each client machine (via the Control Panel). As ODBC itself is not very efficient, the JDBC-ODBC bridge causes an additional and often unacceptable performance overhead. The JDBC-ODBC bridge may not be suitable for applets in a browser, as the browser vendor may not have implemented it due to Internet security issues.

2) The type 2 driver is considered partially native and partially Java. The driver can convert JDBC calls into calls for the database client API (i.e. Oracle, Sybase, etc.) This driver also requires native code on the client side, similar to the type 1 driver.

3) The type 3 driver is defined as an all Java driver which by means of a particular network protocol (e.g. IIOP) sends JDBC calls to a middleware server which then translates them into a database management system protocol. Such a database middleware solution reduces the driver size on the client side and isolates the clients from database specific driver version changes or administrative DBMS upgrades. It provides access to many different DBMSs over one single middleware server interface, even access to Intranet databases behind a firewall (by HTTP tunnelling or IIOP), and enables better scalability due to built-in TP-Monitor technology in the middleware server. Furthermore, this type of driver goes one important step forward towards component based database access. An example of this driver type is I-Kinetics' Databroker (see Figure 3-1).

4) The type 4 driver is a native protocol, *100% Pure Java* driver. This allows direct calls from a Java client to a DBMS server. Because it is written in *100% Pure Java* it requires no configuration on the client machine other than telling your application where to find the driver. However, due to the strict Java security mechanisms an "untrusted" applet can connect only to the host from which the applet was downloaded. The advantage of this driver type is the performance benefit gained from eliminating the ODBC layer.
3.2.3 Database Middleware Tools

Most of the OR-mapping tools listed in Table 3-1 are designed for Java environments. Java is on the best way to conquer the server market, most middleware products already build on Java. Why is this? One reason is surely the ease of use in comparison with C++.

Furthermore, Java is a purely object-oriented language. Another reason is the ability to work on meta-information about classes, their attributes and methods. Any compiled class can be examined during runtime and so enable a schema mapper to apply a mapping without even changing the source code of the class. This is called Reflection. Java also provides the possibility to declare some attributes of a class transient and others persistent. Surely not the last argument for Java as a programming language for server-side persistent business objects is the industry-standard database access API JDBC, which provides a modern (in comparison with ODBC) and truly object-oriented programming interface. Therefore, Java seems to be the ideal language for database middleware such as OR-mapping toolkits, application servers or component servers.

\[\text{Figure 3-1. I-Kinetics' Databroker as an example for a database middleware architecture.}\]

1 The reflection functionality in Java is mainly used for querying information about JavaBeans during runtime.
3.2.4 Persistence-capable Java Virtual Machine

In the "Pjama" project¹, which is a collaboration led by Professor Malcolm Atkinson at the Department of Computing Science, University of Glasgow, Scotland and Dr. Mick Jordan at Sun Microsystems Laboratories in Mountain View, California, they built a modified Java Virtual Machine (VM) that provides persistence according to the following principles:

- **Orthogonal Persistence.** Every component of an application should have the same rights to longevity. All data irrespective of their type can be made persistent. This includes primitive data types as well as all the classes in the Java runtime environment.

- **Transitive Persistence.** All data needed to correctly interpret an object must be retained for as long as that object exists. In other words, any object that is directly or indirectly referenced by a persistent root object is implicitly persistent. "Transitive persistence" is a term coined by the ODMG group and supersedes the original term "persistence by reachability" used by Atkinson in [ATK83].

- **Persistence Independence.** Code should operate unchanged with exactly the same semantics whether it is operating on short-lived or long-lived data. Producing code that is indistinguishable whether it is transient or persistent enables the same software to be reused as either transient applications or in a persistent context.

These three principles are motivated by the goal of maximising application programmer productivity. An environment supporting these three principles hides database-specific logic and allows Java programmers to focus on their applications. However, building software components using a modified Java VM is not an open approach and narrows the possibility to reuse and distribute these components on other VMs.

¹ [ATK96]. Also see http://www.dcs.gla.ac.uk/pjava/.

3-10
3.3 Persistence of CORBA Objects

At the moment CORBA standard implementations and Object Databases coexist on the market and are successfully used in different domains as autonomous tools. CORBA vendors as well as database vendors are working on the integration of both technologies which offers great benefits to both sides. Each tool provides some degree of both distribution and persistence to object systems. RDBMS are often extended by a third-party object-relational mapping tool to provide the same functionality.

CORBA can be complemented by database-specific features necessary for mission-critical applications such as:

- concurrent access to a great amount of persistent objects
- data recoverability
- guarantee of data integrity in the event of failure
- ACID transaction
- better management of server memory

Database products, and especially ODBMS, also profit from the integration with CORBA environments:

- extended heterogeneity for client languages and operating systems
- better distribution: Clients can operate without any knowledge about the logical layout of objects stored in the database and can be essentially lightweight.
- better authorisation mechanisms
- implementation of database views

The next few sections present some technical aspects of the integration of CORBA with database technology.
3.3.1 Non-Integrated vs. Integrated Solutions

Most software products that claim to integrate CORBA with DBMS today do not offer more than either an IDL interface to a persistency layer, i.e. a database middleware component, or simply the ability to invoke other CORBA components in order to use external services. There is no integration that offers true distribution of database objects. However, since most business applications are still request-driven, i.e. they receive a query from a client and return a set of records as a result of the database query, there is very seldom an actual need for database objects to process business logic themselves (offering an own IDL interface) and have a lifetime that exceeds a single client request or transaction (Figure 3-2).

Examples for such database components are Databroker (I-Kinetics) or Delphi DB components (Inprise). Most application server products work this way. All they do is return a recordset to the client as a result of a database request.

Figure 3-2. Using a CORBA interface of a database component.

Future business applications that use Electronic Commerce to complete a whole business process will need more than just a simple, request-driven, OLTP\(^1\)-like application server. Features like cross-transaction caching, management of resources, state, sessions and object

---
\(^1\) Online Transaction Processing. See glossary.
Persistence

Identity, and high lookup-performance for rebind of objects during object faults are needed for an environment where (persistent) business objects are highly integrated with the ORB and actually "live" as transient objects (Figure 3-3).

Figure 3-3. Business Objects integrated with the ORB.

The following section illustrates some technical details of such integration. According to a proposal made by the ODMG the integration framework is called "Object Database Adapter". It co-operates with the object adapter provided by the ORB.

3.3.2 Object Database Adapter (ODA)

3.3.2.1 Skeleton Inheritance vs. Delegation (Tie) Approach

Up to now, various techniques of integration within an ORB have been discussed ([AMI97], [REV96], [SCH98], [VAS94]) but at present two approaches are generally used by the tools existing on the market: direct persistence of CORBA objects (skeleton inheritance) and persistent object wrapping (delegation approach).

Skeleton Inheritance

This approach makes CORBA objects persistent including CORBA runtime-related state data, i.e. the inheritance hierarchy that comes with the implementation object. The persistent
object implementation inherits from an IDL compiler-generated skeleton class which implements the ORB functionality of marshalling and unmarshalling method invocations and parameters. This type of adapter is easy to use and seamlessly integrated into the ORB environment, yet it causes several problems:

- One drawback of this approach is the high amount of overhead data stored together with the business object's state. An example for this approach is IONA's Orbix-ObjectStore-Adapter (OOSA), Version 2.1. Future versions of this adapter are said to solve this problem.

- An even more delicate issue is that of an object's reference count in C++ (not in Java). The CORBA operations duplicate and release update the object's reference count. If the reference count were actually stored in the database, every operation on the object would have to be performed within an update transaction, because duplicate and release appear everywhere.

- ORB implementations keep a per-process table of active objects. A new entry is inserted into this table whenever the constructor of a CORBA object is invoked by the corresponding process. In an ODBMS, however, the constructor of a persistent object is only invoked when the object is added to the database and not when being reactivated and brought into memory by some kind of virtual memory mapping architecture.

- Sometimes, the overhead of such inheritance is too high, or the implementation objects are already placed in a program-specific inheritance hierarchy which does not permit to be extended by the CORBA classes (note that multiple inheritance in C++ can cause a lot of confusion and problems; in Java it is not possible at all). For example, implementing objects using existing legacy code might be impossible if inheritance from some global

---

1 For a more detailed analysis of the OOSA framework see [AM197].
2 The Virtual Memory Mapping Architecture (VMMA) was introduced and patented by Object Design, Inc.
class were required, due to the invasive nature of the inheritance. In some cases delegation can be used to solve this problem. Rather than inheriting from a skeleton class, the implementation can be coded as required for the application, and a wrapper (tie) object will delegate upcalls to that implementation.

### Delegation Approach

Having a wrapper object delegating an invocation to a persistent object realises a loose coupling between the database and the CORBA objects. Unlike the direct persistence approach mentioned above, the CORBA object, which provides the front-end to business logic and state, is different from the persistent object. It stores the reference to the associated persistent object and delegates method invocations to it. This technique strongly resembles the CORBA tie approach. The only difference is that the tie object is designed by a programmer and the implementation object is persistent. Figure 3-4 illustrates a possible implementation of a persistent “Account” object and its CORBA wrapper, using the ORB framework of IONA’s Orbix. The wrapper object `TIE_Account` has to be implemented by the programmer. It bears the responsibility of connecting the database and creating, reading, updating or deleting the persistent object cache, represented by `Account_Impl`. `Account_Impl` not only carries the persistent state of an “Account” object but also implements its methods. `Account` and `CORBA::Object` are part of the ORB framework.

Example IDL:

```idl
interface Account {
    readonly attribute double balance; // persistent value
    double withdraw(in double amount);
    double deposit(in double amount);
};
```

1 This technique is also known as the “Adapter pattern” in [GAM94].
Persistence

Figure 3-4. Persistent object wrapping technique. Syntax similar to IONA’s Orbix.

Persistent object wrapping is a common technique used in most database adapter frameworks. It prevents storing CORBA specific information in the database, increases the adapter’s efficiency and gives the programmer much space to manoeuvre.

Summary

Inheritance is easier to use than the tie mechanism because implementation objects look and behave just like object references. The tie mechanism on the other hand offers the possibility of object-level filtering and delegating invocations to different implementation objects. This is another application of the Object Adapter pattern [GAM94]. Hence using the delegation-based approach the Object Database Adapter is much more flexible.

However, with the programmer implementing the tie object himself, this requires a lot of low level programming. For example a database-specific tie object typically cannot hold a pointer to its persistent object since with some database systems pointers to persistent objects do not remain valid across transaction boundaries. Therefore dangling pointers have to be detected and references to persistent objects have to be restored when requested. Java 2 provides flexible mechanisms to handle references on objects that may or may not be garbage collected by a persistence framework.
3.3.2.2 Architecture

ODA was proposed by the Object Database Management Group (ODMG) in [ODM97] and is compatible with the CORBA specification. Object Database Adapters are complementary to the Basic Object Adapter (BOA) or the future Portable Object Adapter (POA), respectively. They provide the ability to register a subspace of object identifiers with the ORB rather than having to register all database objects (which would cause an unacceptable overhead or would not even be possible with a large number of objects), still allowing direct access to the objects as if they had been individually registered. They use certain hooks in the BOA implementation in order to transparently activate objects and integrate a garbage collection and transaction mechanism. Figure 3-5 outlines the architecture of the object database adapter.

![Figure 3-5. The architecture of the adapter approach.](image-url)

Although it extends the BOA features, a database adapter does not replace it. The same server can host regular transient objects, managed by the BOA, and persistent objects, managed by the BOA in conjunction with the database adapter.
3.3.2.3 **Object Reference Representation**

The client performs a request by having access to an Object Reference for an object and knowing the type of the object and the desired operation to be performed. However, the client and also the application programmer does not know about the internal structure and contents of the reference. Besides IP address, TCP port, object type and some ORB specific information the reference contains an identifier that assures uniqueness within the operating system process the object is "living" in. For transient objects the ORB provides these identifiers. As the reference to a persistent object must contain information identifying the persistent object location in a database (a key, through which the object can be located in the persistent store by the server), the unique identifier must be determined by the database or the ODA, respectively. Thus, persistent object reference representation is crucial for CORBA/DBMS integration.

**Level of uniqueness of OIDs**

Obviously database-maintained references to stored objects seem to be the best candidates to be taken as a basis for the construction of references to wrapper objects. The natural strategy is to set a persistent object's reference, dumped to a string, as an identifier\(^1\) of the related wrapping object. This guarantees the reference uniqueness as well as facilitates the persistent object retrieval.

However, this approach is not sufficient in environments such as the BOCA. Here, a persistent object's reference has to be unique not only for one process or one database, but also amongst several heterogeneous databases and different server processes within a single business system domain (BSD). Thus, database-specific object identifiers have to be extended

\(^1\) The identifier being part of the CORBA object reference is also known as "marker" (IONA Orbix) or "reference data" (Inprise Visibroker).
by type and BSD-specific information provided by the object type manager or domain manager.

**Object Ids for objects in object-oriented databases**

Reverbel [REV96] experimented with three different approaches to integrate ODBMS-persistence of CORBA objects, calling them *Pseudopersistence*, *Smart Pointer-Based Persistence*, and *Virtual Persistence*. The first approach makes CORBA object references persistent, but not transparently storable. The others provide transparent storability of CORBA object references, both in the case of references to local objects and in the case of references to remote objects. The pseudopersistence and smart pointer persistence approaches are not restricted to pure ODBMSs. Through object-relational mapping, e.g., these approaches are applicable to relational DBMSs as well. Virtual persistence, however, applies only to the case of a virtual memory-based ODBMS like ObjectStore. The Business Object Framework introduced in chapter 6 uses the smart pointer-based persistence approach.

**Object Ids for objects in relational databases**

Using OIDs for identifying table rows simplifies the navigation issue in relational databases. Maintaining relationships between objects can be automated because all tables are keyed on the same type of column(s), in this case OIDs. Besides, a special column used as a primary key avoids the problem of the key containing business meaning and therefore potentially being a point of change. Anything that is used as a primary key in one table is virtually guaranteed to be used in other tables as a foreign key and thus causes a lot of problems when a simple change is being introduced (e.g., adding a digit to a customer number). In the relational database world, the OID strategy is referred to as employing *surrogate keys*. 
Performance issues when obtaining OIDs for RDBMS rows

The decision about who will determine new OIDs can greatly affect the runtime efficiency of the application. Using the SQL MAX() function on a single table in an RDBMS or maintaining a single row table which stores the value of an incremental counter both have a serious drawback: The MAX() approach always requires a table lock to ensure uniqueness among concurrent requests of different database clients and the single row table quickly becomes a bottleneck when using it for every OID in the application.

The so-called HIGH/LOW approach presented in [AMB98] provides a technique that avoids these problems. The basic idea is that instead of using a large integer for the OID, requiring the program to go to a single source (and therefore a bottleneck) to obtain the OID, the OID is reorganised into two logical components: a HIGH value that is obtained from a single source and a LOW value that the application assigns itself. The HIGH value is only requested the first time the application needs a new OID. Because HIGH is obtained from a single source it is guaranteed to be unique. At this point the value for LOW is at zero and is incremented every time an OID is needed during the lifetime of the server process.

The advantage of this approach is that there is no longer a table locking or single row bottleneck problem and the object database adapter can assign OIDs by itself without the overhead of requesting a DBMS.

3.3.2.4 Object Activation

Typical CORBA objects require a specific invocation to activate them, before accessing them. They are registered with the ORB individually, a process which involves several ORB calls, which is very slow and consumes memory resources. As the number of Business Objects grows very large, any requirement to independently register each one and activate it would
Persistence

not be an efficient approach and would not use the facilities of a DBMS effectively. Instead, business objects and their corresponding wrapped persistent counterparts should be managed by the object database adapter which activates them transparently without making them visible to the ORB. This is important in practice not only for the transparent model, but also for performance and scalability. This way of activating objects is also used by proxy objects that represent object associations which have to be resolved transparently.

In the future, the Portable Object Adapter (POA) will even provide functionality to construct CORBA object references without creating the corresponding CORBA object. This is an efficient way of returning a large number of object references to the client, e.g. as a result of a query, and complements the transparent activation mechanism. Here, the term “activation” becomes ambiguous: activation of object references is not the same as activation of objects themselves.

Moreover, one has to differentiate between object activation and creation, when persistent objects come into play: A persistent object constructor is invoked only once – when it is added to a database. Once created, an object can subsequently be activated and deactivated. On the other hand object activation will fail if an object has not been previously created and stored.

So, how does the activation mechanism work? When a requested object is not found in the process’s table of active objects, the adapter’s Activator instance is called to activate the appropriate object. The invocation request bears all required information to perform this task, in particular the reference data (object ID as part of the CORBA object reference) needed to identify the database and the persistent object location within this database. When the persistent object is found, the Activator loads it into the server’s memory without

1 Activation of CORBA objects by a programmer-defined mechanism is handled by a “Loader” in Orbix ([ION95] pp. 233) or an “Activator” in Visibroker ([VIS98a] pp. 5-14).
invocation of the object’s constructor and creates a new wrapper object (tie object), connecting both. The wrapper object, being an ordinary CORBA object, is registered by using the requested object’s reference data. Now, the invocation can proceed and return results to the client. The entire activation process should be transparently done by the object database adapter framework.

3.3.2.5 Object Deactivation

Generally the deactivation of CORBA objects seems to be more problematical than their activation. While the program has clearly defined conditions when a CORBA object must be created, the question when this object should be deactivated remains open. Usually, CORBA objects are never deleted until the end of the process they are living in. This is because they are activated on the demand of external clients which may use the objects for an unknown time. CORBA itself does not support distributed garbage collection. Therefore a local garbage collection mechanism is needed, that removes objects from the main memory applying some heuristic method.

Of course, the simplest possible way to accomplish object deactivation is to include an additional method in the object’s interface which allows explicit deactivation of the object. CORBA clients would notify the server when the particular object is no longer needed (O2 has adopted this policy). The implementation is straightforward, but at the same time this method has essential drawbacks: It changes the object’s interface, making the clients aware of the target object’s persistence and transferring the responsibility of managing the server’s memory to clients.

Instead, memory management should be transparent to the client and even to the application programmer. Different possible garbage collection mechanisms should be offered to the application programmer by the adapter framework:
Persistence

- *Client connections* can determine the lifetime of an object in that objects are deactivated as soon as the client, that requested them, terminates the connection with the server. However, this policy is not sufficient, as clients can create an arbitrary amount of server objects while they are connected.

- The *maximum number of active objects* could be determined by the application programmer. The garbage collection mechanism would delete the longest living objects in order to match the given maximum. This approach is easy to realise but does not consider the needs of the clients. Some objects may be more frequently used than others and therefore should be moved to the end of the deactivation queue.

- A more sophisticated approach is to *trigger invocation requests* and delete objects only if they have not been used for a certain time. This *timeout policy* much better meets the actual dynamics of the application.

- There are many other approaches possible. Even user-defined policies could be integrated. The best choice, however, may be to combine different policies to improve the efficiency of deactivation and memory usage.

### 3.3.2.6 Transaction Management

The choice of the transaction boundaries is one of the most problematical issues critical to the performance of CORBA/DBMS integration. Most DBMS require that access to persistent objects be performed within a transaction boundary. Therefore it is not possible for a persistent object to start the transaction which will be used to access itself. Ending transactions is also a delicate task, since at commit or rollback time the persistent object may be flushed out of memory: it cannot access its state afterwards, so it should make temporary copies of any value that it wants to return to the client, which is quite awkward.
Local transactions

In order to cope with these problems and make transaction management transparent, the database adapter should accept responsibility for starting a transaction if necessary before invoking a method on a persistent object, and end the transaction upon the completion of this call. This can easily be implemented by interceptors.

Distributed transactions

The previous approach can cause considerable overhead where many methods of relatively low cost exist in the object. The alternative would be to explicitly manage transactions by an additional service (the Object Transaction Service [COS97]). This would leave the control of transaction boundaries to the client, who additionally has to have stubs for the OMG Object Transaction Service (OTS) IDL interfaces. The database adapter should implicitly choose the appropriate method as requested. An XA-compliant transaction manager like the OTS is necessary in order to provide distributed transactions.

Concurrency control

Because business objects are sharable in a distributed, multi-user, transactional environment, there must be concurrency control to maintain the integrity of the model they represent. A desirable way to improve efficiency of concurrent access is to register different operation styles (e.g. no transaction, read-only or update) with each single method. Again, the interceptor may be used to automatically recognise the appropriate operation style and start a transaction transparently. This can considerably improve the efficiency of concurrent access of multiple clients to a persistent object.

1 [OMG98]
3.3.3 OMG Persistent State Service (PSS)

"It is expected that business object frameworks [...] may use the PSS in the future."

As the successor of the Persistent Object Service (POS)\(^2\) specified in 1994 the Persistent State Service implements a single-level store mechanism. A client has no way to tell if the implementation of an object uses this service. Thus, control over persistence is implementation-specific. PSS provides an internal, datastore-neutral interface to programmers who develop object implementations (Figure 3-6). It deals with the states of a large number of possibly fine-grained CORBA objects.\(^3\)

Persistent objects are supposed to be accessed within a transactional context. PSS implementations provide transactional data access with simple local transactions and transactions managed by the OMG Object Transaction service. Other services that may be integrated with PSS are Concurrency Control, Object by Value, Query and the Portable Object Adapter.

\[
\text{Figure 3-6. The PSS specification does not deal with the external interface of a CORBA server, but with an internal interface between the CORBA world and the datastore.}
\]

---

\(^1\) [BUS98b] Page 70.

\(^2\) The POS has not proved viable and has been deprecated due to major deficiencies in its specification. Problems with the POS specification have been discussed in [KUS97], [KLE96a], [KLE96b] and [ADA95].

\(^3\) This requirement is addressed by the Portable Object Adapter specified in CORBA 2.2.
Persistence

The current revised submission deadline is August 2, 1999. Table 3-2 shows the key players in the specification process that have issued revised submissions in the past or support the submitted proposals. The subsequent voting process can take 10 to 14 weeks. Thus, an adoption of the Persistent State Service specification is to be expected in the end of 1999. Among the participants voting for the final submissions are Boeing, British Telecom, GemStone Systems, Hewlett Packard, Lucent, Microsoft, Novell and Xerox.

Table 3-2. Key players in the PSS 2.0 specification process (supporters in brackets).

<table>
<thead>
<tr>
<th>ORB vendors</th>
<th>DBMS vendors</th>
<th>Middleware vendors</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>IONA Technologies PLC</td>
<td>Oracle Corporation</td>
<td>Persistence Software</td>
<td>Fujitsu Ltd.</td>
</tr>
<tr>
<td>Inprise</td>
<td>Objectivity</td>
<td>(Secant Technologies)</td>
<td>TIBCO Software</td>
</tr>
<tr>
<td>Sun Microsystems</td>
<td>(Ardent)</td>
<td></td>
<td>(Rogue Wave)</td>
</tr>
<tr>
<td>(Expersoft)</td>
<td>(Object Design)</td>
<td></td>
<td>(Hewlett Packard)</td>
</tr>
<tr>
<td>(IBM)</td>
<td>(Poet Software)</td>
<td></td>
<td>(Inline Software)</td>
</tr>
<tr>
<td></td>
<td>(Versant)</td>
<td></td>
<td>(Novell)</td>
</tr>
<tr>
<td></td>
<td>(IBM)</td>
<td></td>
<td>(Rösch Consulting)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(Windward Solutions)</td>
</tr>
</tbody>
</table>

Table 3-2. Key players in the PSS 2.0 specification process (supporters in brackets).

