An Enterprise Information System Data Architecture Guide

Grace Alexandra Lewis
Santiago Comella-Dorda
Pat Place
Daniel Plakosh
Robert C. Seacord

October 2001

TECHNICAL REPORT
CMU/SEI-2001-TR-018
ESC-TR-2001-018

Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

An Enterprise Information System Data Architecture Guide

CMU/SEI-2001-TR-018
ESC-TR-2001-018

Grace Alexandra Lewis
Santiago Comella-Dorda
Pat Place
Daniel Plakosh
Robert C. Seacord

October 2001

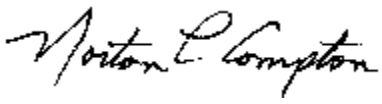
COTS-Based Systems

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER



Norton L. Compton, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Figures

Abstract

Data architecture defines how data is stored, managed, and used in a system. It establishes common guidelines for data operations that make it possible to predict, model, gauge, and control the flow of data in the system. This is even more important when system components are developed by or acquired from different contractors or vendors.

This report describes a sample data architecture in terms of a collection of generic architectural patterns that both define and constrain how data is managed in a system that uses the Java™ 2 Enterprise Edition (J2EE) platform and the Open Applications Group Integration Specification (OAGIS). Each of these data architectural patterns illustrates a common data operation and how it is implemented in a system.

1 Introduction

1.1 Data Architecture

Data architecture defines how data is stored, managed, and used in a system. In particular, a data architecture describes

- how data is persistently stored
- how components and processes reference and manipulate this data
- how external/legacy systems access the data
- interfaces to data managed by external/legacy systems
- implementation of common data operations

The data architecture proposed in this report is for a system based on the Java™ 2 Enterprise Edition (J2EE) platform and the Open Applications Group Integration Specification (OAGIS). The data architecture is described in terms of a collection of generic architectural patterns that both define and constrain how data is managed. Each of these data architectural patterns illustrates common data operations and how these operations are implemented in the target supply system. Without such guidance, common data operations might be implemented differently, making it impossible to predict, model, gauge, or control the flow of data in the system. This data architecture guide can also help to identify and resolve potential design risks resulting from inconsistent or contradictory requirements.

The data architecture is a high-level design that cannot always anticipate and accommodate all implementation details. Some of these details may impose demands that conflict with the data architecture. In these cases, it may be necessary to reevaluate the data architecture to determine what can be done to accommodate the additional demands. It is also allowable to violate the data architecture in places, as long as the rationale for doing so is well understood, well documented, and does not compromise the robustness, performance, and integrity of the overall system.

This document is organized into the following sections:

- Section 2: a brief description of the forces affecting the data architecture
- Section 3: a static view of the proposed architecture, describing application components and business objects (BOs)

- Sections 4 and 5: architectural patterns that describe the way operations take place in the system. Each pattern is instantiated with examples.

1.2 Case Study

This report is based on a case study that provides the context for the proposed data architecture. This case study involves the modernization of a large retail supply system (RSS) for a major U.S. retailer. The RSS consists of approximately 2 million lines of MicroFocus COBOL code running on a Solaris workstation. Data is stored in an Oracle 8i database. However, the overall architecture of the system has remained largely unchanged over 30 years, resulting in a system that is extremely brittle and difficult to maintain. (Comella-Dorda and associates provide a relevant description of an information-system life cycle [Comella 00].) As a result, the decision was made to modernize the RSS to a J2EE platform. In particular, the modernized system will consist of Enterprise JavaBeans™ (EJBs) written in the Java programming language and deployed on an EJB application server. The OAGIS will be used as the standard to integrate systems defined as sets of BOs.

1.3 J2EE Platform and EJBs

The J2EE platform defines a standard for developing multitier enterprise applications. It simplifies enterprise applications by basing them on standardized, modular components, by providing a complete set of services to those components, and by handling many details of application behavior automatically [J2EE].

EJBs make up the server-side component architecture for the J2EE platform. EJBs are reusable, prepackaged pieces of application functionality that are designed to run in an EJB-compliant application server. They can be combined with other components to create customized application systems [Thomas 98]. EJB components are supported by the J2EE platform.