1 Relevant documents are [PSS99a] and [PSS99b].
3.4 Design of a Persistency Layer

From the application developer's perspective there is a need for a separate persistency layer (also called a persistency framework) that manages the persistence of all the business objects used in the application. A separate persistency layer in the form of a class library (e.g. Rogue Wave dbTools++), a component (e.g. I-Kinetic's Databroker), or a development environment (e.g. Secant Persistent Object Service) can save 30% of the whole development process, since database access is often a relevant part of the application logic. A persistency layer product should have forward and reverse engineering capabilities and should utilise standard notations like the Unified Modelling Language (UML)\(^1\) and the CORBA Interface Definition Language (IDL)\(^2\) for the description of the object model. It should provide a transparent mapping between database tables (relations) and programming language objects and should add features like cross-transaction caching and connection pooling.

The following list outlines the technical benefits of a persistency layer:

- A separate persistency layer enables a *loose coupling* between the database and the objects themselves. Only a loose coupling allows the flexibility needed for changes or enhancements of the object model, data model, or even the underlying database management system.

- *Database connection pooling* improves performance by multiplexing connections to databases rather than opening a database connection for each request. This saves system resources and improves scalability. Minimising the number of connections to the database server ensures efficient communication. Connections are recycled for subsequent reconnections to the same database server, eliminating the time-intensive overhead of releasing and

\(^1\) [UML97]  
\(^2\) [OMG98]
creating new connections. The JDBC 2.0 standard contains the connection pooling option and first products supporting connection pooling on the JDBC driver layer were released in the first quarter of 1999.

- The persistency layer represents an object-oriented DBMS, even if the underlying database is relational. This not only simplifies the database API but also makes changes of the database schema easier. Besides, this enables objects to transparently be split up in parts stored in different databases. Such a feature is particularly helpful when integrating legacy data that is to be enhanced by additional attributes and relationships.

- If implemented as a separate component (i.e. a separate executable), the persistency framework provides programming language independence and it gets around multithreading problems for some applications in case the database doesn’t permit more than a single thread at a time.

- Many products come with adapters (plug-ins) that let applications connect to particular data sources, such as specific databases, CICS, 3270 (screen scraping), MOM products, SAP, and DCOM bridges. By using adapters to wrap legacy applications, existing functionality can be re-used and integrated into new applications.

### 3.4.1 Architecture

Figure 3-7 shows the principle components of a possible architecture.\(^1\) The application uses service objects, like factory and query objects, as well as the persistent objects. In case the framework has generated type-safe classes which reflect the domain classes, the application could, for instance, receive persistent Objects of type `Account` querying the `AccountQuery` interface or creating new objects using the `AccountFactory` interface (see Figure 3-8). Other,

---

\(^1\) The suggested architecture complies with [SIG99]. See also [AMB98a] and [DOB99].
more generic frameworks may use generic classes from which the application programmer has to derive his own classes.

The object manager manages the service objects and a cache containing all persistent objects. It also provides generic methods that can be used to implement type-specific service objects. As shown in Figure 3-7 the object manager communicates with a schema mapper component. Each persistent business object class has its own schema mapper which contains the knowledge about how to construct a business object out of one or more data objects. Data objects represent the data structures in the data stores. A data object may represent a single row in a relational database or a segment in an IMS\(^2\) database. Data objects and associated service

---

1 From [SIG99].

2 Information Management System. A hierarchical high performance database for IBM mainframes which is implemented on top of VSAM (an IBM disk file storage scheme).
objects (e.g. factories) form a data service that abstracts from different data stores and provides the schema mapper with a uniform view on them. The wrapper component has already been introduced above as an adapter.

Type-safe classes are easy to work with, but more dependent on the schema mapping.

Figure 3-8. Type-safety vs. generic classes.

### 3.4.2 Persistent Objects and Locking

The persistence of objects should be handled implicitly using transaction boundaries as a trigger for creation, activation, deactivation or update. This way, modified states do not have to be explicitly made persistent using a store command, but instead are committed during transaction termination. Thus, all modifications on persistent state have to be done within a transaction.
In addition, persistence and transactions in a distributed environment have to use locking mechanisms, since concurrent access to a single object is possible. The two most common locking mechanisms are optimistic and pessimistic locking.

- With **optimistic locking** the state of an object is logged at the beginning of a transaction. Before committing the transaction this logged state is compared with the current state in the database. If both are equivalent (i.e. no modifications have been made to the database in the meantime) then the modification will be accepted, otherwise the operation is rejected and the transaction is rolled back. Timestamps that have been added to all tables can also serve as a comparator. This approach allows many clients to work with an object simultaneously, and is therefore best for online processing.

- During **pessimistic locking**, objects are only allowed to be read concurrently but not to be modified concurrently. Before updating the database with the modifications applied to the object the lock is expanded to an exclusive lock. Pessimistic locking is ideal for batch jobs that need to ensure consistency in the data that they write.

The advantage of optimistic locking is that the objects involved in the transaction are locked only for a short time during transaction commit and are otherwise free for concurrent write access. On the other side optimistic locking will cause some transactions to be rolled back although there was no error in the application logic. Furthermore, optimistic locking can cause a considerable overhead of determining whether or not the record has been updated by someone else when you go to save it.

Depending on the design of the persistency framework or on requirements determined by the application logic the efficiency of these locking mechanisms can be increased by introducing different **access modes** (read-only, read/write) and different **isolation levels** (read uncommitted, read committed, serialisable).
3.4.3 Object Manager and Cache

The object manager's main task is managing a cache containing all active persistent objects. The objects are identified by a unique object identifier (OID) or, depending on the schema mapping, by a combination of columns representing the primary key in a relational database. The object manager assures that there are never multiple instances of a persistent object in the cache in order to ensure consistency. The cache (or registry) is implemented using hash tables or binary trees that are indexed by the object identifier (see Figure 3-9).

![Figure 3-9. Making objects unique using a registry.](image)

To improve the persistence framework's efficiency and flexibility the object manager should apply various caching strategies including a read-ahead strategy where more than only one level of objects is read from the database but rather the associated objects are also loaded into the cache at the same time. If the object model has many relationships, the traversal of the object model can cause a large number of expensive database queries. A read-ahead strategy lets the application minimise the number of database queries by retrieving large object composition trees co-instantaneously. Reading objects ahead often results in too much data. Therefore, it is desirable to keep the data in binary format until it is requested, and instantiate the objects and fill them with the cached state on demand only.
In a three-tier architecture multiple clients access an object server concurrently. Thus, the server must be able to process concurrent transactions. Transactions that modify persistent state are not allowed to work on the *shared persistent object cache* but instead manage their own *transactional cache*. When starting a transaction the participating objects are copied from the shared to the transactional cache and can be modified there without influencing the views on the objects taken by other clients.¹

The other main task of the object manager is to provide some generic methods by means of which service objects (or another client) realise database access:

- **Resolve a persistent object** from the database by referencing an OID. If the object already exists in the cache a reference on this object is returned, otherwise the schema mapper first loads the object from the data store.

- **Resolve a collection of persistent objects** using a query interface. The query may be expressed by an SQL or OQL statement.

- **Navigation between persistent objects** either directly as a result of a method invocation that takes the root object and the association definition as parameters, or indirectly using some kind of smart pointer technique (see section 3.4.5).

- **Update the persistent state** during transaction commit. This can be done by co-operating with a transaction manager.

- **Create new persistent objects**, register them in the cache and write them to the database during transaction commit.

- **Deliver meta information** about the managed object type.

In contrast to the suggested architecture shown in Figure 3-7 it may also be possible to have one object manager and schema mapper for each single type of persistent objects. In this case

¹ This concept is illustrated in chapter 6 in more detail.
the object manager may also have type-specific method signatures to avoid additional
downcasting from a (generic) base type.

### 3.4.4 Schema Mapping

The schema mapping components contain the knowledge about the correlation between
persistent objects and *data objects*. Data objects reflect data structures in the data stores. One
persistent object may be related to many data objects. The schema mapper knows which
attribute in a data object correlates with which attribute in the persistent object. Each data
object knows how to get its attributes from the respective database table (in case of an
underlying RDBMS). During initialisation the object manager assigns a specific schema
mapper to a persistent object type. By simply changing the assignment the schema mapping
can be changed without touching the application source code.

By introducing these three steps in the mapping process (data model, mapping model,
persistent object model) the mapping is very flexible and a lot of the source code needed for
the mapping can be easily generated by a development tool. Besides, the complexity of the
mapping process is reduced by splitting it up into several modules. Reverse engineering
becomes easier because in the beginning the object model is quite different to the data model
and there are several steps needed to implement the mapping. During forward engineering
the flexibility can be used to hide optimisations like de-normalisation of relational database
tables from the persistent objects and instead place optimisations inside the data objects.
Thus, the interfaces of the persistent objects are not influenced by database-specific changes.

In component-based architectures, generic data objects can be externalised and encapsulated
as separate components, serving as plug-ins for the persistency framework. This is also one of
the ideas of the OMG Persistent State Service: to provide a uniform interface to different
Persistence
types of data stores. Due to reuse by many clients these data store components can increase scalability if they use multiple threads and database connection pooling techniques.

The schema mapper provides methods to save, to load and to delete objects or to load collections of objects. It may also take the responsibility to recursively load and update associations between objects and associated objects themselves, and to assure referential integrity if associations change. In order to load a persistent object the schema mapper first determines the relevant data services and creates the necessary data objects using a data object factory. On behalf of appropriate key values the data objects now fill themselves with data from the associated data store. Finally, the schema mapper creates the persistent object and transfers the persistent data from the attributes of the data objects to the attributes of the persistent object.

For more information about mapping objects to relations see [DOB99], [COL96], [MÓS96].

3.4.5 Dealing with Relationships

Describing relationships between object classes is an essential element of modelling and design. UML and other object modelling methodologies provide ways of defining the semantics of relationships in terms of their cardinality and navigability. Using object modelling tools for generating persistent object mapping code relieve the application developer from the fairly complex task to manually code the database key maintenance. The implementation details that assure referential integrity can be hidden behind accessor methods (get and set methods; also called mutators). Accessors for one-to-one associations return the member object of the association. An accessor for one-to-many associations returns a collection of member objects or an iterator for a collection, respectively. Mutators (set methods, for example) and collection add/remove methods should automatically invoke
the appropriate referential integrity maintenance behaviour, such as updating the inverse association.

**Smart Pointer Persistence**

Transparendly managing relationships can be accomplished by using *smart pointer* (proxy) objects. In opposite to the read-ahead caching strategy smart pointer objects offer a way to transparently load persistent objects from the database only if they are requested.\(^1\)

1. Instantiate object ‘A’ from the database.
2. Create smart reference (proxy) to object ‘X’.
3. Resolving ‘X’ causes object fault.
4. Look up registry (cache).
4a. If ‘X’ is not yet instantiated, load its state from the database and create it.
4b. If ‘X’ exists, establish connection to proxy.

*Figure 3-10. Proxy objects implementing a smart pointer mechanism.*

When a persistent object is brought into memory and is registered with the object manager’s cache, its associated objects are represented by empty proxies. As soon as the application tries to de-reference the association (through a `get` method) the proxy instantiates the associated

---

\(^1\) ODMG 2.0 specifies a standard API for such a smart reference. See [ODM97].
Persistence

object and initialises it with data from the database. The same procedure happens in case the associated object was instantiated but has already been garbage collected and consequently has left a dangling reference. Dangling references are detected by an object faulting mechanism and resolved transparently.

Java Specialities

Java 2 offers a convenient way to implementing smart pointer classes by means of weak or soft references. *Weak references* point to a referenced object without preventing the garbage collection mechanism to free it. If the object is freed the weak reference returns null. *Soft references* work similar except for the fact that the referenced objects are not garbage collected as long as there is enough memory available. Soft references can be used to implement a cache that relies only on the Java default garbage collection mechanism.

Association vs. Aggregation

Beside inheritance (which this thesis does not deal with) there are primarily two types of relationships: association and aggregation. An *association* represents a non-committal relationship between two arbitrary classes. The association may be one-to-one or one-to-many, and it may be bi-directional or one-way. In opposite to associations *aggregations* represent an existential dependence between two objects. If the aggregate\(^1\) is deleted, its aggregated objects are deleted as well – except if the aggregation is not exclusive, i. e. it is shared, and the aggregated object has more than one aggregate. *Composition* is a form of aggregation with strong ownership and coincident lifetime of part with the whole. The aggregated object is unshared and it cannot be replaced. It lives and dies together with the aggregate.

\(^1\) To make terms clear: the aggregate owns one or more aggregated objects. See [UML97] for details.
The *transitive persistence* scheme\(^1\) can be implemented by aggregation relationships. If the aggregate is made persistent, the whole transitive closure is made persistent as well, that is, all objects that can be referenced within the object graph.

---

\(^1\) Also known as *persistence by reachability* in [ATK83].
4 Distributed Transactions

My transaction hopes are pinned on the impact of the distribution of processing - when it is realised that data is not everything and that process is just as important.

Jim Gray and Andreas Reuter

Transaction Processing Concepts and Techniques

The concept of transactions is an important programming paradigm for simplifying the construction of reliable applications, especially those that require concurrent access to shared data. Not only data but also programming logic requires to be reliable and consistent. Consequently transactions are the key to constructing reliable distributed applications.

The transactions performed as part of a distributed application are much more complex than the types of transaction processing we’ve seen before. Batch transaction processing based on magnetic storage (tape and disc) and online transaction processing (OLTP) based on electronic storage and computer networks were largely responsible for growth in the computer industry. Transaction processing was and still is the primary force that drives business applications. Heading towards a fully automated business process based on Electronic Commerce ("e-commerce") transaction processing needs to be as instant as OLTP while taking a very long time to complete and including several distributed applications and data sources. This chapter introduces the technology of distributed transaction processing on which future business application will be based.

1 [GRA93]
4.1 Introduction

4.1.1 ACID Properties

In order to enable reliable and fault-tolerant computing transaction processing systems support the following four properties: atomicity, consistency, isolation, and durability. These properties have emerged as the unifying concepts for distributed computation.

- **Atomicity.** A transaction's changes to the state are atomic: either all happen or none happen. These changes include database changes, messages and actions on transducers.

- **Consistency.** A transaction is a correct transformation of state. The actions taken as a group do not violate any of the integrity constraints associated with its state. This requires that the transaction be a correct program.

- **Isolation.** Even though transactions execute concurrently, it appears to each transaction, T, that others executed either before T or after T, but not both.

- **Durability.** Once a transaction completes successfully (commits), its changes to the state survive failures.

4.1.2 Standards

4.1.2.1 IBM LU6.2

As part of IBM's networking standard SNA, LU (logical unit) 6.2 defines how a client can invoke a remote transactional server and establish a session with it. LU6.2 specifies the formats and protocols to coordinate the atomic commitment of all members of the transaction, i.e., several transactional servers.
4.1.2.2 ISO OSI-TP

Defining a de jure standard for transaction processing, the International Standards Organisation (ISO) to some extent redefined LU6.2 and repaired some of its minor flaws. The resulting transaction processing standard is part of the network architecture called Open Systems Interconnection (OSI) and specifies how to deal with transaction identifiers, commit propagation and recovery.

4.1.2.3 X/Open DTP

OSI-TP specifies formats and protocols that enable interoperation between different transaction processing systems, but it does not define an API that would provide a way to write portable transaction processing applications or servers. This problem has been solved by a second standards body, X/Open, defining the X/Open Distributed Transaction Processing Reference Model (X/Open DTP).

The X/Open DTP model describes how an application can use transaction processing monitors (TP monitors) like Tuxedo to update databases such as Oracle under transaction control. X/Open DTP is supported by all of the leading UNIX TP monitors, including BEA's Tuxedo, NCR's TopEnd, IBM's Encina, and SNI's OpenUTM. It is also supported by all of the leading UNIX databases, including Oracle, Informix, IBM DB2, and Sybase.

Figure 4-1 outlines the key aspects of the DTP architecture. With the DTP model, a transaction manager allows a transaction to span more than one application, process, or machine by keeping track of the resources involved in the transaction. It assigns a globally unique transaction identifier (TID), which tags all transactional invocations on behalf of that transaction. The transaction manager communicates with each resource through a resource manager. When the client that began the transaction, ends it by issuing a commit or rollback
instruction, the transaction manager co-ordinates a two-phase transaction completion. In the first phase, each resource votes to either commit or rollback. In the second phase, the resource managers all commit or all roll back the transaction.

The TX interface is supported by the transaction manager. This interface is used by a client to begin, commit or rollback a distributed transaction (for example, `tx_begin()`, `tx_commit()`).

The XA interface is supported by the resource manager. This interface is called by the transaction manager to indicate that a distributed transaction has begun (for example, `xa_start()`), and also during a two-phase commit (for example, `xa_prepare()`, `xa_commit()`).

---

1 The Two-Phase Commit Protocol is explained in more detail in section 4.2.1.
4.1.3 Transaction Models

Flat Transactions

Flat transactions represent the simplest type of transaction. The structure of the execution of the single actions inside the transaction is of no significance. There is no way of receiving an intermediate state of execution. Everything inside the transaction brackets is at the same level.

The major restriction of flat transactions is that there is no way of either committing or aborting parts of the transaction. Therefore more complex applications require more sophisticated models including savepoints, chained or nested transactions. Savepoints extend the flat transaction model with the option of stepping back to an earlier state inside the same transaction. Chained and nested transactions are explained in the following sections.

Chained Transactions

The idea of chained transactions is to keep the database context and resources even after committing a part of the transaction. Unlike the savepoint concept the chaining step irrevocably completes a transaction and the next transaction is started within the same context. An important advantage in the performance of chained transactions is that each commit allows the application to free locks that it does not need anymore. In addition, it is possible to keep control over resources that should not be allocated to other users (transactions).

---

1 According to [GRA93] savepoints are “points within the transaction execution to which the application can later roll back. [...] A savepoint is established when the invocation starts. If anything goes wrong, the service can roll back to that savepoint and then return a diagnostic. [...] This provides simple server error semantics without aborting the entire transaction.” This is not to be mixed up with checkpoints which are used to quickly reestablish the current state after the unlikely event of a resource manager or node failure. “A checkpoint is a relatively recent persistent copy of the state that can be used as a basis for restart.”
Nested Transactions

Nested transactions are a generalisation of savepoints. They are organised as a hierarchy rather than a sequence of actions. Subtransactions can be completed or rolled back individually; their commit will not take effect, though, unless the parent transaction commits. Rollback of a transaction anywhere in the tree causes all included lower-level transactions to be rolled back as well. Hence a sub-transaction has only the A, C, and I properties, but it is not durable.

Distributed Transactions

Distributed transactions are a restricted type of nested transactions and are used, for example, in distributed databases. Unlike nested transactions, if a subtransaction issues a commit or rollback, this signals the commit or rollback of the entire transaction, which forces all other subtransactions to commit or rollback, respectively.

Multi-Level Transactions

Multi-level transactions are a more liberal version of nested transactions. They allow the commit of a subtransaction without the possibility of rolling back the changes in case of a rollback of the parent transaction. Instead, the rollback is “simulated” by a compensating transaction which can semantically reverse what the original subtransaction has done. Since by definition an abort must not fail, the compensating transaction must commit; it has not the option to abort!

The advantage of this mechanism is that the results of subtransactions can be pre-committed and hence are made public without losing the atomicity property for the root transaction.
Open Nested Transactions

Open nested transactions do not ensure atomicity and consistency. Subtransactions can abort or commit independently of the final outcome of the parent transaction. Thus an open nested transaction groups other top-level transactions without exercising any further control over them.

Long-Lived Transactions

The transaction models mentioned before are not convenient where the computation done during a transaction takes a long time and would therefore lock resources inefficiently. Assume that at the end of a month a bank has to update a million accounts by crediting or debiting the accumulated interest. By making the whole update a flat transaction would be very expensive in case a failure occurs at the end of the transaction. All the work done up to the point of failure would have to be rolled back and repeated again. Neither savepoints nor nested transactions would help, because in either case the atomicity property is still maintained for the entire transaction.

Using a chained transaction improves the situation considerably, because in case of a failure only the last transaction gets rolled back. On the other side chained transactions do not guarantee atomicity for the overall computation. Furthermore the definition of the chained transaction model does not include state information about the chain as a whole. At restart after a system crash the system has no way to determine in which state the restart should begin, i.e., at which position in the chain. Hence the program needs to maintain some context information in order to be able to resynchronise with the database. This transaction processing context has to be stored persistently so that it can be used to determine the state at which the system should be recovered after a crash. A transaction processing system could store the
Transactions

context data as a database record, or the transaction manager could store context data internally.

The requirements for long-lived transactions can be summarised as follows:

(1) It must be possible to split up bulk transactions in order to minimise the amount of work lost after a system crash.

(2) The application system must be allowed to shut down and recover a transaction without having to commit or roll back the results.

(3) The sequence of transactions belonging to a long-lived transaction must be controlled by the system in order to proceed along the prespecified path or remove from the system what has been done prior to a crash or shutdown. This is accomplished by maintaining context information.

Summary

Currently the most relevant transaction models are flat and distributed transactions. In some systems one can find the savepoint or chaining concept as well as nested transactions. Nested transactions can insulate a global transaction from partial failure of some of its constituent operations and are particularly useful when combined with an object-oriented model and distributed, concurrent systems.

The OMG Object Transaction Service (OTS) supports flat transactions and nested transactions. Since the OTS dominates transaction processing in the CORBA world, these two transaction models will be the relevant models in distributed object computing.
4.2 Transactions in Multi-Tier Environments

Depending on the needs of the application, performance and reliability aspects, there are many different approaches possible for managing transactions in a distributed environment. This section outlines the approaches, and addresses their strengths and weaknesses.

4.2.1 Simple vs. Distributed Transactions

The most simple scenario consists of a non-transactional client communicating with an application that uses a single database. Figure 4-2 shows that although the client invokes a function on an object in the server, only the server and the database need to be involved in the transaction. From the client’s point of view, the transaction is completely hidden.

This approach enables very thin clients because the clients don’t need any transaction functionality. On the other side this approach causes serious performance problems in case the IDL interface to the server object offers fine grained access control. The IDL interface might be something like:

```c
interface message
{
    void get_text(out octet_seq data);
    void get_sender(out string sender);
    void get_recipient(out string recip);
};
```
For each of these operations one possible implementation is to start a transaction, fill the return value, end the transaction, and return. This can be very inefficient, assuming that the cost of starting a transaction is very expensive. One solution to this problem is to make the IDL interface less fine grained in order to minimise the amount of transaction demarcation calls (start, commit). Usually it would be a better choice to have the client defining start and commit of the transaction.

When applications need to manage transactions across multiple, distributed data resources, transaction demarcation (the beginning and end of a transaction) inevitably becomes the responsibility of the client. Also, an independent transaction service is needed to co-ordinate all the components involved in the transaction by means of the two-phase commitment protocol.

![Figure 4-3. A Distributed Transaction.](image)

**Two-Phase Commit (2PC) Protocol**

Since the 1980s, two-phase commit technology has been used to automatically control and monitor commit and rollback activities for transactions in a distributed database system. Two-phase commit technology is used when data updates need to occur simultaneously at
multiple databases within a distributed system, thus ensuring *atomicity* of the overall transaction. By means of synchronised locking of all pieces of a transaction, data integrity can be maintained.