The simplest way to define an EJB is as a component that implements a module of business logic. EJBs are then combined to build an application. There are two types of EJBs: session beans and entity beans. Session beans implement business tasks and entity beans implement business entities.

This report covers some aspects of EJBs and the J2EE platform. For more information about them, see these Sun Microsystems, Inc. Web sites: http://www.javasoft.com/j2ee and http://www.javasoft.com/ejb.

1.4 Open Applications Group Integration Specification (OAGIS)

The OAGIS prescribes a content-based, virtual, business object model (shown in Figure 1) that enables an enterprise business application to build a virtual-object wrapper around itself. To communicate with a business software component in this model, events are communicated through the integration backbone in the form of an OAGIS-compliant, business object document (BOD) to a virtual-object interface.

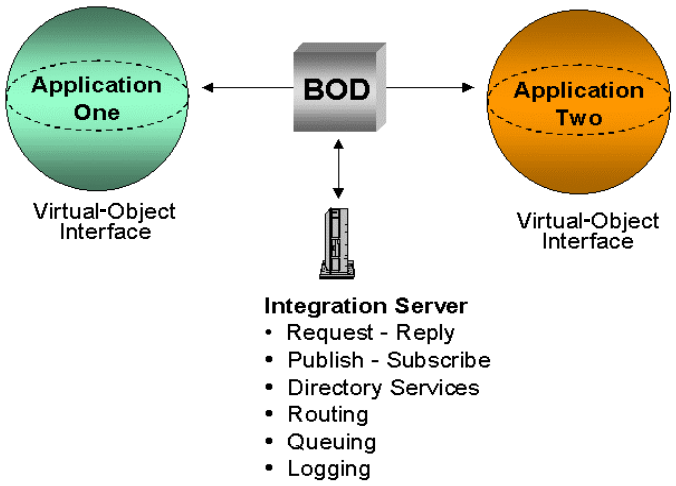


Figure 1. OAGIS Virtual Business Object Model

The BOD uses metadata, in the form of an extensible markup language (XML) schema. It contains the framework necessary to convey its two primary components: the business service request and the business data area. This BOD structure is shown in Figure 2.

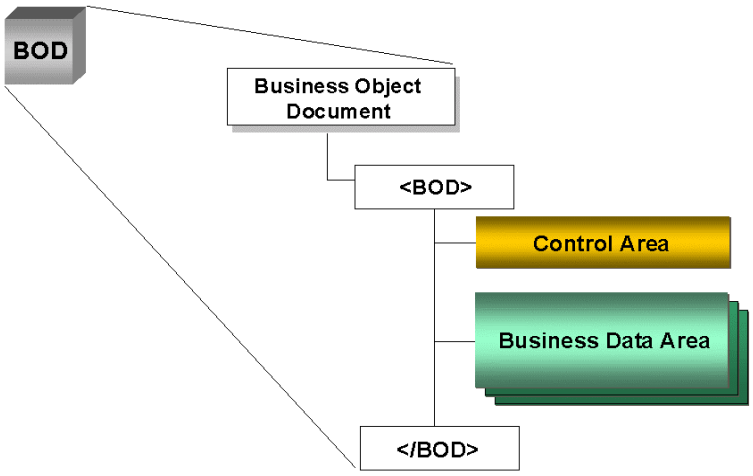


Figure 2. BOD Structure

Each business service request (BSR) contains a unique verb/noun combination such as POST JOURNAL or SYNC ITEM that drives the contents of the business data area (BDA). This BSR and BDA combination corresponds to the object name, method, and arguments model of a procedure call or method-invocation model.

The use of BSRs prescribes a loosely coupled communication mechanism based on messages. The intent of this architecture is to support plug-and-play integration. Applications talk to each other using BSRs and are integrated though a backbone that handles message-based communication, as shown in Figure 3.