Two-phase commit has two distinct processes that are accomplished in less than a fraction of a second:

1. The *Prepare Phase*, where the global co-ordinator (transaction manager) requests that all participants (XA resources; see Figure 4-1) will promise to commit or rollback the transaction. Either a separate transaction service or one DBMS could serve as the global co-ordinator.

2. The *Commit Phase*, where all participants respond to the co-ordinator that they are prepared, then the co-ordinator asks all nodes to commit the transaction. If any participant cannot prepare or there is a system component failure, the co-ordinator asks all databases to roll back the transaction.

The transaction co-ordinator as well as the resources log information about ongoing transactions, so that in case of a machine, network, or software failure during the two-phase commit process the protocol will automatically and transparently recover resources and finish the transaction.

### 4.2.2 TP Monitors

Transaction Processing Monitors, or *TP monitors*, fulfil a number of tasks, some of which are very much like operating system tasks. They integrate other system components like database systems, runtime systems and presentation services and manage resources like processes, memory and files. Application server processes are shared among the clients and mapped to incoming requests (*scheduling*). This provides an efficient way to balance the workload and deal with a very large number of clients (*load balancing*). Another important task of a TP monitor is
Transactions

authentication and authorisation. Each individual request is checked against many parameters including user, time and program. After a crash, the TP monitor is responsible for bringing up the transaction-processing environment. In particular it has to ensure that all resource managers are restored to their correct states according to the ACID principle. A resource manager is a subsystem that takes part in a transaction. It is recoverable and commits or rolls back its part of the transaction when it is told to do so by the transaction manager.

Several vendors offer a TP monitor approach to handling transactions in a distributed environment. In this fixed architecture, a complex monolithic application handles transaction co-ordination using a built-in Transaction Manager component (Figure 4-4).

![Figure 4-4. TP monitor approach.](image)

The main tasks of TP monitors are very similar to that of Object Request Brokers. In fact, TP monitors started to “morph” into the distributed object infrastructure, introducing object-oriented Transaction Services. Examples are OrbixOTS (IONA’s Object Request Broker (ORB) and Transarc’s¹ Encina TP monitor) and BEA Systems’ Iceberg (BEA’s ObjectBroker and Tuxedo TP monitor).

¹ Transarc is a wholly owned subsidiary of International Business Machines Corp. (IBM).
The advantages to the TP monitor approach are:

- Proven technology with a heritage and an installed base.
- Functionally rich load balancing and monitoring capabilities.

The drawbacks of the TP monitor approach are:

- Not a distributed-object model.
- Proprietary solution suited to static, monolithic environments.
- No Java support on server side.

Examples for TP monitors are BEA's Tuxedo, NCR's TopEnd, IBM's Encina, and SNI's OpenUTM.

4.2.3 ORB Interfacing to a TP Monitor

In an effort to rapidly bridge the gap between legacy systems and CORBA applications, some vendors have come up with a solution comprised of an ORB plus a TP Monitor. In this hybrid model, an ORB interface to a monolithic TP Monitor is provided. On the one side transaction management is then provided by a proofed technology that offers high scalability and reliability. On the other side, this solution is neither a fully distributed nor a fully object-oriented approach to transactions. Since the transaction service is not integrated with the ORB, but uses proprietary calls to the TP Monitor instead of IIOP, the transaction service must reside on every node that supports transactions (see Figure 4-5).

A further drawback is the lack of Java support for the server, making it impossible to integrate an application server based on a traditional TP Monitor with the upcoming Enterprise Java Beans component standard. Thus, the “ORB Interfacing to a TP Monitor” approach can only be one step on the way to a fully distributed and component based application system.

1 An example of this approach is Iona's Orbix/OTM.
A further drawback is the lack of Java support for the server, making it impossible to integrate an application server based on a traditional TP Monitor with the upcoming Enterprise Java Beans component standard. Thus, the “ORB Interfacing to a TP Monitor” approach can only be one step on the way to a fully distributed and component based application system.

4.2.4 Integrated CORBA Solution

In an integrated CORBA solution, transaction management, load balancing, multithreading, database connectivity and so on are functionalities represented by separate collaborating components. These different tasks are integrated into and partly provided by the basic infrastructure provided by an Object Request Broker (ORB). The different components are exchangeable (in theory) and based on an open standard, they can be implemented with any programming language and run on any operating system, hence providing a fully distributed solution.

---

1 An example of this approach is Inprise’s Integrated Transaction Service (ITS).
4.2.5 Transactions with Business Components

OMG Business Components (BOC) or Enterprise Java Beans (EJB) are designed for distributed transactions using the OMG Object Transaction Service (OTS) or the Java Transaction Service (JTS)\(^1\), respectively. A component execution system must be able to integrate updates to multiple, heterogeneous data sources within the two-phase commit protocol. Additionally, it automatically scales to a large number of clients, thus offering TP Monitor functionality.

Within the context of a transaction, a message may go from one component execution system to another with all affected objects being recognised as part of the same transaction so that they are not blocked. Commit and rollback operations must be communicated to all objects affected by a transaction. This is done implicitly. This means that the transactional context does not appear in the argument lists of messages, but is passed automatically by the environment. This allows component execution systems to hide the details of transactional mechanisms from the business developer in order to provide a simplified abstraction.

\(^1\)The JTS is actually the Java binding of the CORBA Transaction Service.
4.2.6 Concurrency Control

There is no problem of ensuring that individual, isolated transactions are correct in themselves. However, even if all transactions are individually correct, it is still possible in a shared (multi-threaded) system for transactions that execute concurrently to interfere with one another in such a way as to produce an overall result that is not correct. Figure 4-7 shows a realistic example. Two transactions work on the same object concurrently, but one is lost, because the other overwrites older results.

<table>
<thead>
<tr>
<th>Transaction A</th>
<th>Transaction B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIND object X</td>
<td>FIND object X</td>
</tr>
<tr>
<td>create working copy X'</td>
<td>create working copy X''</td>
</tr>
<tr>
<td>of object X</td>
<td>of object X</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>UPDATE object X</td>
<td>UPDATE object X</td>
</tr>
<tr>
<td>with working copy X'</td>
<td>with working copy X''</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>L</td>
<td>L</td>
</tr>
</tbody>
</table>

Figure 4-7. The Problem of Interference. A's Update is lost at Time t4.

It is clear that in a multiuser environment some sort of concurrency control mechanism is needed in order to avoid such problems. The essential problem is that A and B are both updating the object X on the basis of its initial state – that is, neither one is seeing the output of the other. To prevent this situation there are basically three possible concurrency control mechanisms to apply:

a) It could prevent B’s FIND at time t2, because A has already access to X and may therefore be going to update it. If A updates the object X, then B should be forced to see the updated state.
b) It could prevent A’s update at time t3, on the grounds that B already has access to the object X and has already seen the state values before the update (and therefore can hardly be forced to see the updated state instead).

c) It could prevent B’s update at time t4, because A has already updated the state of X and therefore B’s update is based on a now obsolete state value.

Cases (a) and (b) above can be handled by locking the object’s state. With exclusive locking, multiple transactions on the same object are serialised, so that the different transactions can not interfere each other. This simple solution, often called pessimistic transaction control, works well on centralised systems. For distributed systems, locking always bears the pitfall of a deadlock and of slowing down the system.

In distributed systems, an optimistic transaction control may be a better solution. With optimistic transaction control, conflict situations are determined when the transaction has finished. This applies to case (c) in the example above. Changes to the persistent object’s state are only made persistent at transaction commit and even then only if no other instance has committed any changes to the object’s state in the meantime.

With both, optimistic and pessimistic transaction control Readers do not conflict with Readers (Read-Lock on objects) but Writers conflict with both Readers and Writers (Write-Lock). The ODMG proposes the conventional lock- based pessimistic approach as its default policy [ODM97]. More detailed information about concurrency issues are to be found in [DAT83].
4.3 OMG Object Transaction Service (OTS)

The Transaction Service supports the concept of transaction according to the ACID characteristics. It defines interfaces that allow multiple, distributed objects to co-operate to provide atomicity. These interfaces enable the objects to either commit all changes together or to rollback all changes together, even in the presence of failure. The value of a separate transaction service is that it allows transactions to include multiple, separately defined, ACID objects. Furthermore, these transactions can even include objects and resources from the non-object world.

4.3.1 Introduction

The Object Transaction Service is implemented in three layers. The top layer is a programming API that enables applications to demarcate (start, stop, and abort) transactions. The middle layer consists of multiple transactional and recoverable objects that participate in a single transaction. The bottom layer manages the logging and recovery services that enable commit and rollback operations within one or more persistent resource managers. A transaction co-ordinator uses a two-phase commit protocol to commit the actual datasources.

A transaction can involve multiple objects performing multiple requests. The Transaction Service synchronises the partaking elements of the distributed client/server application. The scope of a transaction is defined by a transaction context that is shared by the participating objects. Usually this context is established by a client that begins the transaction (by issuing a request to an object defined by the Transaction Service). The client's thread is associated with the transaction context and each request invoked by the client is implicitly associated with this transaction. There is no need for propagating the context as an explicit parameter in a request. An explicit transaction context propagation (with the context object being part of the
parameter list of invoked methods) is also possible, but it requires the transactional methods
to be extended by the context parameter and thus distorts the design of the object interfaces.

A transactional object does not necessarily require that all requests be performed inside the
scope of a transaction. It is up to each object to determine its behaviour when invoked
outside the scope of a transaction; an object that requires a transaction context can raise a
standard exception.

4.3.2 Definitions

Applications supported by the Transaction Service consist of the following entities:

- Transactional Client (TC)
- Transactional Object (TO)
- Recoverable Objects and Resource Objects
- Transactional Server
- Recoverable Server

Figure 4-8 shows a simple client/server application structure, which includes these basic
elements.

Transactional Client

A transactional client is an arbitrary program that can invoke operations of many
transactional objects during a single transaction. The program that begins a transaction is
called the transaction originator.

Transactional Object

The behaviour of a transactional object is affected by being invoked within the scope of a
transaction. A transactional object typically contains persistent data that can be modified by
requests. Not all requests have to have transactional behaviour, even when issued in the
Transactions

scope of a transaction. If an object does not support transactional behaviour for a request, then the changes produced by this request might not survive a failure and the changes will not be undone if the associated transaction is rolled back.

Figure 4-8. The basic elements involved in a transactional application.

Recoverable Objects and Resource Objects

An object whose data is affected by committing or rolling back a transaction is called a recoverable object. A recoverable object is by definition a transactional object. However, an object can be transactional but not recoverable by implementing its state using some other (recoverable) object. A client is concerned only that an object is transactional; a client cannot tell whether a transactional object is or is not a recoverable object.

The recoverable object registers a Resource object with the Transaction Service, which participates in the commit protocol.

1 [OMG98a] Figure 10-1.
Transactions

Transactional Server

The objects in a transactional server are affected by the transaction, but they have no recoverable states of their own. Transactional changes are implemented by other recoverable objects. A transactional server does not participate in the completion of the transaction, but it can force the transaction to roll back.

Recoverable Server

The objects in a recoverable server are recoverable. They register Resource objects with the Transaction Service. The resources implement the commit protocol.

4.3.3 Service Architecture and Functionality

The design of the Object Transaction Service is based on the X/Open reference model, with two related improvements:

- The procedural XA and TX interfaces have been replaced with a set of CORBA interfaces defined in IDL.
- All inter-component communication is mandated to be via CORBA method calls on instances of these interfaces.

Thus the distributed transaction processing reference standard has been upgraded to an object-oriented model, promoting software component reuse, and interprocess communication mechanisms have been cleanly defined, facilitating a common standard for vendor interoperability.

---

1 The X/Open Resource Manager implements the XA protocol used by the X/Open Transaction Manager to complete a distributed transaction. The respective prepare, commit and abort methods in the OTS environment are implemented by the Resource object. TX is a Transaction Manager interface used by the transactional client to begin, commit or abort a transaction. The equivalent invocations in the OTS are applied to the Current or Terminator object.
As an improvement to the X/Open reference model, the OTS is fully compatible with X/Open compliant software – in particular, the OMG requires that the OTS be able to import and export transactions to and from XA compliant resource managers and TX compliant Transaction Managers, respectively.

4.3.3.1 Typical Usage

Figure 4-9 illustrates the major components and interfaces defined by the Transaction Service. The interfaces are described in detail in section 4.3.4.

A typical transaction originator uses the Current object to begin a transaction. This object contains transaction information, including an object transaction identifier (OTRID) that uniquely identifies the transaction. The transaction context is then associated with the originator’s thread of control.

---

1 [OMG98a] Figure 10-2.
// associate new transaction with current thread
// or get existing transaction context, if already associated
org.omg.CORBA.Object initRef =
orb.resolve_initial_references("TransactionCurrent");

org.omg.CosTransactions.Current current = CurrentHelper.narrow(initRef);

if (current==null) {
    throw new org.omg.CORBA.TRANSACTION_REQUIRED("TX not available.");
}

If the current thread is already associated with a transaction, a nested subtransaction will be started:

try {
    // start transaction
    current.begin();
}

As the transaction originator issues requests to transactional (or non-transactional) objects, each of these requests is also associated with the transaction context. Propagation of the transaction context can extend to multiple levels if a transactional object itself issues a request to another transactional object. Using the Current object, the transactional object can unilaterally rollback the transaction and can inquire about the current state of the transaction. Using the Coordinator, a transactional object can determine the relationship between two transactions, to implement isolation among multiple transactions. The following code snippet is taken from the transactional object:

// get existing transaction context
// that is associated with current thread
org.omg.CORBA.Object initRef =
orb.resolve_initial_references("TransactionCurrent");

org.omg.CosTransactions.Current current = CurrentHelper.narrow(initRef);

Control control = current.get_control();
Coordinator coordinator = control.get_coordinator();
Most transactional objects will have some persistent data that must be managed as part of the transaction. In this case, the transactional object is also called a recoverable object. A recoverable object uses the *Coordinator* to register a *Resource* object as a participant in the transaction. A resource can only partake in a single transaction but a single transaction can have many resources. Again, some source code from the transactional object:

```java
Resource resource = create_new_resource(tx_name);
try {
    // register resource with the current transaction's coordinator
    coordinator.register_resource(resource);
} catch (Inactive e) {
    // transaction is inactive; termination may have started already
    throw new org.omg.CORBA.TRANSACTION_REQUIRED("TX not active.");
}
```

After the client has invoked several methods on the transactional object, it may terminate the transaction with a commit call on *Current*:

```java
try {
    current.commit(true);
} catch (Exception e) {
    // might be HeuristicMixed, HeuristicHazard or TRANSACTION_ROLLEDBACK
}
```

The *Coordinator* now uses the registered resource objects to perform the two-phase commit protocol on the recoverable object's state. It will first invoke `prepare()` on each Resource and then `commit()` to finish the transaction. This scenario does not include the optional synchronisation protocol.
4.3.3.2 Transaction Context

A client program may use direct or indirect (explicit or implicit) context management to manage a transaction. With indirect context management, a transaction originator uses the Current object provided by the Transaction Service, to associate the transaction context with the application’s thread of control. The ORB will then transparently send the context information together with each following method invocation, “piggy back” the context information in the GIOP request and response headers. This makes its propagation completely transparent. In direct context management, an application passes the context object as an explicit parameter to the server object.

A server object that supports implicit propagation would not typically expect to receive any Transaction Service object as an explicit parameter, since this way of propagation distorts the design of the object’s interface.

A client may use one or both forms of context management, and may communicate with objects that use either method of transaction propagation. It can suspend or resume the association between its thread of control and the context object using the respective methods of the Current object.

4.3.3.3 Synchronisation

The Transaction Service defines support for a synchronisation interface. A recoverable server may register a synchronisation object with the Transaction Service. This object will be notified prior to the start of the two-phase commit protocol, enabling it to flush transient state to the resource object that manages the persistence of this data, before the resource receives the prepare or commit_one_phase call, respectively. The synchronisation protocol also provides a method that is called after a successful commit and that can be used to clear

---

1 Propagation is the act of associating a client’s transaction context with operations on a target object. An object may require transactions to be either explicitly or implicitly propagated on its operations.
up temporary data or do other necessary processing after the outcome is complete. The synchronisation protocol is particularly useful in combination with X/Open Resource Managers that do not have access to the recoverable object’s transient state.

4.3.3.4 Exceptions

There are a few standard exceptions defined in the CORBA specification\(^1\) that may be returned as a result of any operation invocation, regardless of the interface specification:

- The **TRANSACTION_REQUIRED** exception indicates that the request did not carry a transaction context, but an active transaction is required.
- The **TRANSACTION_ROLLEDBACK** exception indicates that the transaction associated with the request has already been rolled back or marked to roll back. Thus, the requested operation either could not be performed or was not performed because further computation on behalf of the transaction would be in vain.
- The **INVALID_TRANSACTION** indicates that the request carried an invalid transaction context. For example, this exception could be raised if a transactional method invocation arrives at a resource that is registered with a different transaction context.

Additionally, the Transaction Service specification\(^2\) adds four heuristic exceptions that deal with unusual circumstances, such as communication failures, that may result in a loss of data integrity. “A heuristic decision is a unilateral decision made by one or more participants in a transaction to commit or rollback updates without first obtaining the consensus outcome determined by the Transaction Service.” [OMG98a].

---

1. [OMG98] Chapter 3, Section 3.15.
2. [OMG98a] Chapter 10, Section 10.2.6.
• The **HeuristicRollback** exception is raised by the commit operation on a resource reporting that a heuristic decision was made and that all relevant updates have been rolled back.

• The **HeuristicCommit** exception is raised by the rollback operation on a resource reporting that as a result of a heuristic decision, all relevant updates have already been committed.

• The **HeuristicMixed** exception may be raised by a request to report that a heuristic decision was made and that some relevant updates have been committed and others have been rolled back.

• The **HeuristicHazard** exception is the worst outcome of a transactional operation. In this case a heuristic decision may have been made, but the disposition of all relevant updates is not known. The resource raises this exception to indicate to the Transaction Service that its own state is not entirely known.

The Transaction Service interfaces also raise some additional exceptions that are described in section 4.3.4, along with the interface descriptions.

### 4.3.3.5 Transaction Models

The OMG Transaction Service supports the flat transaction model without checkpoints. There is one optional function, which is support for nested transactions. However, for an application to be portable across different implementations of the Transaction Service, it should be designed to use the flat transaction model. The Transaction Service Specification treats flat transactions as top-level nested transactions.
4.3.3.6 Transaction Integrity

Some Transaction Service implementations enforce "checked behaviour" for the transactions they support, to provide an extra level of transactional integrity equivalent to that provided by the interfaces which support the X/Open DTP transaction model. The purpose of checks is to ensure that all transactional requests made by the application have completed their processing before the transaction is committed. Unchecked transactions rely completely on the application to provide transaction integrity. This can lead to a loss of data integrity in case the transaction is committed before all changes are made durable.

4.3.4 Transaction Service Interfaces

Figure 4-9 (page 4-22) illustrates the IDL interfaces defined by the OTS specification, with an indication of the entities which use them. Each interface is defined in a module called CosTransactions. All IDL descriptions are to be found in the appendix. The lifecycle of objects belonging to the Transaction Service is handled transparently, thus there are no operations defined in these interfaces for destroying objects.

Current

This pseudo-interface\(^1\) allows a transaction client to begin and complete transactions. It also provides operations for suspending and resuming transactions, via which a thread can associate and disassociate itself from begun transactions. Use of the Current pseudo-object can be seen as an indirect way of accessing the "real" transactional interfaces, detailed below. Current has nine methods that control a transaction and present information about it.\(\text{begin}()\) creates a new transaction. If the client thread is already associated with a transaction, the new

---
\(^1\) "Pseudo" interfaces or objects, respectively, are not real distributed objects, although they are defined in IDL. They are not derived from CORBA::Object, and are always local to the process. Their main purpose is to give access to basic ORB services that would be impossible or too inefficient to implement as normal CORBA objects.
transaction is a subtransaction of that transaction. If the OTS implementation does not support nested transactions, the SubtransactionsUnavailable exception is raised.

The commit() and rollback() operations are forwarded to the Terminator object. suspend() and resume() offer the possibility to switch between implicit and explicit propagation model. Most of the other methods give or change parameters of the transaction.

**Control**

Instances of this interface should be considered to represent the transaction. It is simply an encapsulation of two other objects which provide methods for transaction manipulation: a Coordinator and a Terminator. Two methods are supported which return references to these contained objects (get_terminator() and get_coordinator()).

**Terminator**

The Terminator interface supports operations to commit or rollback a transaction. Typically, these operations are used by the transaction originator.

- commit(boolean report_heuristics) completes the transaction. It raises NoTransaction if there is no transaction currently associated with the thread of control or it raises the standard exception NO_PERMISSION if the thread has no permission to commit, e.g. if the OTS implementation restricts the commit operation to the transaction originator. If the report_heuristics parameter is true, the Transaction Service will report (possibly) inconsistent outcomes using the HeuristicMixed and HeuristicHazard exceptions. Depending on heuristic decisions the Transaction Service may rollback the transaction during the commit operation.

- rollback() rolls back all changes to recoverable objects made in the scope of this transaction. This operation may also raise the NoTransaction and NO_PERMISSION exceptions.
Coordinator

This interface provides a variety of methods for obtaining information about the transaction. It also exposes the `rollback_only()` method, by which the transaction may be marked for rollback without actually rolling it back. The main function of the `Coordinator` is to allow a `Resource` to register itself with the transaction, in order to be called back on transaction completion. You can also register a `Synchronisation` object with the `Coordinator`.

Transaction Factory

Besides the `resolve_initial_references()` method on an ORB interface the client can use the `FactoryFinder` interface of the Life Cycle Service to receive a `TransactionFactory` object. With two methods, `create()` and `recreate()`, it creates a new representation of a top-level transaction, returning a `Control` object.

Transactional Object

This empty interface is used by the OTS to determine if the transaction context should be implicitly transferred to a remote object. If the remote object inherits from `TransactionalObject` then the OTS transparently “piggy-backs” the transaction information to be extracted by the OTS library at the other end.

Resource

The `Resource` interface defines methods to be invoked during the two-phase commitment protocol used by the Transaction Service. Each registered resource is implicitly associated with a single top-level transaction. Its main methods are `commit()`, `prepare()` and `rollback()`. `commit_one_phase()` is invoked if this is the only resource participating in the transaction.
SubtransactionAwareResource

This is similar to a Resource, in that it is implemented by the user of the OTS, and is called back on transaction completion. However, it is specific to completion of a nested transaction.

RecoveryCoordinator

A recoverable object can register its Resource with a RecoverableCoordinator to drive the recovery process in certain situations of failure. After the transaction is prepared the server can call replay_completion(resource) on this object as a hint to the Coordinator that commit or rollback have not been called yet.

Synchronisation

This call-back object is implemented by the OTS user, and is registered with the Coordinator in exactly the same fashion as a Resource object. The OTS then invokes the methods before_completion() and after_completion() before and after the two-phase commit process. Synchronisation objects are intended to inform the server when to flush a temporary cache to a more persistent store, and can drive the release of locks acquired through an Object Concurrency Service (OCCS) interface.
5 Middleware Technologies

The recent developments in the middleware market appear chaotic, populated by many unknown products and low-profile vendors. Moreover, there is no universal definition of middleware but many different types ranging from database connectivity frameworks to transaction processing monitors. Different technologies melt together in order to combine functionalities that were separated in the past. There is a trend towards OMG standards and three-tier architectures that provide thin clients and highly scalable and reliable application servers. This chapter describes recent developments in software technologies and middle-tier products. After a short overview of "traditional" technologies that have dominated the market for business applications until early 1998, their evolution into new types of application server products is described.

The Gartner Group defines distributed computing middleware as "networking system software, layered between an application, the operating system and the network transport tier, that facilitates some aspect of communication for distributed computing". This "aspect" of communication includes an efficient session and presentation layer protocol (OSI layer 5 and 6) and additional services like directory, security and transactions.

Under this definition, the following technologies can be considered middleware:

- TP Monitor
- Message Broker
- Object Request Broker
- Database Gateways (ODBC/JDBC, OR-Mapping, OODBMS with extensions)
- Remote Procedure Call services (e. g., RMI)
- Object Transaction Managers

[1] [IAM99].
5.1 Traditional Technologies

The three-tier architecture (also referred to as multi-tier architecture) has emerged to overcome the limitations of the two-tier architecture, i.e., thick clients encapsulating business logic in their GUI, poor scalability, less reuse due to proprietary stored procedures, and other disadvantages. In the three-tier architecture, a middle-tier was added between the user interface client environment and the database management server environment. There are a variety of ways of implementing this middle-tier, such as transaction processing monitors and messaging servers. These solutions have shown to improve performance for a large number of users and to improve flexibility when compared to the two-tier approach. A limitation with three-tier architectures is that the development environment is more complex and more difficult to use than the visually oriented development of two-tier applications. This is mainly due to concurrency and networking problems.

5.1.1 Three-tier Architecture with TP Monitor

For the last twenty or so years Transaction Processing (TP) monitors have been used extensively for mainframe-based, monolithic applications. Over the last 3-5 years, a majority of these TP Monitors have become available on additional operating systems such as UNIX and NT. Without TP Monitors, every client is connected to the DBMS, each consuming an own database connection. The number of database connections is limited, thus being a bottleneck to database access. TP Monitors act as a database connection concentrator since the clients are now connected to the TP Monitor (middle-tier) and not directly to the DBMS. This reduces overhead and increases performance. The TP Monitor technology is a type of message queuing, transaction scheduling, and prioritisation service. The transaction is accepted by the monitor, which queues it and then takes responsibility for managing it to completion. It provides robust security and the ability to update multiple different DBMSs in a single transaction (by use of two phase commit technology). A limitation to TP Monitor
technology is that the implementation code is usually written in a lower level language (such as COBOL), and not yet widely available in the popular visual toolkits [SCH95].

Examples: IBM CICS, BEA Tuxedo, IBM Encina, NCR TOP END

5.1.2 Three-tier with Message Oriented Middleware (MOM)

Messaging is another way to implement three tier architectures. A message is prioritised and processed asynchronously. It is a self-contained object that carries information about what it is, where it needs to go, and what should happen when it reaches its destination. Messages consist of headers that contain priority information and the address of the queuing process. The message server connects to the relational DBMS and other data sources. The difference between TP monitor technology and message server is that the message server architecture focuses on intelligent messages, whereas the TP Monitor environment has the intelligence in the monitor, and treats transactions as dumb data packets. Messaging systems are good solutions for wireless infrastructures [SCH95].

Examples: IBM MQSeries, BEA MessageQ, Microsoft MSMQ, NEON NEONet

5.1.3 Three-tier with an Object Request Broker (ORB) Architecture

Developing client/server systems using technologies that support distributed objects holds great promise, as these technologies support interoperability across languages and platforms, as well as enhancing maintainability of the system. The key benefit of the object paradigm is that it provides a systems approach to the representation of business concepts, including persistent data and business logic.

There are currently two prominent distributed object technologies: The Common Object Request Broker Architecture (CORBA) from the OMG and the Distributed Component
Object Model (DCOM) from Microsoft. These architectures do not provide many ingredients necessary for Enterprise Network Applications. Additional services are needed to provide scalability and fault tolerance as it is expected from transaction processing environments. Therefore we can see DCOM being extended to a transaction processing system (MTS) and also CORBA ORBs being extended by third-party products.

"IONA was probably the first CORBA vendor to offer a real Object Transaction Monitor (OTM) package, Orbix-OTM. To create this product, IONA has combined its existing ORB with Transarc's Encina toolkit source code, which they licensed in 1997. The Encina elements have enabled IONA to add monitoring to its ORB and provide services like load balancing, roll backs, and transparent restarts for stateless servers." IONA has announced future support for Enterprise JavaBeans (EJB), Microsoft Transaction Server (MTS), and CICS. In this context, the term "stateless server" refers to a server that does not cache business state which might be accessed by many clients concurrently. This server type is only used for request processing.

As a competitor to IONA's Orbix-OTM Visibroker Integrated Transaction Service (ITS) is not an OTM. Instead, Visibroker ITS, which is written in Java, incorporates Sun's JTS standard, the Java implementation of the OMG's Transaction Service. JTS is a key element of EJB and this implementation will be a key component in the upcoming Inprise Enterprise Application Server based on EJB. One can expect a fully functional OTM product from Inprise in the future [HAR98].

---

1 [HAR98]
<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
<th>Features</th>
<th>Lang</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orbix-OTM</td>
<td>Object Transaction Manager (OTS)</td>
<td>- load balancing</td>
<td>C++</td>
</tr>
<tr>
<td>(IONA Technologies)</td>
<td>(based on Transarc's Encina)</td>
<td>- secure socket layer (SSL)</td>
<td>Java</td>
</tr>
<tr>
<td>Visibroker ITS</td>
<td>Integrated Transaction Service (ITS)</td>
<td>- option: extension for collaboration with IBM's CICS, IMS/TM, MQ Series, BEA's Tuxedo</td>
<td>Java</td>
</tr>
<tr>
<td>(Inprise Corp.)</td>
<td>(written in Java, as such a JTS)</td>
<td></td>
<td>C++</td>
</tr>
<tr>
<td>BEA M3</td>
<td>fully functional OTM based on BEA's TP monitor Tuxedo and the ORB &quot;ObjectBroker&quot;</td>
<td>- M3 Software Development Kit (TP framework, class library, ...)</td>
<td>C++</td>
</tr>
<tr>
<td>(BEA)</td>
<td></td>
<td>- BEA Engine (management and database connectivity)</td>
<td>future: Java</td>
</tr>
</tbody>
</table>

Table 5.1. CORBA ORBs on their way to Object Transaction Managers.

5.1.4 Three-tier with an Object DBMS

An extension on the theme of object request broker is the idea of using an object-oriented DBMS (ODBMS) together with an ORB as the middle layer. As an object-relational mapper and cache for relational data ODBMSs meet some of the requirements needed for a middle-tier server. Some vendors offer CORBA support via an ODAF² compatible framework, e.g., ObjectStore, Versant, Objectivity/DB and O2. Unfortunately this is only a simple way of bringing CORBA functionality to business objects and it lacks many important features like management of object identity and navigation, multithreading and inclusion of the OMG Object Transaction Service (OTS) and is therefore not an efficient distribution mechanism (Table 5-2).

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
<th>Components</th>
<th>Lang*</th>
<th>DOM**</th>
</tr>
</thead>
<tbody>
<tr>
<td>ObjectStore</td>
<td>Extension to ObjectStore 5.0 for legacy data access</td>
<td>- visual mapping tool</td>
<td>C++</td>
<td>CORBA via OODA</td>
</tr>
<tr>
<td>Dbconnect</td>
<td></td>
<td>- API: extensions to ObjectStore</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Object Design, Inc)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O2 Java Relational Binding (JRB)</td>
<td>development toolkit for relational DB access in Java</td>
<td>- development toolkit</td>
<td>Java</td>
<td></td>
</tr>
<tr>
<td>(Ardent Softw., Inc.)</td>
<td></td>
<td>- runtime library</td>
<td></td>
<td></td>
</tr>
<tr>
<td>O2 DBAccess O2 ODBC</td>
<td>connecting O2 applications to relational DBMSs</td>
<td>- development toolkit</td>
<td>C++</td>
<td></td>
</tr>
</tbody>
</table>

2 Object Database Adapter Framework, built by IONA Technologies conforming OMG/ODMG recommendations.
5.2 Application Server

Application Servers have become one of the hottest new Internet product categories. However, the definition of the term *application server* is too vague to be useful. Vendors and analysts employ this term to describe Web server extensions, middleware servers, object caches, object relational servers, deployment servers, and so on. This section tries to reveal the real distinctions between the products.

5.2.1 Introduction

When talking about 3-tier architectures today, most people mean the approach of an application server. With this approach, all of the application’s business logic is located at a shared host server, just like the X architecture of the 1980’s. The application server shares business logic, computations, and a data access engine (see Figure 5-1). Advantages are that with less software on the client (Web browser) there is less security to worry about, applications are more scalable due to multi-threading and database connection pooling, and support and installation costs are less on a single server than maintaining application logic on each desktop client [SCH95].

---

Sun Microsystems about their recently acquired “Netdynamics Application Server Platform”:

“An application server is the lynchpin of the modern computing environment -- layering over
the complexity of disparate databases, applications and legacy systems. The application server sits between Web servers and back-end data sources, running business logic on a middle tier and connecting various systems to HTML and/or Java clients.”

Definitions like these are not commonly accepted. The problem is, that there is no real definition about which functionality an application server has to offer or which type of application it is thought to run. Five articles in popular computer press will produce five different definitions of what constitutes an application server and each software vendor abuses this term claiming to be among the first in the market segment of development tools and runtime environments for business logic. All these different types of application server products have one feature in common. They are all construed to be highly scalable, capable of serving thousands of potential clients in the big world of the Internet. Therefore, some people are also talking of “Internet application servers”.

Figure 5-1, Figure 5-2, Figure 5-3 and Figure 5-6 are adapted from [RYM98].
In order to prepare the reader for the next sections, some new terms around the various aspects of application processing have to be defined first:

- **Front office.** Most application servers are designed for front office processing, replacing the Web’s Common Gateway Interface (CGI) as a means for browser users to access databases and applications that process corporate data, thus mediating between the page-based Web and the record- and transaction-oriented world of business applications. The term “front office” does not reflect the concept of client tier (front office applications run on the middle-tier). It rather indicates the type of application, which is mainly order management and information publishing.

- **Middle office.** Wall Street coined the term “middle office” to describe applications that are independent of the front-office’s order management and the back office’s clearing and settlement operations: “[...] business logic is moving from ‘front office’ client/server desktops and from ‘back office’ data stores into middle-tier application servers, creating a ‘middle office’ which allows users to share information with optimal performance and flexibility.” [HPC98]. Middle office applications perform decision processing (see below) to deal with dynamic business situations. They process the business rules that govern the organisation’s activities. Therefore they have to be integrated with information and transactions processed by various back office applications. Examples are portfolio risk analysis, transportation and supply logistics, and comprehensive customer care.

- **Back office.** Back office systems include legacy databases and transaction processing systems like CICS or enterprise application systems like SAP. Here are core business applications and databases to be found. “The core back office processing is the reliable recording of the momentary status of business activities, including settlement and clearing.

---

1 These definitions comply with and partially quote [RYM98].
2 Wall Street Middleware Working Group (WSMWG) at a meeting on April 14, 1998, in New York. See [HPC98].
Middleware

in trading firms, reconciliation in banking, and inventory in manufacturing, distribution, and transportation.” Again, the term “back office” should not be mixed up with the database tier, it only describes a type of application.

- **Decision processing.** Decision processing analyses variables both inside and outside of the company to figure out what to do about a customer, supplier, or operational situation. “Decision processing applies filtering, correlation, rules processing, transaction processing, workflow, and other operations to a given set of conditions (inputs) to determine the best course of action (outputs).” See also [CUT99].

The following three sections describe different types of application servers. It is helpful to divide the different products into three categories, according to three different types of Internet applications:

![Figure 5-2. Types of Application Servers.](image)

- **Web Information Server.** Products that are designed to create Web sites and, in some cases, to link Web pages to a single database that can generate data to be inserted into the
HTML pages. Simply being Internet extensions to those databases, their primary purpose is request processing for Web users.

- **Component Server.** Products that provide database access and transaction processing services to software components including ActiveX, CORBA objects and JavaBeans (EJB). Component servers provide an execution environment for server-side components and access to back office databases and other services. These components do not represent or carry business state in the form of business objects, but instead offer stateless services for request processing.

- **Object Transaction Manager (OTM).** Products that are designed to run server-based business logic, usually by employing object (component) servers, while providing access to back office systems. In contrast to component servers, OTMs are stateful. They are not mere request handlers but they manage data and transactions against those data, performing decision processing tasks.

### 5.2.2 Web Information Server

The terms *Web Application Server* and *Web Information Server* denote the same thing. The first emphasises on what is actually done: application processing with application logic on the middle-tier. The second term emphasises on the main type of application, which is Information retrieval and request processing; this term is more meaningful, but less commonly used.

The architecture and functionality of Web application servers varies between different products. Some offer not more than servlet functionality and database access, others integrate with open protocols in order to access other services from the CORBA, DCOM or Java
Middleware

world. They also differ very much in the number of clients they can serve concurrently. This depends on the thread model and the way they deal with database connections.

Following the demand for highly scalable middle-tier servers for new Intranet and Internet applications Web Information Servers were first designed to handle hundreds of simultaneous Web requests and to create Web pages in reply. They are mere request brokers generating Web displays and publishing information both internally, to employees, and externally, to customers (see Figure 5-3). They have since pushed the use of the Web beyond information publishing towards management of commerce. Enabling users to interact, they not only display customer service information but also allow self-service to a certain extend. As time goes by, vendors of Web application servers are pushing their products more and more into the middle office territory, adding features and functions to handle application logic and transaction processing.

The advantage of a Web application server is the ability to integrate various products of different vendors (IDEs, management consoles, database access, Web server, transaction server) in a single consistent platform. As a mediator between client and database an application server also serves as a uniform security system for different database systems.

![Stateless Web Information Server and Component Server](image_url)

*Figure 5-3. Web Application Server and Component Server functions.*
The following enumeration lists important features a Web application server should offer. Figure 5-4 summarises its architecture.

- High scalability
- Load balancing and fail-over
- Integrated Development Environment (IDE) for Java and HTML, offering simple access to databases and also full Web functionality
- Authentication and encryption
- Centralised application management
- Transactions against different types of database systems, also legacy systems
- Integration of open services and protocols (e.g., CORBA, XML)
- Firewall compatibility of HTML and Java pages

![Figure 5-4. Typical architecture of a Web Information Server.](image)

Different Web application servers use different kinds of state management. Although they are called “stateless” (they do not cache business objects), they need to maintain information about a client session, thus require some kind of state management. Web servers provide
state mechanisms that rely on client-based storage, called "cookies". Java-capable Web servers that support servlets can maintain session state in the servlet itself, if designed to do so (Web information servers are primarily servlet servers). In this case, it is necessary to transparently include a client session identifier in each HTML page that is sent to the client. Since HTTP is a connectionless protocol, the identifier would always have to be transmitted when sending some user input to the server, in order to be able to identify the client and link its requests to the correct state information. Alternatively, if the client consists of an applet, some state information could be held on the client. The simplest way to deal with state is by using a connection-oriented protocol like RMI or IIOP, but this is not a common feature among Web application servers.

<table>
<thead>
<tr>
<th>Title</th>
<th>Components</th>
<th>Lang*</th>
<th>Com**</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apptivity</td>
<td>- visual component library</td>
<td>Java</td>
<td>IIOP (Orbix)</td>
</tr>
<tr>
<td>(Progress Software)</td>
<td>- data-aware controls</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- several wizards</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- CORBA productivity tools</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NetDynamics</td>
<td>- CORBA-based Java application server</td>
<td>Java</td>
<td>IIOP (Visibroker)</td>
</tr>
<tr>
<td>Application Server</td>
<td>- platform adapter components</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Sun Microsystems)</td>
<td>- command center</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Java object framework</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- development studio</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Netscape Application Server</td>
<td>- application server environment</td>
<td>Java</td>
<td>IIOP (Orbix)</td>
</tr>
<tr>
<td>(Netscape)</td>
<td>- application builder</td>
<td>C++</td>
<td>COM</td>
</tr>
<tr>
<td></td>
<td>- deployment manager</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sapphire Web</td>
<td>- development environment</td>
<td>Java</td>
<td>IIOP (Visibroker, Orbix, Sun), COM</td>
</tr>
<tr>
<td>(BlueStone Softw.)</td>
<td>- application server</td>
<td>ANSI C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- management engine</td>
<td>KR-C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- plug-ins</td>
<td>C++</td>
<td></td>
</tr>
<tr>
<td>WebSphere Application Server</td>
<td>- servlet engine</td>
<td>Java</td>
<td>-</td>
</tr>
<tr>
<td>Standard Edition</td>
<td>- management tools</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(IBM)</td>
<td>- IBM HTTP server</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- extensions for third-party products</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Supported language (Java/C++)

** Communication Options beside Java (IIOP, COM)

Table 5-3. Examples for Web Information Server.¹

¹ The products listed in Table 5-3 to Table 5-5 reflect the developments during 1998 and early 1999. As all these products evolve, they move towards the OTM territory.
5.2.3 Component Server

The line between Web Information Servers and Component Servers is very thin, as evolving Web Server products also add component technology (e.g., Apptivity). Both are stateless servers that only perform request processing, as shown in Figure 5-3. Component servers have two functions. First, they provide an execution environment for server-side components. Second, they provide access to back office systems (databases, TP monitors, SAP, …) and other services.

Components usually live in containers that offer them a wide range of services like persistence, transactions, security, life cycle management, load balancing and so on. There is a strong tendency towards Enterprise Java Beans (EJB) and CORBA components. Both provide component interoperability across different platforms and, which is very important, across different development tools, since EJB and CORBA are open standards. Moreover,

---

1 Stateless with regard to business state, not session state.
the Object Management Group (OMG) has incorporated a mapping for EJB as part of the proposed CORBA Component Specification and EJB components use IIOP as the underlying communication protocol. Nearly every middleware vendor has promised to support EJB or may already do so and it is quite likely that even Microsoft will eventually support EJB containers for DCOM. One reason why ActiveX components do not play a major role anymore is because of their lack of a reliable security system, which is a substantial part in future e-commerce applications.

<table>
<thead>
<tr>
<th>Title</th>
<th>Components</th>
<th>Lang*</th>
<th>OM**</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTS (Microsoft)</td>
<td>- DCOM based</td>
<td>any (ActiveX)</td>
<td>DCOM</td>
</tr>
<tr>
<td></td>
<td>- run-time environment</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- graphical administration tool</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- API for automation of administration</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inprise Application Server</td>
<td>- visual development environment</td>
<td>Java</td>
<td>CORBA</td>
</tr>
<tr>
<td>(Inprise)</td>
<td>- application server runtime</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- management console</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Web server</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Integrated Transaction Service (ITS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- VisiBroker SSL Pack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oracle Application Server</td>
<td>- application server</td>
<td>Java</td>
<td>CORBA</td>
</tr>
<tr>
<td>(Oracle)</td>
<td>- CORBA 2.0-compliant ORB</td>
<td>Perl C</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Java environment</td>
<td>PL/SQL LiveHTML</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Cartidges for PL/SQL, LiveHTML, PERL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- system management tool</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Orbix OTM (IONA Technologies)</td>
<td>- CORBA services: Naming, Trader, Security, Events</td>
<td>C++</td>
<td>CORBA</td>
</tr>
<tr>
<td></td>
<td>- Object Transaction Service (OTS) based on Transarc's Encina</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sybase Enterprise Application</td>
<td>- execution engine</td>
<td>Java</td>
<td>CORBA</td>
</tr>
<tr>
<td>Server (Sybase)</td>
<td>- administration and monitoring tool</td>
<td></td>
<td>ActiveX</td>
</tr>
<tr>
<td>WebSphere Application Server</td>
<td>- EJB engine</td>
<td>Java</td>
<td>CORBA</td>
</tr>
<tr>
<td>Advanced Edition (IBM)</td>
<td>- servlet engine</td>
<td></td>
<td>ActiveX</td>
</tr>
<tr>
<td></td>
<td>- management tools</td>
<td></td>
<td>JavaBeans</td>
</tr>
<tr>
<td></td>
<td>- IBM HTTP server</td>
<td></td>
<td>EJB</td>
</tr>
<tr>
<td></td>
<td>- extensions for third-party products</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Supported language (Java/C++)

** Component Object Model (CORBA, ActiveX (COM), DCOM, JavaBeans, EJB)

Table 5-4. Examples for Component Server.
5.2.4 Object Transaction Manager (OTM)1

[MCC98] writes: “Object Transaction Managers represent the best of all worlds for mission-critical enterprise systems. OTMs combine the flexibility of component computing with the ease of use of client/server systems and the integrity and reliability of transaction processing middleware.”

OTMs are the object-based counterpart of TP monitors. They should support the object model without compromising the scalability, stability and performance characteristics offered by TP monitors. Transaction management, load balancing, automatic failover and session concentration belong to the standard features. OTMs should also combine support for the synchronous communication model (found in most of the middleware categories, such as Corba and COM) with support for the asynchronous communication model (found in message-oriented middleware products such as Microsoft MSMQ and IBM’s MQSeries). See [DOL98].

[HAR98] writes: “This type of product is necessary if companies are going to use objects or components on Internet-based systems and want to handle large numbers of clients at once. More to the point, this approach is required if companies want to process financial transactions over the Internet.”

The main difference between traditional TPM systems and object-oriented TPM systems as well as between Component Servers and Object Transaction Managers is the presence of state in objects. In other words, a procedure is just an operation, but an object is both a set of operations and attributes with associated data values. These objects support server-side

---

1 The term “Object Transaction Manager” was coined by Gartner Group. Also known as “Object Transaction Monitor”.

5-16
business logic which can be part of distributed transactions. Stateless servers rely either on
databases or back office systems like transaction monitors to manage business data. For each
application task that touches business data, the stateless application server must submit a
request to the appropriate back office system for the data, load the data into memory, and
then submit changes to the back office system. This process causes overhead on virtually
every application operation. Stateful servers are a way to reduce this overhead, and improve
performance and flexibility by caching business objects (data and logic), thus enabling the
application server to perform most of the business logic (see Figure 5-6).

By default, in Object Transaction Managers a business object’s state remains in memory once
activated, even across transaction borders. This enables the objects to be shared between
different application sessions and offers an efficient way for different clients to work on these
objects concurrently. This requires a highly sophisticated execution environment that
manages the object life cycle: When the number of clients accessing these objects grows large,
memory gets full, the system starts to page, and response time deteriorates. Therefore, any
system that combines TPMs and ORBs for large-scale applications needs to develop a way of
managing client and server state more efficiently. In the upcoming version of the CORBA
specification this problem is addressed by the Portable Object Adapter (POA), which is already used in the BEA M3 TP Framework and the GemStone/J application server.