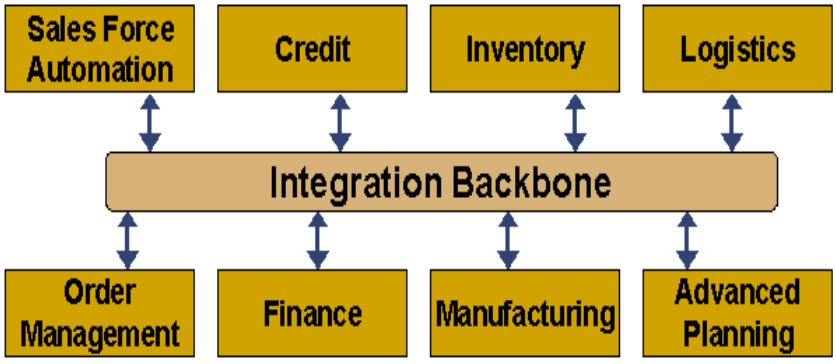


Figure 3. Application Integration Using BSRs

2 Forces Affecting the Data Architecture

Data, organizational, and technology requirements all constrain and influence the data architecture. The data architecture has to reflect the forces imposed by these often-incompatible requirements.

2.1 Data Requirements

Data requirements are driven by functional requirements. Examples of common data requirements in enterprise information systems are summarized in the following paragraphs.

Reports and queries, including flexible (ad hoc) queries: Reports and queries involve extracting, relating, and summarizing data from one or more tables. Reports evolve and new reports are often added. Support for ad hoc query capability is required so that users can enter their queries in structured query language (SQL) or using query tools.

Persistent summaries and roll ups: Summaries and roll ups are reports produced from consolidated information that involves the extraction of data from multiple tables. Roll-up and summary information is then stored in the database. The process for collecting and analyzing this data may be computationally intensive, potentially requiring the creation of interim tables to store data temporarily.

Data warehousing: A data warehouse is a collection of data designed to support management decision making at the enterprise level. Data warehouses contain a wide variety of data that presents a coherent picture of business conditions at a single point in time. The development of a data warehouse includes the development of systems to extract data from operating systems plus the installation of a warehouse database system that provides managers with flexible access to the data. The term *data warehousing* generally refers to combining many different databases across an entire enterprise. A data mart is a database, or collection of databases, designed to help managers make strategic decisions about their business. Whereas a data warehouse combines databases across an entire enterprise, data marts are usually smaller and focus on a particular subject or department. Some data marts, called *dependent data marts,* are subsets of larger data warehouses. Data in a data mart is accessed using a business intelligence (BI) application. An example of data marts and data warehouses as seen by IBM in its Information Aggregation pattern is shown in Figure 4 [IBM].

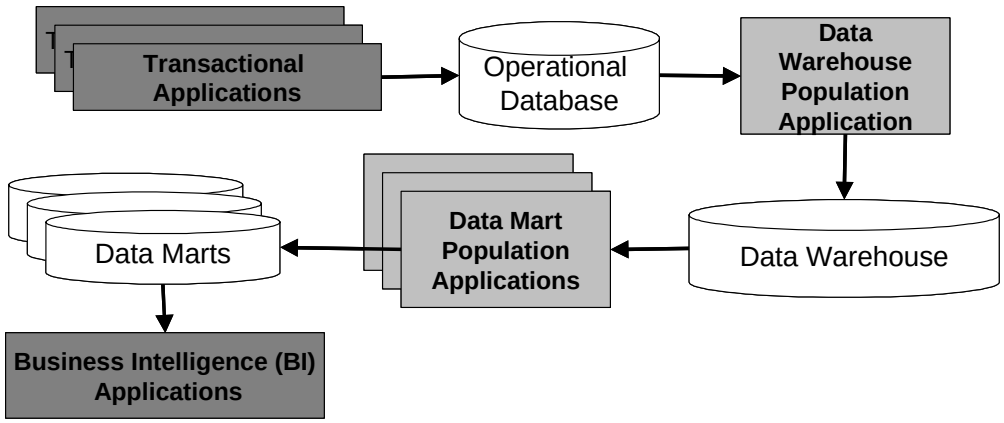


Figure 4. Data Marts and Data Warehouses

Complex transactions: Enterprise information systems must support high volumes of transactions involving data elements in dispersed areas of the system in an efficient manner.

2.2 Organizational Requirements

In any organization, data needs to be fully integrated and seamlessly accessible. Given the distributed nature of today's organizations (as shown in Figure 5) the data in the headquarters, in any region, and in any office has to be accessible, given that the user has the appropriate permissions.

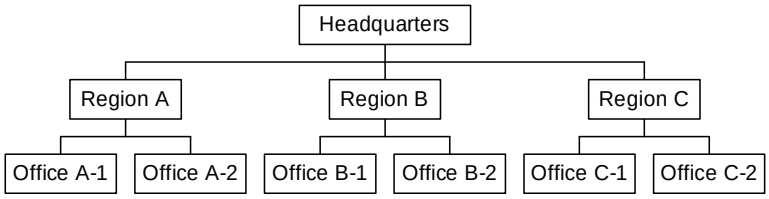


Figure 5. Distributed Organization

The challenges in data distribution are best summarized by the promises of distributed databases [Ozsu 99]:

- transparent management of distributed, fragmented, and replicated data
- improved reliability/availability through distributed transactions
- improved performance
- easier and more economical system expansion

2.3 Technology Requirements

Technology requirements are often suggested or imposed by existing organizational integration frameworks or standards. These frameworks exist for a variety of reasons. By identifying standard products, an organization often hopes to share expertise between development projects and lower licensing costs by leveraging site or multiple-license discounts. While these frameworks are often a reasonable approach, care has to be given to provide a process for obtaining waivers when these technology choices are inappropriate for a project and to consider the effect of the continued evolution of the framework and its constituent products.

In our case study, the following technology is "suggested" by the organizational integration framework:

- J2EE platform
- EJBs written in the Java programming language and deployed on an EJB-application server
- the OAGIS as the mechanism to integrate systems defined as sets of BOs
- MQSeries as the message-oriented middleware package for exchanging BSRs between OAGIS BOs
- Oracle 8i as the relational database management system (RDBMS)

3 Overview of the Proposed Data Architecture

A simplified view of an application that combines the J2EE platform and the OAGIS is shown in Figure 6. The architecture can be viewed roughly as a three-tier, layered architecture where: the application components represent the presentation tier; the BOs represent the business logic tier; and the database represents the data tier. A more detailed view of the architecture is shown in Figure 7 and discussed in the following sections.

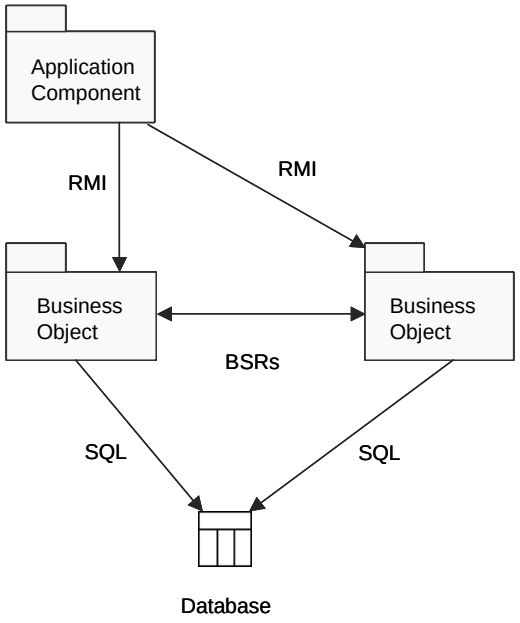


Figure 6. Simplified View of an Application that Uses the J2EE Platform and the OAGIS[1]

[1] RMI (remote method invocation) is used as the mechanism for communication between application components and business objects. SQL is used as an interface between business objects and the database.