<table>
<thead>
<tr>
<th>Title</th>
<th>Components</th>
<th>Lang*</th>
<th>OM**</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEA M3 (BEA)</td>
<td>- M3 Software Development Kit (TP framework, class library, ...)</td>
<td>C++ future: Java</td>
<td>CORBA</td>
</tr>
<tr>
<td></td>
<td>- BEA Engine (management and database connectivity)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Forté (Forté)</td>
<td>- application server environment</td>
<td>TOOL Java</td>
<td>proprietary</td>
</tr>
<tr>
<td></td>
<td>- GUI generation tool</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- data-aware GUI components</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- central management (load balancing, ...)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- integration with CORBA and COM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gemstone/J (Gemstone)</td>
<td>- application server for EJB and CORBA components</td>
<td>Java</td>
<td>CORBA EJB</td>
</tr>
<tr>
<td></td>
<td>- Java Object Transaction Monitor (OTM)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- security: SSL 3.0, JSA, JCA, JCE</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- centralized management</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Persistence PowerTier Server (Persistence Software)</td>
<td>- application server development</td>
<td>Java C++</td>
<td>CORBA</td>
</tr>
<tr>
<td></td>
<td>- &quot;automates development of robust CORBA servers&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- integration with object modeling tool</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rational Rose</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secant Extreme Persistent Object Service (Secant Technologies)</td>
<td>- development and runtime environment</td>
<td>Java C++</td>
<td>CORBA</td>
</tr>
<tr>
<td></td>
<td>- integration with Rational Rose</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- schema compiler</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Supported language (Java/C++)

** Integration of any Object Model (CORBA, RMI, DCOM, EJB)

Table 5-5. Examples for Object Transaction Managers (OTM).

5.3 Where do we go from here?

One can clearly see that different middleware products are melting together, heading towards a universal solution for all scalability and reliability needs of the Internet age. This solution is called Object Transaction Manager or Object Transaction Monitor and combines the open and object-oriented platform of CORBA ORBs with transaction processing features of traditional TP monitors. Most vendors license CORBA ORB products in order to support
IIOP and the OMG Object Transaction Service (OTS) standard. Additionally, there is a trend to Java and Enterprise JavaBeans as the preferred component architecture.

Sun licences Inprise’s Transaction Service for the NetDynamics Application Server and plans to integrate this package into the operating system Solaris in order to compete with Microsoft MTS. BEA acquired WebLogic’s Java-based Internet application server in order to extend its transaction middleware with electronic commerce and intranet applications. Hitachi adds the Visibroker ORB and OTS interface to its TPBroker product. Oracle adds the Visibroker ORB and also the VisiBroker for ActiveX (DCOM) Bridge to its application server platform.

“Clearly, the battle has shifted from RDBMS/stored procedure-based TP-lite versus TP monitor based TP-heavy to a battle between DCOM-based MTS versus CORBA-based OTS. MTS is certainly ahead in price, ease of entry and simplicity. OTS is ahead in scalability and cross-platform support, given that the traditional TP monitor vendors are behind it. By the year 2000 or so, expect to see MTS dominate at the low-to-midrange and OTS to dominate at the high end.” [GAL97]
6 Implementation of a Business Object Framework

This chapter illustrates the implementation of the business object framework that evolved as a result of this research thesis. It starts by describing the design and functionality of the whole framework, continues with a description of the separate parts (mini-frameworks) and closes with some examples for business applications that require the state management and distributed transaction facility provided by the framework.

6.1 Introduction

The prototype implementation in the following will be referred to as Business Object Framework (BOF)\(^1\) or just framework. The Business Object Framework provides an abstraction layer which sits between the business object and the ORB’s object adapter. It serves as a container in which business objects live and offers generic services like activation and deactivation (garbage collection), transactions\(^2\), database access or query functionality. These capabilities are coded separately from normal business functions and are transparent to the client (see Figure 6-1).

\[\text{Figure 6-1. Business Object Framework in relation to the ORB and business functions.}\]

\(^1\) BOF is not to be mixed up with the abbreviation for Business Object Facility used by the OMG.

\(^2\) Throughout this chapter, 'tx' will be used as an abbreviation for 'transaction'.

6-1
Implementation

What exactly does the term “Business Object Framework” describe? It looks as if it is just another term for a middleware concept we know as Application Server or Object Transaction Manager. In fact, “framework” is a technical term for an environment, which offers various services. It serves as a container for components (Component Server), it is a server environment for highly scalable business applications (Application Server), and it is an object server that takes part in distributed transactions (Object Transaction Manager). Other services are persistence, security, concurrency, naming, state management, etc. The prototype implementation can be considered as the CORBA equivalent for an Enterprise JavaBeans (EJB) server, except for the fact that the present framework concentrates on persistent objects (entity beans\(^1\)) and does not offer session objects (session beans) that are unshared and dedicated to a single client each.

When the research for this thesis started, there was no component model for business components available, neither had anybody coined the different terms for server environments listed above. In December 1997, Sun announced the draft for its Enterprise JavaBeans specification and publicised the first version in March 1998. Also Microsoft introduced a new component model by delivering the Microsoft Transaction Server (MTS). The OMG is still working on a component model specification which seems to become more or less a CORBA mapping for the EJB specification. EJB developed to the favourite component model in 1999: Forté Software and IBM moved from proprietary component models towards EJB and most of the other players in the young application server market support EJB as well.\(^2\) The present implementation of a business object framework addresses all relevant issues from database access to scalability to distributed transactions. It does not represent a complete production framework and there are a lot of implementation details that

---

\(^1\) See chapter 2.

\(^2\) Companies supporting the EJB technology or having announced to support EJB technology in a future release of their products include BEA, Bluestone, Gemstone, IBM, Inprise, Netscape, Oracle, Persistence, Progress, Secant, Siemens, Sun; March 1999.
Implementation

would have to be added or optimised to make it a stable and rounded product. The following sections point out these details and suggest further extensions and amendments. Yet, the framework could be a basis for an implementation of the upcoming OMG Component Model specification. The interfaces used for business object and type manager are a subset of the interface specifications proposed in submissions to the OMG Business Object Domain Task Force in 1998.¹

6.2 Design

The BOF prototype was designed to be easily portable among different ORBs, extensible with regard to different database systems and additional functionality, modular, i.e., made up of exchangeable components², and configurable. It can easily be complemented by a GUI tool that allows definition of business objects, their logic and properties, as well as generation of source code necessary to integrate the business objects into the framework. Parts of the framework are implemented according to the research summarised in the preceding chapters. The reader will find references to the specific sections.

Figure 6-2 illustrates the main components of the system. The disc in the middle contains the components that are situated in the same operating system process. Type manager, business object and resource have a CORBA interface each. A CORBA client uses the interfaces of type manager and business objects, the Object Transaction Service (OTS) uses the resource interface during transaction completion.

¹ Relevant documents are [BUS98a] and [BUS98b].
² The term components in this context is not used according to common componentware definitions like in [HOF97], but it merely designates parts of the whole framework, often called mini-frameworks. If componentware is addressed, the term software component is used instead (see glossary for a definition).
Figure 6-2. The different components of the business object framework.

- The *type manager* provides the client with factory and query methods that return business objects. It collaborates with the ORB’s *object adapter* and *activator* (which will be combined in the future POA) in order to activate and deactivate business objects. It also manages a *cache* of all active business objects.

- The *business object* itself offers some methods concerning its state and relationships. The business domain object, which contains the actual business logic, inherits from the *BusinessObject* interface and presents itself to the client with a set of domain-specific methods that process the business logic.

- The *resource* is the CORBA equivalent for an XA resource. Its public interface is used by the Object Transaction Service (OTS) only and takes part in the termination of
distributed transactions. The resource does not share the state of the persistent object but rather copies the state to its own transactional cache.

- The *datastore* component can be a separate software component (e.g., a third-party product) but for efficiency purposes it should run in the same address space.

- The *garbage collector* runs in its own thread but it should not be implemented as a separate software component because it accesses methods that should be invisible to any CORBA clients.

- There is also a *configuration component* which is omitted in Figure 6-2 because it plays only a minor role. It initialises and monitors the process, its threads and current state.

- The *Domain Manager* is for the Business System Domain (BSD)\(^1\) what the type manager is for one specific business object type. It manages different type managers and can be used to manage replication of BOF processes and to produce parts of object identifiers according to the High/Low approach outlined in chapter 3 (section 3.3.2.3).

- The framework uses the OMG *Object Transaction Service* to process distributed transactions. It could also use Naming, Concurrency Control and even the Query Service, but these services are not necessary for a fully working system.

Each component is implemented as a mini-framework and is explained in detail in one of the following sections.

### 6.3 Object Model

The overall framework is rather complex but the main functionality hides in just a few classes or mini-frameworks. These are:

\(^1\) See chapter 2 and [BUS98b].
Implementation

1. Main program (initialisation).
2. Configuration (settings for the business object, its methods and transaction handling).
3. Resource object (integration of the business logic, implementation of the two-phase commit protocol and locking issues).
5. Persistent state and types (implementation of orthogonal persistence).
6. Type manager (business object factory and manager).
7. DataStore (generic database component).

These classes are described in detail in the following sections. Each is illustrated by a class diagram containing the relevant associations to other classes, by activity diagrams and some code snippets. A complete class diagram of the overall framework can be found in the appendix.

Some of the sample code in the following sections is based on the sample IDL interface for a bank account. A bank account is for distributed transactions what 'Hello world!' is for the first programming lessons. Performing a money transfer from one Account object to another requires a distributed (OTS) transaction with the two-phase commit protocol.

IDL description for Account:

```idl
#pragma prefix "kus.vas"
module example {

    interface Account : bof::BusinessObject {
        float balance();
        void credit(in float amount);
        void debit(in float amount);
    };
}
```
6.3.1 Main Program

The main program is quite short and merely initialises the main components of the system. The start-up process operates as follows. Only one business object type per process is assumed; irrelevant lines are omitted.

```java
// initialise ORB and BOA
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
org.omg.CORBA.BOA boa = orb.BOA_init();

// test if transaction service is available
if (orb.resolve_initial_references("TransactionCurrent")) {
    transaction_service_startup = true;
}

// connect to DomainManager using a proxy object
DomainManagerProxy dom = new DomainManagerProxy("DomainManager");

// Activator simulates different POA instances
BOFactivator act = new BOFactivator(dom);

// initialise configuration
Configurationlmpl con = new Configurationlmpl(args[i+1]);
// register configuration object with BOA
boa.obj_is_ready(con);

// initialise type manager
// with given configuration, domain manager and activator
TypeManagerlmpl typ = new TypeManagerlmpl(con, dom, act);
// register type manager object with BOA
boa.obj_is_ready(typ);
```

Note: if there are more than one type manager objects (more than one business object type) running in the server process, the Activator will need the domain manager for routing an incoming request to the appropriate type manager. The Activator is explained in detail in section 6.3.5.

6.3.2 Configuration

Information about user-defined business object classes and about the configuration of database access, transaction isolation\(^1\), garbage collector policy, marshalling\(^2\) and so on is

---

\(^1\) See chapter 4, section 4.2.6.

\(^2\) See glossary.
Implementation

saved in a separate configuration file and loaded during server start-up. The filename is given as an input parameter to the executable. The configuration file is created as a result of the development process with a GUI-based development tool and contains the serialised state of a configuration class. In the present prototype implementation the configuration information is hard-wired in the ConfigurationImpl class, since there is not yet a development tool available that would create an appropriate file.

6.3.2.1 Initialisation

The constructor of ConfigurationImpl takes the filename of the configuration file as an input parameter. During start-up the configuration object also instantiates the following objects, according to the information given in the configuration file:

- a database component (vas.db.I_DataStore),
- a garbage collector component (vas.own.I_GarbageCollector),
- a factory object for OTS resources (vas.own.I_ResourceFactory), and
- a B0state object which contains and duplicates the initial state for each newly created BusinessObject instance.

The datastore object is later requested by business objects during local transactions or by resource objects during distributed transactions to load and update persistent state. The factory object for OTS resources is used by BusinessObjectTIE to create instances of B0resource objects when starting or taking part in a distributed transaction. Figure 6-3 shows the dependencies of various classes to ConfigurationImpl.
During process lifetime, the associated objects request information from the configuration object:

- The type manager requests information about the business object type, the datastore component and the ResourceFactory.

- The business object requests a reference to the I_DataStore component, the initial state for a newly created business object instance, marshalling information², information about access mode (read-only/read-write) and transaction policy (distributed, local, none).

---

¹ The following is a short guide to UML notation (see [UML97a] for details).

- **Generalisation.** Solid line with a hollow triangle at the end of the more general element.
- **Refinement.** (implementation of an interface). Dashed generalisation symbol.
- **Association.** Solid line connecting two class symbols. The association may have a name and roles, and may show cardinality and navigability.
- **Aggregation.** Hollow diamond attached to the end of the association path.
- **Dependency.** Dashed arrow. Here, indicating an ‘instantiates’ relationship.
- **Stereotypes.** Meta information placed above the class name within French quotation marks («Skeleton» for the ORB skeleton class, «Interface» for a Java interface class).

² Marshalling information contains knowledge about the structure and the types of parameters during method invocations. It is part of the communication protocol. See also the definition in the glossary.
type name and a time stamp. The unique time stamp is later requested from the business object during garbage collection.

- The BObject object, which implements the business object cache as a Java hash map, requests information in the initialisation phase only (when created by the type manager), about the maximum size of the hash map and the percentage of objects to be deactivated during garbage collection.

6.3.2.2 Configuration Information: Transaction Policy

There are five methods in the TypeManager and BusinessObject interfaces as well as $n$ methods in the domain interface (e.g. Account) which result in a database update, rather than read-only access. These methods deal with persistent state and may take part in distributed (OTS) transactions. However, the business object developer may decide different policies for different methods. This feature allows for performance optimisations according to the reliability needed for the application. For instance, an Account object taking part in money transfer may allow only OTS transactions on its methods withdraw and deposit in order to perform a proper two-phase protocol, but an Order object created on request of an Internet client may also allow local transactions against the local database. Other methods may not require transactions at all, because they are read-only or do not need to be reliable (which is very seldom the case for business components, of course). Table 6-1 shows the respective methods including the Account example, together with the possible transaction policies. Default policies are checked, a hyphen means "not possible". Note that read-only methods are always processed without a transaction. During the development of business objects with the development tool (code generator) the developer decides which policies shall apply to the various methods.

A hash map maps keys to values (OIDs to Java object references). A map cannot contain duplicate keys; each key can map to at most one value.
<table>
<thead>
<tr>
<th>Method Name</th>
<th>Transaction Policies¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. TypeManager::create_object</td>
<td>—</td>
</tr>
<tr>
<td>2. TypeManager::create_from_object</td>
<td>—</td>
</tr>
<tr>
<td>3. TypeManager::resolve_persistent_id</td>
<td>✓</td>
</tr>
<tr>
<td>4. TypeManager::query</td>
<td>✓</td>
</tr>
<tr>
<td>5. BusinessObject::get_PersistentID</td>
<td>✓</td>
</tr>
<tr>
<td>6. BusinessObject::is_identical</td>
<td>✓</td>
</tr>
<tr>
<td>7. BusinessObject::get_TypeManager</td>
<td>✓</td>
</tr>
<tr>
<td>8. BusinessObject::add_to_relationship</td>
<td>— — ✓</td>
</tr>
<tr>
<td>9. BusinessObject::remove_from_relationship</td>
<td>— — ✓</td>
</tr>
<tr>
<td>10. BusinessObject::will_notify_for_relationship</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>11. BusinessObject::delete</td>
<td>— — ✓</td>
</tr>
<tr>
<td>12. Account::balance</td>
<td>✓ ✓</td>
</tr>
<tr>
<td>13. Account::credit</td>
<td>✓</td>
</tr>
<tr>
<td>14. Account::debit</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 6-1. Transaction policies for TypeManager, BusinessObject and domain object methods.

The two factory methods (1, 2) are processed as local transactions by default but could also be processed without transactions at all. If the creation of a business object fails after performing the database update, the consistency of the application may be spoiled. But this may not be important to some applications which only require fast object creation. The next two methods (3, 4) cause a database query without changing anything. They do not need the reliability provided by an ACID transaction.

¹ See section 6.3.4.3 for a description of these transaction policies.
The same applies to the read-only methods of the BusinessObject interface (5, 6, 7, 10). The two methods that change relationships (8, 9) are required to be performed within an OTS transaction. In this framework, relationships between business objects are assumed to be bi-directional and managed, so that a change in one business object triggers a change in the related business object as well. Since the two related business objects can live in different processes, a two-phase commit protocol is required to perform a transaction. If the client has not started an OTS transaction prior to an invocation on (8) or (9) the framework will start one before and commit it after the method call.

The delete method (11) may only be processed within transaction boundaries. It depends on the application whether this should be a local or a distributed transaction.

The domain-specific methods debit and credit are required to be processed within the boundaries of an OTS transaction. Only then can it be assured that a money transfer from one Account object to another is atomic. It is not sufficient just to have the framework start an OTS transaction (in the case of which the next client call would result in a separate transaction), but rather the client is obliged to start and commit the transaction.

6.3.2.3 Configuration Information: Marshalling information

The marshalling information is a key element for integrating a CORBA interface into the ORB framework. It provides appropriate information for the object adapter to demarshal and forward requests to the invoked method implementations.

The following two code snippets are generated for the Account interface. The first method returns the three Interface Repository Identifiers that apply to the Account object. They tell, which interfaces are implemented by Account: the Account interface itself and the inherited interfaces BusinessObject and TransactionalObject.
public java.lang.String[] __ids() {
    java.lang.String[] tmp = {
        "IDL:kus.vas/example/Account:1.0",
        "IDL:kus.vas/bof/BusinessObject:1.0",
        "IDL:omg.org/CosTransactions/TransactionalObject:1.0"
    };
    return tmp;
}

The second method produces routing information that is used by the `Basic Object Adapter` to route the incoming request to the appropriate method. A `MethodPointer` contains the name of the method, the interface identifier and the method identifier.

public org.omg.CORBA.portable.MethodPointer[] __methods() {
    org.omg.CORBA.portable.MethodPointer[] methods = {
        new org.omg.CORBA.portable.MethodPointer("balance", 0, 0),
        new org.omg.CORBA.portable.MethodPointer("credit", 0, 1),
        new org.omg.CORBA.portable.MethodPointer("debit", 0, 2),
        new org.omg.CORBA.portable.MethodPointer("get_PersistentID", 1, 0),
        new org.omg.CORBA.portable.MethodPointer("is_identical", 1, 1),
        new org.omg.CORBA.portable.MethodPointer("get_TypeManager", 1, 2),
        new org.omg.CORBA.portable.MethodPointer("add_to_relationship", 1, 3),
        new org.omg.CORBA.portable.MethodPointer("remove_from_relationship", 1, 4),
        new org.omg.CORBA.portable.MethodPointer("will_notify_for_relationship", 1, 5),
        new org.omg.CORBA.portable.MethodPointer("delete", 1, 6),
    };
    return methods;
}

The marshalling information shown above should be generated by a development tool. The present implementation of BOF does not provide such a tool and requires the business object programmer to input this information manually.

### 6.3.2.4 Configuration Information: Initial Business Object State

Newly created business objects can be initialised with default values, other than the default values provided by the Java environment. If the client does not provide initial values as a parameter to the factory method `TypeManager::create_object(in NameValues init_values)`, these default values apply to the new object. The method `ConfigurationImpl::initializeState()` builds up a hash map containing the initial state. This hash map will afterwards be aggregated by a `BusinessObjectTIE` instance.
Implementation
code generator has to insert the lines in the middle. In the example given below, the initial state for an account object is inserted into a hash map object, containing only one attribute: a variable named “amount” with the initial float value 0.0.

```java
public BOstate initializeState()
{
    // create new HashMap(initialCapacity, loadFactor)
    BOstate tmp = new BOstate(10, (float)0.8);

    //************************************ initialise attributes ************************************/
    vas.types.PtypeBase attribute = new Pfloat((float)0.0);
    tmp.put("amount", attribute);

    //************************************ end initialise attributes ************************************/
    return tmp;
}
```

6.3.3 Resource

Before describing the business object itself in detail in the next section this one introduces the concept of distributed transactions and concurrency. This may help to understand the de-marshalling issues in the “Business Object” section. Besides, the BusinessObjectTIE class only delegates method invocations to a resource object, which actually implements the business logic.

6.3.3.1 Integration of Business Logic

Using the Account example mentioned above the following class diagram illustrates the relationships between business object, resources and business logic implementation. Besides the configuration class, the code generated by the development tool consists of the classes AccountResourceFactory and AccountResource. Using the factory pattern\(^1\), the framework does not have to be aware of the specific type of resource, e.g. Account. Instead, the BusinessObjectTIE receives an instance of type I_ResourceFactory from

\(^1\) See [GAM94].
the configuration class and invokes the method \texttt{I\_ResourceFactory::createResource} on it. Furthermore, \texttt{BusinessObjectTIE} does only communicate with the \texttt{BOresource} type. \texttt{BOresource} is an abstract class that relies on the derived class, e.g. \texttt{AccountResource}, to implement the skeleton code in the method \texttt{BOresource::\_execute}. The skeleton code is shown below.

```
public boolean \_execute(  
    int \_method_id,  
    org.omg.CORBA.portable.InputStream \_input,  
    org.omg.CORBA.portable.OutputStream \_output)  
{
...
```

Figure 6-4. Class diagram: Implementing the factory pattern.

Skeleton code in \texttt{AccountResource::\_execute}; only the marshalling code for two methods is shown: \texttt{BusinessObject::add\_to\_relationship} and \texttt{Account::credit}.
// in case the client tries to invoke on an object that the client
// itself has deleted short before during this tx
if (_marked_delete == true) {
    throw new org.omg.CORBA.OBJECT_NOT_EXIST();
}

switch(_method_id) {
    case 1003: { // BusinessObject::add_to_relationship
        try {
            java.lang.String relationship_name;
            relationship_name = _input.read_string();
            vas.bof.BusinessObject member;
            member = vas.bof.BusinessObjectHelper.read(_input);
            add_to_relationship(relationship_name,member);
        } catch(vas.bof.BOexception _exception) {
            vas.bof.BOExceptionHelper.write(_output, _exception);
            return true;
        }
        return false;
    }
    case 2: { // Account::credit
        float amount;
        amount = _input.read_float();
        debit(amount);
        return false;
    }
} // end of switch
throw new org.omg.CORBA.MARSHAL();

The skeleton code receives an input stream, which contains all input parameters, and an output stream, which contains all output parameters. If an exception occurs, the output stream only contains the exception object. The last method call inside each of the marshalling sections is the call to the method that performs the actual business logic. This method is implemented by the business object developer and inserted into the AccountResource class by the code generator, e.g.:

    public void debit(float amount)
    {
        System.out.println("Account.debit(" + amount + ")");
        _amount.value(_amount.value()-amount);
    }
Implementation

The following five steps summarise the process of marshalling and de-marshalling, respectively:

1. The Object Adapter receives a method call and forwards it to the addressed object (BusinessObjectTIE). It uses the marshalling information described in section 6.3.2.3 and invokes BusinessObjectTIE::execute(method_id, in, out).

2. Inside this method the BusinessObjectTIE object performs some transaction-specific code and then asks the resource factory to create a new resource by invoking I_ResourceFactory::createResource(...).

3. Now, the client request is being forwarded again. This time, the resource object continues the de-marshalling in BOresource::execute(method_id, in, out). The actual implementation is to be found in AccountResource::execute (see code snippet above).

4. Finally, the marshalling code forwards the request to the method implementation.

Why is the de-marshalling so complicated? Why is it necessary to forward a request to a resource object, and why does not the BusinessObjectTIE instance implement the business logic? The design could have been different, of course. The following ideas lead to this way of integrating business logic into the framework:

1. For distributed transactions, we need the concept of a resource that the transaction service communicates with during transaction termination (see next section).

2. The resource hosts the transactional cache (a copy of the business object's persistent state). The business logic has to be able to access this cache easily.

3. In order to minimise the code generation effort, the business logic should be placed inside a class that implements not much more than the business logic, but rather inherits functionality provided by the framework. This functionality includes automatic persistence, initialisation and state management.
4. Type-specific resource objects can easily be integrated by means of a *generic factory* \((\text{I\_ResourceFactory})\). Each call to the generic factory interface will result in the creation of a type-specific resource \((\text{e.g. AccountResource})\).

5. It would be even nicer to have a separate object that processes the business logic and does not inherit from any framework class. But this would be a third programming language object involved in request processing, and since every object instantiation takes a considerable amount of time, the decision fell towards performance and against better design.

### 6.3.3.2 **BOresource Interface**

The **BOresource** interface inherits the **CosTransactions::Resource** interface, implements the mutators (state-modifying methods) of the **BusinessObject** interface and implements some additional methods that are required for internal purposes. Table 6-2 gives an overview over all methods.

<table>
<thead>
<tr>
<th>Method name</th>
<th>Interface / etc.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prepare</td>
<td><strong>CosTransactions::Resource</strong></td>
<td>prepare phase of 2PC</td>
</tr>
<tr>
<td>commit</td>
<td><strong>CosTransactions::Resource</strong></td>
<td>commit phase of 2PC</td>
</tr>
<tr>
<td>commit_one_phase</td>
<td><strong>CosTransactions::Resource</strong></td>
<td>commit phase of 2PC</td>
</tr>
<tr>
<td>rollback</td>
<td><strong>CosTransactions::Resource</strong></td>
<td>rollback of OTS transaction</td>
</tr>
<tr>
<td>forget</td>
<td><strong>CosTransactions::Resource</strong></td>
<td>delete resource and forget transactional cache</td>
</tr>
<tr>
<td>add_to_relationship</td>
<td><strong>bof::BusinessObject</strong></td>
<td>associate another business object in a relationship</td>
</tr>
<tr>
<td>remove_from_relationship</td>
<td><strong>bof::BusinessObject</strong></td>
<td>remove another business object from a relationship</td>
</tr>
<tr>
<td>delete</td>
<td><strong>bof::BusinessObject</strong></td>
<td>delete business object from database</td>
</tr>
<tr>
<td>__execute</td>
<td><em>public abstract</em></td>
<td>skeleton method</td>
</tr>
<tr>
<td>create_object</td>
<td><em>public</em></td>
<td>create new <strong>BusinessObjectTIE</strong></td>
</tr>
<tr>
<td>coordinator</td>
<td><em>public</em></td>
<td>'get' method for <strong>Coordinator</strong> attribute</td>
</tr>
<tr>
<td>init</td>
<td><em>protected abstract</em></td>
<td>initialising persistent state attributes</td>
</tr>
<tr>
<td>unlockState</td>
<td><em>private</em></td>
<td>in <strong>BusinessObjectTIE</strong></td>
</tr>
</tbody>
</table>
Implementation

<table>
<thead>
<tr>
<th>lockState</th>
<th>private</th>
<th>acquiring lock for shared state in BusinessObjectTIE</th>
</tr>
</thead>
<tbody>
<tr>
<td>removeResource</td>
<td>private</td>
<td>releasing lock on shared state, deactivate resource object with BOA, and unregister resource object with BusinessObjectTIE</td>
</tr>
</tbody>
</table>

Table 6-2. BResource interface.

The Resource interface defines methods to be invoked during the two-phase commitment protocol used by the transaction service. Its main methods are commit(), prepare() and rollback() (see Figure 6-4). commit_one_phase() is invoked if this is the only resource participating in the transaction.

Each resource is registered with the transaction co-ordinator when it is created (as a result of a transactional call) and thereby associated with a single top-level transaction. In the commit phase every participating resource is called by the transaction service first to prepare and then to commit the changes made to the business object’s persistent state.

The three methods of the BusinessObject interface modify persistent state and are therefore implemented in the resource object, only modifying the state in a transactional cache. When invocations of these methods arrive at the BusinessObjectTIE, they are directly delegated to the appropriate resource object. Almost the same applies to the create_object method, which is part of the TypeManager interface, except for the fact that the parameter list is slightly extended.

The __execute method implements the main part of the invocation marshalling and has to be public in order for the BusinessObjectTIE to be able to delegate the invocation to it. Most of the other methods are declared private and for internal use only.
6.3.3.3 The Role of the Resource Object during an OTS Transaction

Figure 6-5 shows the interaction of the main participants during transactional invocations. One can clearly see that the Resource object plays the central role in the whole framework. In the example shown below, the client starts an OTS transaction, asks the TypeManager to create a new BusinessObject, then issues a transactional method that modifies persistent state, and finally asks the transaction service to commit the changes.

1. The client invokes `CosTransactions::Current.begin()` to start an OTS-managed transaction. All the following method invocations will transparently carry the associated transaction context.

Figure 6-5. The process of an OTS transaction.
2. a) The client asks the TypeManager to create a new business object by invoking `bof::TypeManager.create_object(init_values)`.

b) The TypeManager object recognises the invocation as a transactional invocation and therefore creates a new resource object (e.g., `AccountResource`), which processes the call. If the invocation was not transactional, it would not need a resource object but would rather create the `BusinessObjectTIE` itself in a local transaction (or no transaction at all).

c) The TypeManager registers the new Resource with the transaction service (i.e. the transactional co-ordinator).

d) The TypeManager delegates the client call to the resource object.

e) The resource object starts a local transaction against the database and asks for an object key (OID).

f) The resource object creates a new `BusinessObjectTIE` instance with the OID provided by the datastore and initial values provided by the client. It then initialises its transactional cache and registers with the `BusinessObjectTIE`.

g) Finally, it registers the `BusinessObjectTIE` with the type manager and the BOA. The reference to the `BusinessObjectTIE` is then returned to the client.

3. a) The client's next method is an invocation on the business object (e.g., `Account::credit()`). This invocation also carries the transaction context.

b) According to the transaction ID given in the transaction context, the `BusinessObjectTIE` chooses the appropriate `BOresource` and delegates the client call to this resource object. Note that in this situation there is only one resource object associated with the `BusinessObjectTIE`, so there is not really much to choose from. The resource object processes the invoked business logic and modifies its transactional cache.
4. The client may invoke some more methods on the same business object, resulting in the same resource object processing the requests. It may also invoke some methods to other business objects in other application servers, all participating in the OTS transaction. Finally, the client invokes `CosTransactions::Current.commit()` on the transaction service.

5. a) The transaction service maintains a list of all the `Resource` objects and other XA resources that have participated in this transaction. It now starts the two-phase commit protocol, invoking `Resource::prepare()` on each resource in the list.

b) By default, the resource uses optimistic locking to allow concurrent write access. The current implementation does not allow for pessimistic locking, but it could easily be modified to do so. First, it acquires an exclusive lock on the shared cache in `BusinessObjectTIE` and compares the shared cache with a copy that has been taken at the time the transaction started, or the time the resource object has been created, respectively. If they are not equal, this means that some other client has changed the data in the meantime. As a result, the whole transaction has to be rolled back in order to avoid inconsistencies. This is the drawback of optimistic locking.

Alternatively, some databases simply compare timestamps of the last access to table rows. This is easier than fetching a whole copy of the row at the beginning of the transaction and comparing all the values afterwards, but it does not take into account that the row may have changed twice, finally returning to the original state, hence not causing any inconsistencies. Comparing the actual state instead of timestamps may avoid unnecessary rollbacks.

This way of locking persistent state assumes that in the meantime nobody else has changed the data in the database itself. If this should be allowed, the database state has to be verified instead of the shared cache.

Figure 6-6 shows the three different copies of the persistent data. The resource object
maintains two copies: one is the transactional cache, the other is needed for the implementation of optimistic locking.

![Diagram of Persistent state in database, business object, and resource.](image)

Figure 6-6. Persistent state in database, business object, and resource.

c) The next step is for the resource to update the database and invoke a prepare call on it. Therefore, the datastore component has to be able to take part in the two-phase commit protocol. An extension to JDBC 2.0 offers this capability.

6. a) If all resources have agreed to commit, the transaction service invokes `Resource::commit()` on each.

b) The resource object asks the local database to commit the changes.

c) Finally, the resource object unregisters with the `BusinessObjectTIE` and deletes itself. The transaction is successfully completed.

### 6.3.3.4 Transaction Termination

The persistent state of a business object will not be touched until the end of the transaction. The characteristic of optimistic locking is not to lock the shared state until transaction termination but instead to work on a copy and update the state only at the very end.

Transaction termination starts with the `prepare` call invoked on the `Resource` by the `Transaction Service`. If all resources that belong to the current transaction vote for commit, i.e. they promise to lock the persistent state and commit the changes afterwards, then the `Transaction Service` will invoke the `commit` command on the resource object. If there is only
one resource participating in the transaction, the Transaction Service will not invoke prepare, but commit_one_phase instead.

During the prepare call, first the initial state of the transactional cache is compared to the current state. If there is no difference, no changes were made and a database update is not necessary. The resource will return VoteReadOnly, which tells the Transaction Service that no persistent data associated with the resource has been modified by the transaction and that a subsequent commit call is not necessary. The resource will then forget all knowledge of the transaction and delete itself.

If any changes have to be made persistent, the shared state of the business object first has to be locked exclusively. For the purpose of consistency no other instance is allowed to read or update the object's shared state during transaction termination. Then the resource verifies that the shared state has not been modified in the meantime by any other resource. If this is the case, VoteRollback is returned, the resource will delete itself, and the whole transaction will be rolled back. Often, transactional resources simply compare timestamps instead of the data itself.

In the normal case the resource now begins to start a transaction against the local database, updates the persistent data, and invokes prepare on the local database itself. If the database cannot be prepared for any reason, the transaction will be rolled back. Otherwise, the resource returns VoteCommit to the Transaction Service and waits for the commit call. During the commit phase the local database is asked to commit the changes, the shared state of the business object is updated, all locks are released, and the resource removes itself.
Special Case: Deleting a Business Object

If a business object is to be deleted as a result of the transaction termination, all the other active resources of this business object have to be taken into consideration. It would not be
very “nice” for the other resources to find the object deleted when they are trying to prepare and commit their transaction. Besides, it is no problem for the current resource to wait with the deletion of the object until the other resources have finished their tasks. Of course, a flag has to be set that indicates the forthcoming deletion, so that no other transaction will start working with it before the deletion has been confirmed (committed) or undone (rolled back).

6.3.3.5 Future Extensions

The B0resource class is currently missing an important functionality that is absolutely necessary in a production environment. It is not recoverable, i.e. it does not survive a server crash. Therefore it needs to persistently log information about running transactions and use the OTS RecoveryCoordinator to recover and finish the transaction after the server restart.

6.3.4 Business Object

Being the main subject throughout this thesis, business objects are also a central component of the business object framework. However, the implementation provided by the BusinessObjectTIE class mainly delegates client calls to a resource object. It serves as a “tie” between the object adapter and the object implementation, i.e. the resource object. Figure 6-8 shows the business object tie and its relationship to other classes. The following paragraphs outline these relationships:

1. The type manager keeps a cache of all active business objects. This cache is implemented as a hash map (B0table). A type manager manages many business objects, but a business object only belongs to one type manager.

2. The business object keeps its persistent state in a hash table called B0state. B0state provides synchronised and comfortable access methods to the hash map. The state data is shared among all clients.
3. BusinessObjectTIE is derived directly from the abstract Skeleton class and implements the necessary skeleton code itself. The skeleton code is optimised for invocation delegation to associated resource objects.

4. BusinessObjectTIE is derived directly from the abstract Skeleton class and implements the necessary skeleton code itself. The skeleton code is optimised for invocation delegation to associated resource objects.

5. The interface I_WithTimeStamp is used in collaboration with the garbage collection component. The only method get_TimeStamp returns the time stamp of the last invocation on the object.
6. The interface BusinessObject is the CORBA interface the specific business domain class (e.g. Account) is derived from. Its methods provide information, modify relationships, or delete the object. The interface TransactionalObject is part of the CosTransactions module specified in the OTS specification and is only an identifier that enables the objects of this type to transparently receive the transaction context during a transactional method call.

7. The configuration object provides the generic business object class with type-specific marshalling information, information about transaction types and access modes.

8. The BusinessObjectTIE needs two lock variables to lock access to its shared state during transaction commitment and to lock the whole object in case it is in the state of being deleted (details about these locks follow in section 6.3.4.4).

9. Two more attributes, locked_for_remove and _lock_state, are not of a basic type, but a Java class type, because they have to be synchronised during simultaneous access.

10. An indefinite number of resource objects are held in a synchronised vector (SyncVector), that provides some thread-controlling methods. These methods are used during deletion of business objects.

6.3.4.1 BusinessObjectTIE Interface

The BusinessObjectTIE class implements the bof::BusinessObject interface, deals with transaction and resource management, and delegates incoming requests to domain-specific methods (e.g. Account::credit).

<table>
<thead>
<tr>
<th>Method name</th>
<th>Interface / etc.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_PersistentID</td>
<td>bof::BusinessObject</td>
<td>provides the “persistent identity”, containing type name and an opaque ID</td>
</tr>
<tr>
<td>is_identical</td>
<td>bof::BusinessObject</td>
<td>verifies whether a given business object reference points to the same instance</td>
</tr>
<tr>
<td>get_TypeManager</td>
<td>bof::BusinessObject</td>
<td>returns the type manager for this business object</td>
</tr>
</tbody>
</table>

6-28
<table>
<thead>
<tr>
<th>Method</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_to_relationship</td>
<td>bof::BusinessObject</td>
<td>see section 6.3.4.5</td>
</tr>
<tr>
<td>remove_from_relationship</td>
<td>bof::BusinessObject</td>
<td>see section 6.3.4.5</td>
</tr>
<tr>
<td>will_notify_for_relationship</td>
<td>bof::BusinessObject</td>
<td>see section 6.3.4.5</td>
</tr>
<tr>
<td>delete</td>
<td>bof::BusinessObject</td>
<td>deletes the business object from the database</td>
</tr>
<tr>
<td>_ids</td>
<td>marshallng</td>
<td>delegates to ConfigurationImpl (see section 6.3.2.3)</td>
</tr>
<tr>
<td>_methods</td>
<td>marshallng</td>
<td>delegates to ConfigurationImpl (see section 6.3.2.3)</td>
</tr>
<tr>
<td>_execute</td>
<td>marshallng</td>
<td>de-marshalling (see section 6.3.4.2)</td>
</tr>
<tr>
<td>_Domain_execute</td>
<td>marshallng</td>
<td>de-marshalling (see section 6.3.4.2)</td>
</tr>
<tr>
<td>_BusinessObject_execute</td>
<td>marshallng</td>
<td>de-marshalling (see section 6.3.4.2)</td>
</tr>
<tr>
<td>get_TimeStamp</td>
<td>I_WithTimeStamp</td>
<td>return timestamp of last invocation</td>
</tr>
<tr>
<td>ping</td>
<td>public</td>
<td>renew timestamp</td>
</tr>
<tr>
<td>prepare_remove</td>
<td>public</td>
<td>two methods used if object is going to be deleted during transaction commit</td>
</tr>
<tr>
<td>undo_prepare_remove</td>
<td>public</td>
<td>undo 'prepare_remove' in case of a rollback</td>
</tr>
<tr>
<td>delete_related_first</td>
<td>public</td>
<td>deep deletion of aggregated objects</td>
</tr>
<tr>
<td>current</td>
<td>private</td>
<td>get TransactionCurrent</td>
</tr>
<tr>
<td>resource</td>
<td>private</td>
<td>get the resource associated with the current transaction</td>
</tr>
<tr>
<td>create_new_resource</td>
<td>private</td>
<td>as it says</td>
</tr>
<tr>
<td>resources</td>
<td>public</td>
<td>return vector of resources</td>
</tr>
<tr>
<td>state</td>
<td>public</td>
<td>returns a reference to the shared state</td>
</tr>
<tr>
<td>lock</td>
<td>public</td>
<td>returns the _lock_state object</td>
</tr>
<tr>
<td>set_values</td>
<td>public</td>
<td>initialise state attributes</td>
</tr>
<tr>
<td>repository_id</td>
<td>public</td>
<td>returns the object's repository ID</td>
</tr>
<tr>
<td>reference_data</td>
<td>public</td>
<td>returns the object's reference data (OID)</td>
</tr>
</tbody>
</table>

Table 6-3. BusinessObjectTIE interface.

The seven methods that implement the BusinessObject interface conform to the OMG Business Objects Interoperability Specification\(^1\). They represent the minimal functionality needed for a working Business Object Component Architecture (BOCA).

\(^1\) [BUS98b]
The following five methods are used during de-marshalling of client invocations. Domain-specific (user-defined) methods are separated from the mandatory BusinessObject methods. The process of de-marshalling requests is explained in detail in the next section. The other methods deal with resources, transactions, locks, state initialisation and life cycle.

6.3.4.2 De-Marshalling Requests

Figure 6-9 illustrates the important steps during the de-marshalling process. Little black balls mark the method invocations from one object to another.

1. During de-marshalling, the client request is first forwarded to the object implementation by the Basic Object Adapter (BOA). The object instance is chosen on behalf of the object key which is imbedded in the object reference.
2. The two methods _ids() and _methods() help the object adapter determining the identifier of the method to be invoked. These two methods are implemented in and delegated from the business object to the ConfigurationImpl instance.

3. The method identifier, together with a stream of input parameters, is then passed to the method _execute() of the BusinessObjectTIE instance. Before the invocation is delegated to the implementations in BusinessObjectTIE, BOresource and Account, transaction policies and isolation levels are determined.

6.3.4.3 Transaction Policies

The _execute() method is split in three separate methods which sum up to more than 500 lines of code. The whole de-marshalling process represents half of the overall logic in the BusinessObjectTIE class. This is because there are a lot of different cases to be considered. First, the execute method differs between mutator methods and read-only methods of the BusinessObject interface. Read-only methods can be executed without most of the locks and without all the transaction management. Second, incoming calls may carry a transaction context and therefore take part in a distributed transaction. In case the client request does not carry a transaction context, there are four different types of transaction policies, each of which can be applied to any domain-specific (user-defined) method and the mutator methods of the BusinessObject interface. Figure 6-10 illustrates the first few steps of de-marshalling. The four different transaction policies are explained below.

The _execute() method first acquires a lock on the business object to ensure that the object is not accidentally deleted during the de-marshalling process. In case the object is just in the process of being deleted from the database, the request is rejected and an exception is sent back to the client indicating that this object does not exist.
As long as this method invocation lasts, it is not possible to delete the object. On the other side, if in a transaction the object is marked for deletion, and the OTS has already called prepareO, the thread will have to wait for this lock to be released. In the following step, locked_for_remove will then be true.

This lock is different to the one above. The object may have been marked for deletion as a result of a commit() call. The actual deletion in the database has already taken place or is in process.

Figure 6-10. Activity diagram: Locks and transaction policy decisions before the execution process.
If the client call carries a transaction context, the business logic is processed as part of this transaction, otherwise it is processed according to the transaction policy the business object developer has defined for the method:\footnote{1}

1. **tx\_distributed.** The method call has to be processed in an OTS transaction, because subsequent calls from this business object to others are possible and the transaction context has to be forwarded to them. Therefore a new OTS transaction has to be started prior to the processing of the business logic.

2. **tx\_distr\_only.** The same as above, except that the clients themselves are obliged to start the OTS transaction. If the client misses to start the OTS transaction before invoking the call to the server object, an exception is thrown to indicate the requirement for a transaction context. Bank accounts, for example, may only be invoked in a distributed transaction, thus ensuring consistency among all bank accounts.

3. **tx\_local.** If the client did not send a transaction context, it is possible to process the method call in a local transaction against the local database. This is much faster and does not involve the transaction service.

4. **tx\_none.** If the client did not send a transaction context, it is possible to process the method call without any transaction at all. This is not safe with regard to consistency, but very efficient.

These four transaction policies are roughly the same as the policies for Enterprise JavaBeans. Figure 6-11 illustrates the subsequent steps of the execution process. The following paragraphs explain the differences between them.

\footnote{1 See also the transaction policy table in section 6.3.2.2.}
Figure 6-11. Activity diagram: Execution process of four different transaction policies.

Take part in an existing OTS transaction

According to Figure 6-11, taking part in an existing OTS transaction seems to involve fewer steps than processing the business logic without any transaction mechanism at all. This is
Implementation

simply because the locking and database access is done during the prepare call invoked on the resource by the transaction service (see section 6.3.3.4).

If the method call is not the first call to the business object in this transaction context, an appropriate Resource object already exists and the call will be forwarded to this object. This assures that all method calls on the business object work on the same transactional cache.

**Start new OTS transaction**

This process is similar to the previous, except for the fact that the client did not start the OTS transaction itself. According to the transaction policy the business object developer has determined for the business object (i.e. tx_distributed), the business logic may only be executed in the context of an OTS transaction. Therefore the business object will first start a new transaction and afterwards commit (or in the case of an exception, rollback) it.

**Start local transaction**

The business object developer may also allow business logic to be processed without the OTS being involved. In this setting, the business object framework will start a local transaction against the local database before and commit it after the business logic is executed. This is faster, as long as the client does not invoke several server methods of the same business object after another.

Each method invocation causes a separate resource object to be created, which executes the business logic, and which is deleted afterwards. This produces a huge overhead if many invocations follow after another. Hence, it would be easier to start an OTS transaction and to commit it after all invocations have finished. The invocations would then work on the same transactional cache with only one resource object involved.

On the other side, optimistic locking is not applied and the shared cache has to be locked during the whole operation to ensure consistency. Technically, optimistic locking could be applied, but in case a rollback is necessary, the user could only be notified by throwing a
system exception. This usually confuses a client program more than it can handle by its exception handling. Besides, the business object framework is not built for batch processing, where a client invokes a large number of unrelated requests, but instead it deals with online transaction processing (OLTP) and distributed objects. Therefore, the design decision fell towards an efficient way of dealing with single, non-distributed, but reliable requests.

**Process business logic without any transaction mechanism**

Even faster than the way mentioned above, this policy is suitable for unreliable requests or read-only requests in general. It is not suitable for methods that process complicated business logic, since the shared cache is locked during the whole operation. However, this locking is only necessary, if more than one persistent attribute is requested (otherwise inconsistencies might happen due to simultaneous access).

There are eighteen different combinations of transaction policy, access mode and locking mechanism possible. Table 6-4 shows all combinations. Recommended settings are indicated with a plus sign in the fourth column. In the current version of BOF, distributed transactions are implemented with optimistic locking only.

<table>
<thead>
<tr>
<th>Access Mode</th>
<th>Transactions</th>
<th>Locking</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>read-only</td>
<td>none</td>
<td>none</td>
<td>(2) very fast, but may cause consistency problems</td>
</tr>
<tr>
<td>read-only</td>
<td>none</td>
<td>pessimistic</td>
<td>+ fast and reliable</td>
</tr>
<tr>
<td>read-only</td>
<td>none</td>
<td>optimistic</td>
<td>o unnecessary overhead, if business logic is simple</td>
</tr>
<tr>
<td>read-only</td>
<td>local</td>
<td>none</td>
<td>- makes no sense</td>
</tr>
<tr>
<td>read-only</td>
<td>local</td>
<td>pessimistic</td>
<td>- makes no sense</td>
</tr>
<tr>
<td>read-only</td>
<td>local</td>
<td>optimistic</td>
<td>- makes no sense</td>
</tr>
<tr>
<td>read-only</td>
<td>distributed</td>
<td>none</td>
<td>- makes no sense</td>
</tr>
<tr>
<td>read-only</td>
<td>distributed</td>
<td>pessimistic</td>
<td>- makes no sense</td>
</tr>
<tr>
<td>read-only</td>
<td>distributed</td>
<td>optimistic</td>
<td>- makes no sense</td>
</tr>
<tr>
<td>read/write</td>
<td>none</td>
<td>none</td>
<td>- not allowed, inconsistency prone</td>
</tr>
<tr>
<td>read/write</td>
<td>none</td>
<td>pessimistic</td>
<td>o very fast, but not reliable (3)</td>
</tr>
</tbody>
</table>
Implementation

<table>
<thead>
<tr>
<th>Access Mode</th>
<th>Transaction Policy</th>
<th>Locking Mechanism</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>read/write</td>
<td>none</td>
<td>optimistic</td>
<td>very fast, but not reliable (3)</td>
</tr>
<tr>
<td>read/write</td>
<td>local</td>
<td>none</td>
<td>not allowed, inconsistency prone</td>
</tr>
<tr>
<td>read/write</td>
<td>local</td>
<td>pessimistic</td>
<td>fast for single, unrelated requests; slow for multiple related requests (4)</td>
</tr>
<tr>
<td>read/write</td>
<td>local</td>
<td>optimistic</td>
<td>fast for single, unrelated requests; slow for multiple related requests (4)</td>
</tr>
<tr>
<td>read/write</td>
<td>distributed</td>
<td>none</td>
<td>not allowed, inconsistency prone</td>
</tr>
<tr>
<td>read/write</td>
<td>distributed</td>
<td>pessimistic</td>
<td>fast for multiple related requests in one single tx; slow for unrelated requests (6)</td>
</tr>
<tr>
<td>read/write</td>
<td>distributed</td>
<td>optimistic</td>
<td>fast for multiple related requests in one single tx; slow for unrelated requests (6)</td>
</tr>
</tbody>
</table>

(1) Rating: + recommended; - not recommendable; o possible, but with drawbacks.
(2) No locking means no transactional cache. Business logic operates on the shared cache.
(3) This may be enough for simple web applications, but not for applications that require reliability, e.g., e-commerce.
(4) The decision for optimistic or pessimistic locking depends on the application architecture, on the nature of the business object, and on the number of clients accessing the business object concurrently.

Table 6-4. Settings for access mode, transaction policy and locking mechanism of business object methods.

Exceptions during Processing of Business Logic

The business object developer can define his own exception types. If an error occurs during processing of the business logic, a user-defined exception is thrown and sent back to the client. This does not lead to a roll-back, but instead it is the client’s responsibility to react on the error and roll back the transaction where appropriate.

6.3.4.4 Locks

The BusinessObjectTIE class has two locks, _lock_state and tx_lock_for_remove, that manage concurrent access to the shared state and to the whole object, respectively. It is not very efficient to lock the whole object each time its state is read or modified. Besides, locking a whole object increases the danger of running into a deadlock situation. Therefore, locks should be applied to the smallest possible granularity and for as short a time as possible. Furthermore, a Boolean flag (locked_for_remove) complements the locks to indicate a certain condition without having a thread waiting for the outcome of the operation that set
the flag. In some situations, access to this flag also has to be synchronised. The following paragraphs explain when to apply the locks mentioned above:

- During the creation of a new resource object the resource acquires a lock on the shared cache (_lock_state) in order to copy a consistent state of the cache into its transactional (unshared) cache.

- Each time a resource updates the shared state cache with the outcome of a transaction the cache has to be locked (_lock_state). This is necessary to ensure a consistent view on the shared state at all times.

- Before entering the skeleton code the tx_lock_for_remove lock has to be acquired. If the lock is already taken by another thread, this indicates that a resource is going to delete the business object from the database and the main memory and it is just waiting for all other resources of this business object to finish their transactions and for the transaction service to order the commit of the transaction. In this case the thread, which requests this lock, waits for the outcome of the deletion process. Most probably, the business object will be removed, the waiting thread will be notified and an OBJECT_NOT_EXIST exception is thrown. If the deletion is rolled back, the business object will not be deleted, the waiting thread will be notified, it releases the lock and continues as usual. The process of deleting a business object is described in detail in the next section.

- As described above, a resource object acquires a lock on tx_lock_for_remove to prepare for deleting the business object. If the transaction, during which the lock has been acquired, is rolled back, the lock is released.

The two thread locks do not affect each other. Each of them locks only a part of the business object and mostly this locking still allows for concurrent operations. There are six more situations when access to the whole object or to parts of it (list of resources, shared state, timestamp, locked_for_remove flag) has to be synchronised.
6.3.4.5  BusinessObject::delete()

Deleting a business object is not a simple task in a multi-threaded environment. As long as other resources work on the object it cannot be deleted. To be technically precise, it could be deleted, but then all active resources would have to roll back their transactions. However, in most cases it is better to have one resource waiting for a short time, instead of having a number of other resources rolling back their transactions.

Furthermore, there is a short time of uncertainty, between the prepare and commit call to the resource, which wants to delete the object. As long as the according transaction is not committed, the object cannot be deleted, but each new method request to the business object has to wait for the commitment. Usually, the time between prepare and commit is very short, so that waiting client requests are not delayed too long.

The following example illustrates the main aspects of the deletion process. The business object is deleted in the scope of a distributed transaction (transaction policies tx_distributed or tx_distr_only). The invocation BusinessObject::delete() is forwarded to the appropriate resource. The resource just sets its private flag _marked_delete to true and returns. The actual process of deleting the object starts with the transaction service's prepare() call to the Resource. This is where the activity diagram in Figure 6-12 starts. The prepare() call locks the business object so that no other client request can entry the object until transaction termination. Deleting the persistent state in the database and removing the business object from the type manager's and the Basic Object Adapter's list of active CORBA objects occurs during the commit() operation.
Figure 6-12. Activity diagram: Deleting a business object during transaction completion.
6.3.4.6 Relationships

The business object interface has three generic methods that deal with relationships. Relationships are basically references to other business objects. A relationship has a name and one or more members, depending on its cardinality. An application can add or remove members, and can ask if the relationship is bi-directional and managed, so that the associated objects are notified of any changes. For example, if a customer object is deleted from a database, its associated accounts are removed as well.

```java
interface BusinessObject : CosTransactions::TransactionalObject {
    // ...
    void add_to_relationship(in string rel_name, in BusinessObject member);
    void remove_from_relationship(in string rel_name, in BusinessObject member);
    boolean will_notify_for_relationship(in string relationship_name);
}
```

Relationships are not implemented in the prototype. The recommended way to implement this functionality is by using smart pointer objects, which hide activation and deactivation of the associated objects (see chapter three, section 3.4.5). The associated objects are very likely to exist remotely on another machine, therefore binding and unbinding has also to be done in the smart pointer class.

As already explained in chapter two, section 2.2.4, associated objects are not to be mixed up with composite or dependent objects. In general, business objects have an independent identity and lifecycle, and are referred by multiple clients. A composite object should not be modelled as a business object. Instead, a composite object is better implemented as a Java class (or several classes) and is included as part of the business object on which it depends.

"A dependent object [i.e. composite] can be characterised as follows. An object B is a dependent object of an object A, if B is created by A, accessed only by A, and"
removed by A. This implies, for example, that if B exists when A is being removed, B is automatically removed as well. It also implies that other programs can access the object B only indirectly through object A. In other words, the object A fully manages the life cycle of object B." [SUN99]

For example, a purchase order might be implemented as a business object, but the individual line items on the purchase should be implemented as helper classes, not as business objects with their own skeleton and all the overhead that comes with concurrency and transaction management. An employee record might be implemented as a business object, but the employee address and phone number should be implemented as helper classes, not as business objects.

The state of a business object that has composite objects is often stored in multiple database records and spans multiple tables. In addition, the business object developer must take into consideration that every method call to a business object is potentially a remote method call and that the resulting overhead is too high for most fine-grained object interactions.

**6.3.4.7 Future Extensions**

In order to increase scalability, the business object containers should be replicated. This causes a problem regarding the unique identity of a business object. The container process can be replicated, but not the business objects themselves, as long as they leverage cross-transaction caching. One solution would be to cut the range of values of object identifiers into a number of separate ranges according to the High/Low approach outlined in chapter 3 (section 3.3.2.3). Each running business object container, i.e. each single type manager, would be given a separate range to work on. This would ensure that a business object exists only in one process. The domain manager could serve as a proxy for client requests and forward the requests to the appropriate container.
The persistence of business objects is currently managed transparently by the container. In some situations it might be more effective and easy enough to have the business object developer managing the persistence himself. Therefore the datastore component should offer an easy-to-use API to manually access the database.

The transaction management could be extended by two more features: First, the OMG Object Transaction Service provides nested transactions, which could be implemented by the container. Second, pessimistic locking could be implemented as the second locking mechanism, which allows to sequentialise transactions. With pessimistic locking, a lot of overhead can be avoided by reusing the resource object.

The skeleton code in the present implementation is implemented in a generic way, where possible. Using a comfortable tool for code generation, more type-specific skeleton code could be generated, which would speed up the marshalling process.

Finally, an event mechanism could add value to this container implementation. Not only notification of changes in relationships, but also user-defined events could be implemented using the OMG Event Service.

6.3.5 State management

Business objects require an efficient and intelligent way of dealing with their life cycle. Not all persistent objects can be active in main memory at the same time. Transparent activation and deactivation (garbage collection) have to be applied. Sections 3.1.9.4 and 3.1.9.5 explain the details. Figure 6-13 shows the class diagram according to the implementation.
6.3.5.1 Activation

The activation mechanism is straightforward: The object adapter receives a request for an object, whose object id is not found in the current table of active objects. The object adapter forwards the request to the activator object (BOFactivator), which finds the appropriate type manager instance and invokes resolve_opaque_id(oid) on it. By means of the object identifier the persistent object can be located in the database and instantiated in main memory. The client request is now forwarded to the transparently activated business object instance. The type manager keeps its own table of active objects. Unfortunately, with the Basic Object Adapter (BOA) it is not possible to share the adapter’s active object table; this will change with the introduction of the Portable Object Adapter (POA).

6.3.5.2 Deactivation

The deactivation mechanism is a little bit more complicated. Deactivation does not exactly mean "garbage collection", although this term is used in the same context. By this mechanism, business objects are not persistently deleted, but only deactivated and removed.
Implementation

from the main memory. A garbage collection mechanism simply keeps track of all references to an object and deletes the object if the reference counter decreases to zero. This is not possible, because there is no reference counting mechanism for distributed references. The server will never know, when a client has freed his stub or if the client will ever use the object again after the first invocation. The server does not even know, how many clients hold references on the business object, because CORBA references can be copied from client to client without the server knowing about it.

Section 3.1.9.5 discusses several strategies of how and when to decide for deactivating an object. The current implementation uses a simple approach that is sufficient in most cases, but that can be complemented by an additional garbage collector, which has to be configured in the configuration file. This approach uses time stamps on business objects to determine when an object has last been "touched" or requested by a client. Almost each method invocation on a business object results in its time stamp to be updated. Any garbage collection mechanism can utilise the time stamp to decide which objects have to be removed. This ensures that frequently used objects stay alive, while rarely used objects are freed first. The type manager can be configured to limit the number of active objects to a maximum. If the maximum is reached, a certain percentage of objects (30% by default) is deactivated according to their time stamps.

6.3.5.3 Future Extensions

In some situations a transaction-level activation policy might be more efficient than the state management mechanisms described above. If the life cycle of business objects is bound to the scope of transactions, the objects are only instantiated when a transaction starts and

---

1 There is some confusion about the matter of reference counting in CORBA. Newsgroups and newspapers are full of misleading statements about this issue. There is actually a reference counting mechanism in C++ ORBs, but it only counts local references and not at all distributed references, which leave the process to any client.
removed when it terminates. Many locking and concurrency problems disappear in this model. Unfortunately, the advantages of cross-transaction caching disappear as well. On the other side, this transaction policy helps to implement replication of business object containers for scalability purposes, because only exclusive locking is applied on database level.

6.3.6 Persistent State and Types

In order to support transparent persistency and at the same time allow an easy way of defining and implementing business objects a mini-framework is required. The business object developer should only have to know the names and types of the business object's attributes, and not to invoke any methods that manage their persistency.

6.3.6.1 Persistent Type Hierarchy

Figure 6-14 shows the classes that implement persistent attributes. All attributes are kept in a hash map. When a resource (BOresource) object is created, it receives two copies of the shared state (invoking copy_state()), on which it can work until transaction completion. When the transaction is committed, it updates the shared state invoking update_state(). Before that, the resource compares the shared cache with its transactional cache using is_equivalent().

There are ten different types available, which specialise the abstract base class PtypeBase. The DataStore component will only work on this generic base type in order to update the persistent attributes. PtypeBase allows to request type information, which is necessary for exchanging data between the database and the business object.
6.3.6.2 Access to Persistent Attributes

Inside the business object component (e.g. AccountResource) the attributes are made accessible as instance attributes like

```java
Pfloat _amount;
```

which are initialised by references to the according objects in the BOstate hash map:

```java
protected void init()
{
    _amount = (Pfloat)_currentState.get("amount");
}
```

The business object developer can now access the values of the persistent attributes by invoking the `value()` method on it:

```java
public float balance()
{
    return _amount.value();
}
```
public void debit(float amount)
{
    _amount.value(_amount.value() - amount);
}

6.3.6.3 Future extensions

A better design of the framework should introduce a “clean” distinction between (a) a business object entity, which represents only the persistent state, (b) a business process object, which only processes the business logic, and (c) the business object wrapper, which deals with transaction logic and marshalling. With such a model, it would be much easier to integrate object-oriented data sources like object-relational mappers.

6.3.7 Type Manager

A type manager represents all business objects of a type within a business system domain. The type manager object is transactional and persistent, like business objects, and can have attributes, relationships and operations that apply to all members of the type. The type manager provides queries against all members of the type and its sub-types (the type extent), and it manages, directly or indirectly, the creation, activation and deactivation of members of the type.

6.3.7.1 Design

The primary purpose of the type manager is to manage the life cycle of business objects. One type manager instance is restricted to one single business domain type, e.g. Account or Customer. It also uses one specific datastore component which has been created for this business object type. Figure 6-15 shows the type manager and its relationships to other classes.
As it implements the TransactionalObject interface the type manager will take part in distributed transactions and automatically receive the associated transaction context. Its public CORBA interface provides methods to request information about type and configuration, factory methods for creating business objects, and methods that will query a database (see IDL description in the appendix).

### 6.3.7.2 Factory Methods

The TypeManager interface contains two factory methods:

- `BusinessObject create_object (in NameValues init_values);`

- `BusinessObject create_from_object (in BusinessObject source);`

The following activity diagram shows the action states and transitions that are performed during a `create_object` call.
First, a **Current** object is requested as a reference to the Transaction Service. If the **Current** object is not available due to a Transaction Service failure, the client will receive an exception telling that the Service is not available. If **Current** is available and it contains a transaction context, i.e. a client has started a distributed transaction which has been associated with the client's thread of control, a **BOresource** object is created and the
operation will continue in this OTS resource object. This enables the creation of the
BusinessObject to be committed (or rolled back) using a two-phase commit protocol
performed by the OTS. If the Current object is available but does not carry a transaction
context (a Control object), the operation is performed as a local transaction against the
database, or the operation is performed without a transaction at all, or an exception is raised,
according to the policy the business object developer has determined during the development
process (see section 6.3.2.2).

6.3.7.3 Future Extensions

The implementation of the type manager has only a generic interface, which applies to any
business object type. Factory and query methods work with the base type BusinessObject.
But it may also be very useful to have type-specific factory and query methods. This will
increase the amount of code generation, but it makes life much easier for the application
developer who uses the interfaces.

Another useful feature is introspection. Introspection is a capability to access information
about objects of a type. This is of particular interest for general-purpose tools for testing and
debugging. The type manager for each type provides access to the available meta data for that
type.

Currently the management of business objects is not done very efficiently. The BOA holds
one table, the type manager a second one. Using the new POA technology, it would be
possible to do without one of them. Also, the creation of object identifiers is much easier
with the POA.

Another valuable feature is monitoring of threads, objects, the garbage collector and the
activator. A central management instance could monitor all active business object containers
and apply intelligent load balancing mechanisms according to the current states of the different processes.

6.3.8 DataStore

The datastore component implements the most simple and generic interface to a persistency layer, as it is described in chapter 3, section 3.4. It encapsulates features like database connection pooling and caching. It can be implemented by means of an object-to-relational mapping tool or by low-level JDBC. It can wrap access to legacy applications in order to leverage existing data sources. The only feature that is still difficult to implement today is the two-phase-commit protocol, because many database systems and especially JDBC drivers still do not support this protocol. Due to this fact, the present datastore component is only a dummy implementation.

6.3.8.1 DataStore Interfaces

![Class diagram: DataStore component and its relationships.](image)

*Figure 6-17. Class diagram: DataStore component and its relationships.*
All methods beside `resolve()` change database state and therefore need a transaction identifier as an input parameter. According to the transaction identifier subsequent invocations to the datastore component can be associated with the transactions they belong to. The datastore component has to log information about these invocations during runtime to ensure recoverability after a system crash. To update persistent state, the `update()` method receives the `BOstate` object as an input parameter. There is no conversion into another format necessary. The `BOstate` object contains all meta-data required to map the persistent attributes to table columns in a relational database. The connection to a specific database is initialised during creation of the datastore component according to the information in the configuration file.

Since the datastore component usually receives multiple requests concurrently, it has to be thread-safe and scalable. For the purpose of scalability it can be externalised and run as a separate component on another machine.

### 6.3.8.2 Future extensions

Using object-relational mapping tools, the developer will get objects instead of result sets from the database layer. To reduce unnecessary overhead it would be good to integrate the database objects into the object model instead of copying and synchronising with `BOstate` objects. A simple but effective solution would be to have the database objects implement the generic `BOstate` interface, so they could be referenced by the `BusinessObjectTIE` instance. The prerequisite for this, that (a) the life cycle management of these database objects, which is usually done by the persistency framework, can co-operate with the life cycle management of the business objects themselves, and that (b) the `DataStore` component offers an interface, which creates a new `BOstate` object.
The present prototype version does not address locking issues on the database level. It assumes that the Business Object Framework is the only instance that modifies the database. Of course, this can not be assured in any case. Therefore the locking mechanisms on the framework level should be extended to the datastore component.

A scalable datastore component would need mechanisms like connection pooling, caching, read-ahead caching, online schema-evolution, and functionality to access legacy data and applications, since in many cases, the underlying data source may be an existing application rather than a database.
7 Conclusion

7.1 Review

This thesis gives a definition of business objects, introduces component models, and discusses the technical issues of persistence, transactions, and concurrency. It compares different types of Application Servers and discusses their ability to host business objects. It presents a possible design of a business object framework, the technical basis for an Object Transaction Manager. The design of the framework includes an extended state management for fine-grained business objects and a solution for concurrent access of multiple transactions. The prototype implementation shows the difficulties of concurrency and their impact on the overall performance. Alternative designs and possible extensions are suggested.

Looking back at the design of the business object framework, it appears that the decision for generic components instead of more type-specific components increases overhead and diminishes flexibility. In this respect the Enterprise JavaBeans (EJB) specification offers a much better solution. The design did also not enough consider the important issue of portability. The business object component contains ORB-specific marshalling code and is therefore dependent on a single middleware vendor. In contrast, EJB are completely decoupled from the container implementation they are finally deployed in.

Despite the fact that the design and the implementation of the present framework has been “overtaken” by the younger EJB specification, it offers a working solution and has been a rich experimental environment to learn from.
7.2 Future Developments

The future will bring portable CORBA components. At this time, portable components are found only in the Windows 98/NT environment as COM objects or in the Java environment as JavaBeans. CORBA components are not portable between different ORB products, unless in combination with Java. A Java binding alone is not sufficient, though. Only support for the new Portable Object Adapter (POA) Specification, which provides a vendor-neutral standard for object persistence and activation, will bring true portability. Thinking of the POA, Sun is even exploring the possibility of portable object containers, which could be a future enhancement of EJB.

The future will bring new software markets. This statement is fairly obvious. There is already a big market for application server products and with the introduction of EJB Sun has created a new market for development tools that specialise in EJB components. Tight integration of object design tools in the development process and the use of object-relational mapping in application server products will be standard features. An open environment based on CORBA and Java standards allows for a market of services and vertical-domain facilities. Business components are packaged in suites for special business domains.

The future will bring „zero latency business“ processes. This term was coined by the Gartner Group to describe businesses that are able to complete a whole business process end-to-end in one stroke, rather than dividing it into queued steps and allowing the entire process to complete over a period of time, sometimes even requiring user intervention. With distributed business objects, an entire business process can be handled as a single software transaction, even if it accesses multiple databases from different vendors, possibly in different geographic locations. This will be made possible by means of a standardised infrastructure for communication and transactions among distributed objects.
The future will bring more "self-service" applications. Completing a whole business process in one atomic transaction does not only speed up customer service, but also enables new applications, which allow customers to initiate and complete business transactions themselves. Simple examples that are already in production are online book shops and reservation/booking services. These applications go hand in hand with e-commerce and electronic money.

7.3 Example Applications for Business Objects

The technology of business objects and application servers provides the foundation for Internet commerce. With enterprise business objects, the Internet will grow from an information and presentation technology to an infrastructure for commercial applications, capable of supporting large-scale transaction processing. These applications include Electronic Data Interchange (EDI), Internet payment systems, security solutions, document management and workflow.

The latest wave of Internet applications is database publishing. These systems feature simple static access to dynamic data. Examples include stock quote checking, or review of purchase order or delivery status via the Internet. These applications will evolve into transactional business applications. Sybase has coined the term "WebOLTP" for applications that move beyond simple viewing to real-time updates of business critical information. Figure 7-1 illustrates the position of WebOLTP. It is taken from a now obsolete Sybase white paper [SYB98].

The next two paragraphs from [SYB98] give examples for WebOLTP applications.

"A large national bookstore chain is designing an Internet-based inventory search and order entry application. Currently, when a customer visiting one of the chain’s locations can’t find a book, a customer service representative can offer to order the book and call the customer when it arrives. With dozens of stores around the country, however, it is likely that the book is in stock somewhere -- meaning it could be sold, with scheduled delivery to the customer’s address. The
Internet-based application currently in design will allow customer service reps to search the chain’s entire inventory, locate the book, tag it as sold, send delivery instructions to the location in which the book resides, and close the sale. The result is a win-win, with the customer knowing the book is on the way and the store ringing up another sale.”

“A large state university is currently designing an application that will simplify and streamline a cumbersome process at campuses around the country. Students will soon be able to register for classes over the Internet. First, of course, they will want to check out where they stand in terms of the requirements for their majors. Then they will be able to choose their courses for the current semester and confirm their enrollment. With that completed, students will move to another screen that lists the books for the course, purchase the ones they want, and arrange to pick them up or have them delivered. If they want to charge the books to their student account, they can do that; or, if necessary, they can arrange to move funds from their bank account to their student account to cover the purchase.”

![Figure 7-1. The future of Internet Applications is WebOLTP.](image)

**7.4 Vision**

It is very difficult to make long-term predictions about the IT industry, but there are a few developments that are very likely to be seen in the future, because they are highly desirable from a technical point of view. We can already see the integration of business object frameworks into database systems; the Oracle Application Server is the first example. This
integration will finally lead to the possibility of accessing any data in the form of a distributed object. Object-relational mapping tools or object-oriented extensions to relational databases have long been trying to solve the problem of the impedance mismatch between objects and relations. Architectures for distributed application processing like CORBA and DCOM have introduced object-oriented interfaces to data, which removes all the hassle about different proprietary database drivers and query technologies. Object and distribution technology together with a component standard for database entities (e.g. EJB Entity Beans) will make it possible to access virtually any data by means of an object-oriented interface. Distributed component technology may even supersede the proprietary object-oriented extensions of relational database systems. This also means that each address or account, each picture or text document, each bill or patients record will have an Internet URL. Security technology becomes even more important.

From the Internet user’s point of view there will be two “impressive” developments taking place. First, the industry will turn service into self-service and force the user to use the Web more extensively. They will sell the idea of self-service as a new level of comfort, concealing the fact that the user will be encumbered with much more responsibility filling in forms and reading legal terms. Life becomes more difficult. Second, the online bank account shown in a browser application changes its value immediately after the user has clicked the button for submitting an order form in another browser window. Impressive, but expensive. As always, a new technology has good and bad effects. However, Enterprise Business Objects will be gratefully embraced by the IT industry.
# A - 1 Literature

<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>Author(s)</td>
<td>Title</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------</td>
<td>-------</td>
</tr>
<tr>
<td>[GAM94]</td>
<td>GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J.:</td>
<td>Design Patterns: Elements of Reusable Object-Oriented Software. Reading, Massachusetts: Addison-Wesley, 1994.</td>
</tr>
<tr>
<td>[GRA93]</td>
<td>GRAY, Jim, REUTER, Andreas</td>
<td>Transaction processing: concepts and techniques.</td>
</tr>
<tr>
<td>[HEU92]</td>
<td>HEUER, Andreas</td>
<td>Objektorientierte Datenbanken: Konzepte, Modelle, Systeme.</td>
</tr>
</tbody>
</table>
Appendix


 Redistribution of this article has been granted by the publisher. For a Free trial subscription to HPCwire, send e-mail to trial@hpcwire.tgc.com.

A-3
<table>
<thead>
<tr>
<th>Reference</th>
<th>Authors</th>
<th>Title</th>
<th>Publisher</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ORF96]</td>
<td>ORFALI, Robert, HARKEY, Dan, EDWARDS, Jeri</td>
<td>The Essential Distributed Objects Survival Guide.</td>
<td>New York; Chichester; Brisbane; Toronto; Singapore: John Wiley &amp; Sons Inc., 1996</td>
<td></td>
</tr>
<tr>
<td>[ORF97]</td>
<td>ORFALI, Robert, HARKEY, Dan</td>
<td>Client/Server Programming with JAVA and CORBA</td>
<td>New York; Chichester; Weinheim; Brisbane; Toronto; Singapore: John Wiley &amp; Sons Inc., 1997</td>
<td></td>
</tr>
<tr>
<td>[REV96]</td>
<td>REVERBEI, Francisco</td>
<td>Persistence in Distributed Object Systems: ORB/ODBMS Integration</td>
<td>Ph.D. Dissertation to the Computer Science Department of the University of New Mexico, April 1996</td>
<td></td>
</tr>
</tbody>
</table>
Appendix


## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2PC</td>
<td>Two-Phase-Commit Protocol</td>
</tr>
<tr>
<td>ACID</td>
<td>Acronym for Atomicity, Consistency, Isolation, Durability</td>
</tr>
<tr>
<td>API</td>
<td>Application Programmer Interface</td>
</tr>
<tr>
<td>BOA</td>
<td>Basic Object Adapter</td>
</tr>
<tr>
<td>BOCA</td>
<td>Business Object Component Architecture</td>
</tr>
<tr>
<td>BOF</td>
<td>Business Object Facility/Business Object Framework</td>
</tr>
<tr>
<td>BSD</td>
<td>Business System Domain</td>
</tr>
<tr>
<td>CASE</td>
<td>Computer Aided Software Engineering</td>
</tr>
<tr>
<td>CDL</td>
<td>Component Definition Language</td>
</tr>
<tr>
<td>CGI</td>
<td>Common Gateway Interface</td>
</tr>
<tr>
<td>CICS</td>
<td>Customer Information Control System</td>
</tr>
<tr>
<td>CLI</td>
<td>Call-Level Interface</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>DBMS</td>
<td>Database Management System</td>
</tr>
<tr>
<td>DCOM</td>
<td>Distributed Component Object Model</td>
</tr>
<tr>
<td>DTP</td>
<td>Distributed Transaction Processing</td>
</tr>
<tr>
<td>EJB</td>
<td>Enterprise JavaBeans</td>
</tr>
<tr>
<td>GIOP</td>
<td>General Inter-ORB Protocol</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hyper-Text Transfer Protocol</td>
</tr>
<tr>
<td>IDL</td>
<td>Interface Definition Language</td>
</tr>
<tr>
<td>IIOP</td>
<td>Internet Inter-ORB Protocol</td>
</tr>
<tr>
<td>IMS</td>
<td>Information Management System. Synonym for IMS/VS.</td>
</tr>
<tr>
<td>IMS/VS</td>
<td>Information Management System/Virtual Storage.</td>
</tr>
<tr>
<td>ISO</td>
<td>International Standards Organization</td>
</tr>
<tr>
<td>ITS</td>
<td>Integrated Transaction Service</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java Database Connectivity</td>
</tr>
<tr>
<td>JDK</td>
<td>Java Developer Kit</td>
</tr>
<tr>
<td>JTS</td>
<td>Java Transaction Service</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>MOM</td>
<td>Message-Oriented Middleware</td>
</tr>
<tr>
<td>MTS</td>
<td>Microsoft Transaction Service</td>
</tr>
<tr>
<td>ODBC</td>
<td>Open Database Connectivity</td>
</tr>
<tr>
<td>ODMG</td>
<td>Object Database Management Group</td>
</tr>
<tr>
<td>OLTP</td>
<td>Online Transaction Processing</td>
</tr>
<tr>
<td>OMA</td>
<td>Object Management Architecture</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>OQL</td>
<td>Object Query Language</td>
</tr>
<tr>
<td>ORB</td>
<td>Object Request Broker</td>
</tr>
</tbody>
</table>

---

1 A directory of abbreviations in Computer Science and Telecommunications can be found at http://zaphod.cs.uni-sb.de/Corner/Abklex/abklex.html.
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSI-TP</td>
<td>Open Systems Interconnection – Transaction Processing</td>
</tr>
<tr>
<td>OTM</td>
<td>Object Transaction Manager</td>
</tr>
<tr>
<td>OTS</td>
<td>Object Transaction Service</td>
</tr>
<tr>
<td>POA</td>
<td>Portable Object Adapter</td>
</tr>
<tr>
<td>PSS</td>
<td>Persistent State Service</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>SNA</td>
<td>Systems Network Architecture</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>VM</td>
<td>Java virtual machine</td>
</tr>
<tr>
<td>X/Open DTP</td>
<td>X/Open Distributed Transaction Processing</td>
</tr>
</tbody>
</table>
A - 3 Glossary

Most of the descriptions provided in this glossary are taken from Web resources like dictionaries and white papers. Online glossaries can be found at http://www.matisse.net/files/glossary.html and http://foldoc.doc.ic.ac.uk/foldoc/index.html.

Activation. Bringing an executable component into a live state, after which it can respond to invocations.

Application Server. A server program that allows the installation of application-specific software components, in a manner so that they can be remotely invoked, usually by some form of remote method call.

Business object. An object which represents a corresponding entity in the real world of the business. A business object has identity that corresponds to the identity of the real-world entity. Business objects are (usually) persistent and recoverable. Operations on business objects are in a transactional context that assures concurrency control and supports commit and roll-back operations.

Business system domain (BSD). A run-time, distributed system that implements a consistent business model, is managed by a single business entity, is maintained consistent with revisions and is recoverable to a consistent state. Different business system domains may interoperate, but must be loosely coupled through adapters to achieve compatibility and maintain integrity within each BSD. Related BSD's may operate on different schedules and will have independent administrative controls.

Component. According to [HOF97] "A Software component is pieces of software with one or more well-defined interfaces that are configurable, integrable, and not modifiable."

Component Standard. A definition of how software components cooperate, and in particular the roles and interfaces of each.

CICS. Customer Information Control System. IBM's general-purpose online transaction processing (OLTP) software. It is a powerful application server that runs on a range of operating systems from the smallest desktop to the largest mainframe.

Deadlock. A situation where two or more processes are unable to proceed because each is waiting for one of the others to do something. A common example is a program communicating to a server, which may find itself waiting for output from the server before sending anything more to it, while the server is similarly waiting for more input from the controlling program before outputting anything.
Extent. The logical set of all members of a type and its sub-types within a business system domain. The extent of a type defines the scope of queries on the type—it is all "known" instances of the type.

Factory. An object of a particular type is instantiated in a remote address space by requesting the creation of an object from the appropriate factory object in that address space. Consequently, each existing factory object in a distributed environment represents an address space where an object of the type it creates can be created. In the BOCA, factories are managed by the type manager.

Failover. The ability to respond resiliently to a component failure by switching to another component.

Framework. In object-oriented systems, a set of classes that embodies an abstract design for solutions to a number of related problems.

Introspection. The ability of an object to provide information about itself. A fundamental level of introspection is the ability for an object to identify its attributes and relationships. Such information can be very useful for the implementation of generalized tools.

Iterator. An object which retrieves successive members or groups of members from a collection. An iterator may operate on a real or virtual collection, as where the members being returned are not determined until they are requested. Iterators are used to access the collections implicit in one-to-many relationships, to provide query results and for other situations where access is provided to a controlled and/or shared collection.

Legacy system. A legacy system is typically a system being replaced. In general terms it is a system that does not comply with current architectural specifications for the computing environment or is in some way incompatible with systems currently under development. The primary concern with legacy systems is integration and/or migration. Integration involves efforts to build an transparent interface for cooperation. Migration involves replacing components of the legacy system until it is no longer needed.

Life cycle. A set of operations on objects related to their creation and destruction. These are create, delete, copy and move. For business objects, this is extended to include activate and deactivate, i.e., retrieval from persistent storage and removal from main storage.

Marshalling. Marshalling is the process of packing one or more items of data into a message buffer, prior to transmitting that message buffer over a communication channel. It converts different types into a standard representation agreed with the recipient of the message.

Object key. The object key is a unique identifier for a business object. It is an opaque value, interpretable by the business object's implementation. This identifier provides a unique and persistent identity for a business object within a Business System Domain and within its type extent. It is assumed to be imbedded in the object reference of the associated business object. In the BOF, the object key consists of one or more concatenated attribute values, separated by tildes (~).
Object reference. The value passed by object request brokers to for references to an object. The object reference identifies the address space and the object within the address space. It is assumed that, for business objects interoperability, the object reference contains an imbedded object identifier which can be used to retrieve the persistent state of the object when the object reference is no longer valid and to determine if two object references refer to the same business object.

Online Transaction Processing (OLTP). OLTP is a class of program that facilitates and manages transaction-oriented applications, typically for data entry and retrieval transactions in a number of industries, including banking, airlines, mail-order, supermarkets, and manufacturers. Probably the most widely installed OLTP product is IBM's CICS (Customer Information Control System). New OLTP software uses client-server processing and brokering software that allows transactions to run on different computer platforms in a network.

Orthogonal persistence. In an orthogonal persistent system, the application programmer can be unaware of whether an object persists or not. Objects persist if they need to, and they are always in a consistent state. Thus the persistence of data is orthogonal of (independent of) all its other properties.

Persistent. Objects are persistent if their state information is preserved when the computer is turned off. Typically this means that their state is stored in a database. Business objects and their associated dependent objects are expected to be persistent. The interfaces of the BOCA specification assume that the activation, database update and deactivation of objects is handled by the object implementation without explicit action by the users of the objects.

Persistent identifier. The persistent identifier is a pass-by-value object which contains the type of the referenced business object and a unique identifier within its type (→ object key).

Pooling. Maintaining a collection of objects, servers, connections, or other resources for ready access, so that one does not need to be created anew each time one is needed.

Skeleton. A server-side software component that serves to relay remote calls from a client to the methods of a servant running in a server. Usually a skeleton is automatically generated by a special compiler.

Stub. A client-side component that serves to forward remote calls to a remote server, and receive the subsequent responses. Usually a skeleton is automatically generated by a special compiler.

Transient. An object's state, i.e., the values of an object's properties at a given point in time, can be transient or persistent. When an object's property values are assigned at the initiation of a method call and do not persist beyond the execution of that method call, the object is considered transient.
A - 4  IDL Descriptions of BOF Components

A - 4.1  Business Object Framework by Kai-Uwe Schäfer

A - 4.1.1  datatypes.idl

#ifndef _DTYPES
#define _DTYPES
#pragma prefix "kus.vas"

module bof {

typedef string AName;

struct NameValue {

    AName the_name;
    any the_value;

};

typedef sequence<NameValue> NameValues;

typedef sequence<AName> Names;

}; // end module bof
#pragma prefix ""
#endif

A - 4.1.2  exception.idl

#ifndef _EXCEPTION
#define _EXCEPTION
#pragma prefix "kus.vas"

module bof {

struct Error {

    string exception_source;
    long  exception_code;
    string exception_reason;

};

typedef sequence<Error> Errors;

exception BOexception {

}
Errors errors;
);
}
}; // end module bof

#pragma prefix ""
#endif

A - 4.1.3 bof.idl

#ifndef _BOF
#define _BOF

#include "exception.idl"
#include "datatypes.idl"
#include "CosTransactions.idl"

#pragma prefix "kus.vas"

module bof {

interface BusinessObject;
interface Iterator;
interface TypeManager;

typedef sequence<octet> OpaquelD;

struct PersistentID {
   string type;
   OpaquelD opaque_id;
};

typedef sequence<string> QueryExpression;

interface Configuration {

   NameValues get_GarbageCollectorPolicy();

   void set_GarbageCollectorPolicy(in NameValues policies)
      raises (BOexception);
      // Exception: 101 Undefined name.
      // Exception: 102 Invalid value.

   string get_ThreadPolicy();

   long get_numberOfThreads();

   long get_numberOfBusinessObjects();
}; // end of interface Configuration

// --------- TypeManager --------- //
interface TypeManager : CosTransactions::TransactionalObject {

    readonly attribute string type;
    readonly attribute Configuration config;

    BusinessObject create_object(in NameValues init_values)
        raises (BOexception);
        // Exception: 201 Undefined name in expression.

    BusinessObject create_from_object(in BusinessObject source)
        raises (BOexception);
        // Exception: 202 Invalid source.

    BusinessObject resolve_persistent_id(in PersistentID persistent_id)
        raises (BOexception);
        // Exception: 203 Unable to resolve.

    Iterator query(
        in QueryExpression the_query,
        in NameValues names_in_expression,
        in Names ordered_by)
        raises (BOexception);
        // Exception: 204 Invalid query expression
        // Exception: 205 Undefined name in expression
        // Exception: 206 No members satisfy query
    )}; // end interface TypeManager

// --------- BusinessObject --------- //

interface BusinessObject : CosTransactions::TransactionalObject {

    PersistentID get_PersistentID();

    boolean is_identical(in BusinessObject obj);

    TypeManager get_TypeManager();

    void add_to_relationship(in string relationship_name, in BusinessObject member)
        raises (BOexception);
        // Exception: 401 Unrecognized relationship name
        // Exception: 402 Member assignment type mismatch
        // Exception: 403 Member already in relationship

    void remove_from_relationship(in string relationship_name, in BusinessObject member)
        raises (BOexception);
        // Exception: 404 Missing member to remove
        // Exception: 405 Unrecognized relationship name
        // Exception: 406 Member missing from complementary relationship

    boolean will_notify_for_relationship (in string relationship_name)
        raises (BOexception);
        // Exception: 407 Unrecognized relationship name

    void delete()
        raises (BOexception);
        // Exception: 408 Delete related object first.

}
typedef sequence<BusinessObject> BOsequence;

// Tuples
typedef sequence<string> AttributeNames;
typedef sequence<any> AttributeValues;

struct Tuple {
    PersistentID obj_id;
    AttributeValues values;
};
typedef sequence<Tuple> TupleValues;

struct Tuples {
    AttributeNames names;
    TupleValues values;
};

// Iterator
interface Iterator {
    void goto_start();
    void goto_end();
    long how_many()
        raises (BOexception);
        // Exception: 501 Not supported
    void delete_iterator();

    // The following methods are for queries
    Tuple get_next_tuple()
        raises (BOexception);
        // Exception: 502 No members to get
        // Exception: 503 Not supported for attributes
    Tuples get_next_n_tuples(in long max_number)
        raises (BOexception);
        // Exception: 504 No members to get
        // Exception: 505 Invalid max_number
        // Exception: 506 Not supported for attributes
    Tuple get_previous_tuple()
        raises (BOexception);
        // Exception: 507 Not supported
        // Exception: 508 No members to get
        // Exception: 509 Not supported for attributes
    Tuples get_previous_n_tuples(in long max_number)
}
Appendix

raises (BOexception);
// Exception: 510 Not supported
// Exception: 511 No members to get
// Exception: 512 Invalid max_number
// Exception: 513 Not supported for attributes

void position_at (in PersistentID member)
raises (BOexception);
// Exception: 514 Not supported
// Exception: 515 Invalid member
// Exception: 516 Not supported for attributes

Iterator query(
    in QueryExpression the_query,
    in NameValues names_in_expression)
raises (BOexception);
// Exception: 517 Invalid query expression
// Exception: 518 Undefined name in expression
// Exception: 519 No members satisfy query criteria

BusinessObject get_next_object()
raises (BOexception);
// Exception: 520 No more objects to get

BOsequence get_next_n_sequence(in long max_number)
raises (BOexception);
// Exception: 521 No more objects to get
// Exception: 522 Invalid max_number

BusinessObject get_previous_object()
raises (BOexception);
// Exception: 523 No more objects to get

BOsequence get_previous_n_sequence(in long max_number)
raises (BOexception);
// Exception: 524 No more objects to get
// Exception: 525 Invalid max_number


A - 4.2 OMG Object Transaction Service (OTS)

// CosTransactions.idl
// provided by Inprise Corp. for Inprise Integrated Transaction Service (ITS)

#ifndef _costransactions_idl_
#define _costransactions_idl_

#pragma prefix "omg.org"

module CosTransactions
{


// Forward references for interfaces defined later in module

// In Java we will generate pseudo classes for Current
interface Current;

interface TransactionFactory;
interface Control;
interface Terminator;
interface Coordinator;
interface RecoveryCoordinator;
interface Resource;
interface Synchronization;
interface SubtransactionAwareResource;
interface TransactionalObject;

// DATATYPES
enum Status
{
    StatusActive,
    StatusMarkedRollback,
    StatusPrepared,
    StatusCommitted,
    StatusRolledBack,
    StatusUnknown,
    StatusNoTransaction,
    StatusPreparing,
    StatusCommitting,
    StatusRollingBack
};
enum Vote
{
    VoteCommit,
    VoteRollback,
    VoteReadOnly
};

// Structure definitions
struct otid_t
{
    long formatID; /*format identifier. 0 is OSI TP */
    long bqual_length;
    sequence <octet> tid;
};
struct TransIdentity
{
    Coordinator coordinator;
    Terminator terminator;
    otid_t otid;
};
struct PropagationContext
{
    unsigned long timeout;
    TransIdentity current;
    sequence <TransIdentity> parents;
    any implementation_specific_data;
};

// Heuristic exceptions
exception HeuristicRollback {};
exception HeuristicCommit {};  
exception HeuristicMixed {};  
exception HeuristicHazard {};  
// Other transaction-specific exceptions  
exception SubtransactionsUnavailable {};  
exception NotSubtransaction {};  
exception Inactive {};  
exception NotPrepared {};  
exception NoTransaction {};  
exception InvalidControl {};  
exception Unavailable {};  
exception SynchronizationUnavailable {};  

// Current transaction  
//interface Current : CORBA::ORB::Current  

// In Java we will generate pseudo classes for Current  
interface Current  
{
    void begin()  
        raises(SubtransactionsUnavailable);  
    void commit(in boolean report_heuristics)  
        raises(  
            NoTransaction,  
            HeuristicMixed,  
            HeuristicHazard  
        );  
    void rollback()  
        raises(NoTransaction);  
    void rollback_only()  
        raises(NoTransaction);  
    Status get_status();  
    string get_transaction_name();  
    void set_timeout(in unsigned long seconds);  
    Control get_control();  
    Control suspend();  
    void resume(in Control which)  
        raises(InvalidControl);  
};  

interface TransactionFactory  
{
    Control create(in unsigned long time_out);  
    Control recreate(in PropagationContext ctx);  
};  

interface Control  
{
    Terminator get_terminator()  
        raises(Unavailable);  
    Coordinator get_coordinator()  
        raises(Unavailable);  
};  

interface Terminator  
{
    void commit(in boolean report_heuristics)  
        raises(  
            NoTransaction,  
            HeuristicMixed,  
            HeuristicHazard  
        );  
}
interface Coordinator
{
  Status get_status();
  Status get_parent_status();
  Status get_top_level_status();
  boolean is_same_transaction(in Coordinator tc);
  boolean is_related_transaction(in Coordinator tc);
  boolean is_ancestor_transaction(in Coordinator tc);
  boolean is_descendant_transaction(in Coordinator tc);
  boolean is_top_level_transaction();
  unsigned long hash_transaction();
  unsigned long hash_top_level_tran();
RecoveryCoordinator register_resource(in Resource r)
  raises(Inactive);
void register_synchronization (in Synchronization sync)
  raises(Inactive, SynchronizationUnavailable);  
void register_subtran_aware(in SubtransactionAwareResource r)
  raises(Inactive, NotSubtransaction);
void rollback_only() 
  raises(Inactive);
string get_transaction_name();
Control create_subtransaction()
  raises(SubtransactionsUnavailable, Inactive);
PropagationContext get_txcontext ()
  raises(Unavailable);
};

interface RecoveryCoordinator
{
  Status replay_completion(in Resource r)
    raises(NotPrepared);
};

interface Resource
{
  Vote prepare() 
    raises(
      HeuristicMixed, 
      HeuristicHazard
    );
void rollback() 
  raises(
    HeuristicCommit, 
    HeuristicMixed, 
    HeuristicHazard
  );
void commit() 
  raises(
    NotPrepared, 
    HeuristicRollback, 
    HeuristicMixed, 
    HeuristicHazard 
  );
void commit_one_phase()
   raises(HeuristicHazard)
   ;
void forget();
;
interface TransactionalObject
{
};

interface Synchronization : TransactionalObject
{
   void before_completion();
   void after_completion(in Status status);
};

interface SubtransactionAwareResource : Resource
{
   void commit_subtransaction(in Coordinator parent);
   void rollback_subtransaction();
}; // End of CosTransactions Module

#pragma prefix "
#pragma endif

A-19
A - 5 Class Diagram for BOF
A - 6  Source Code on CD

- Source code of the business object framework
- Documents

The software used to implement the business object framework includes
- Inprise Integrated Transaction Service (ITS) 1.0
- Borland JBuilder 2.01 (Enterprise Edition)
- Java Development Kit 1.2