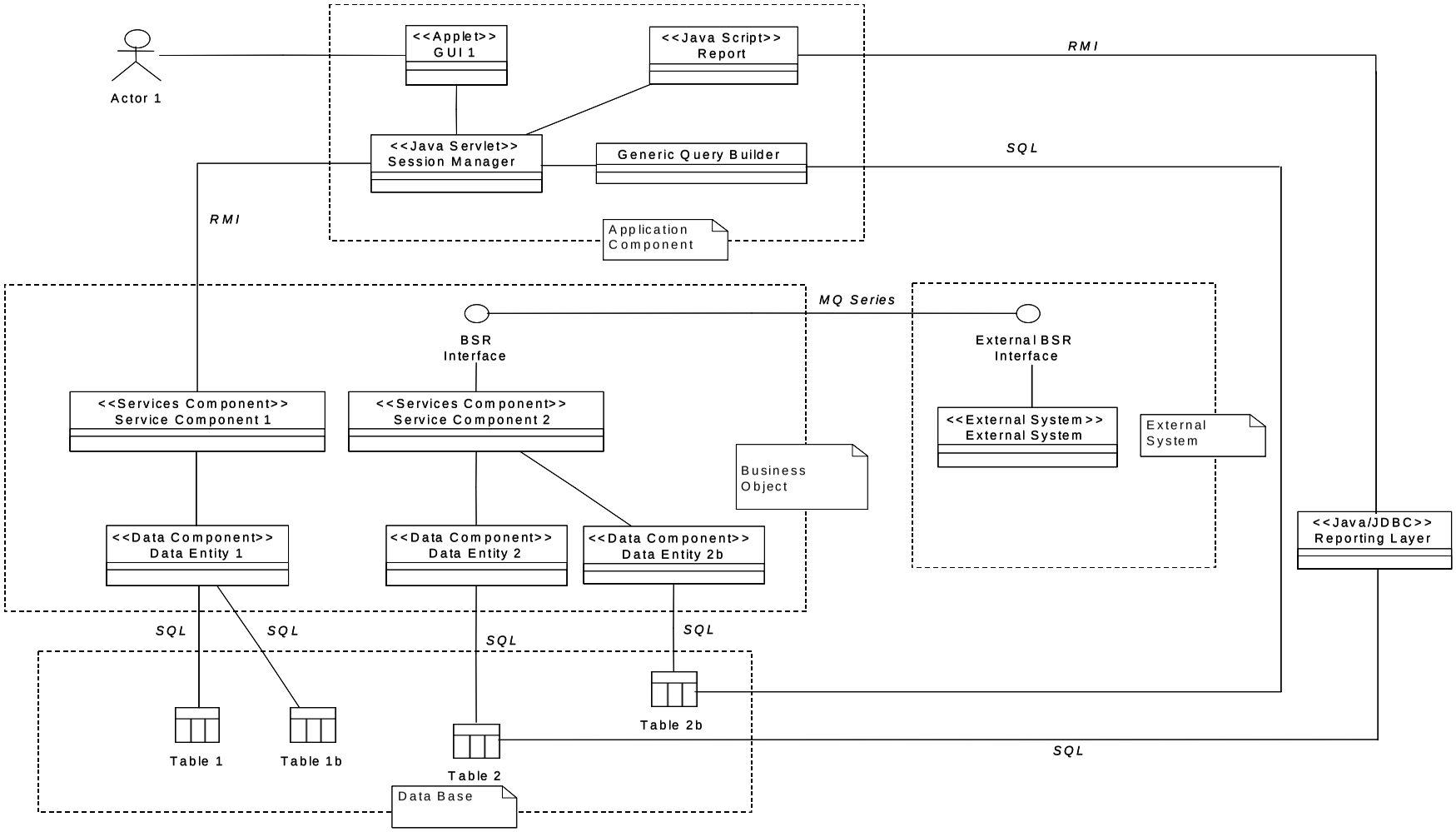


Figure 7. Conceptual Architecture of an Application that Uses J2EE and OAGIS Application Components

Application components encapsulate application-specific logic, including user interfaces, reports, query building, and application-specific workflows.

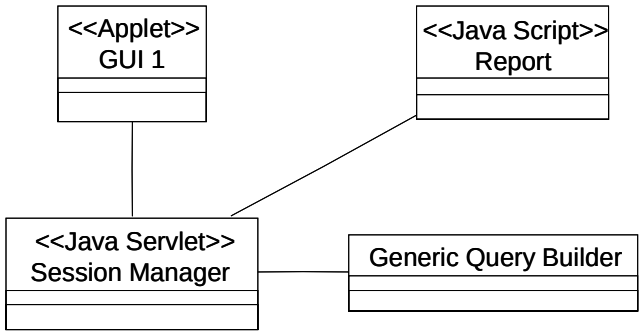


Figure 8. Application Component Structure

The structure of an application component is shown in Figure 8. The application component contains

- a graphical user interface (GUI): The GUI is represented by an applet that displays and obtains information from the user.
- reports: These are preset reports implemented by Java scripts that use the services of the reporting layer. The reporting layer provides a level of abstraction using the Java Database Connectivity (JDBC) layer to access the database. This makes the reports independent of the database implementation.
- a generic query builder: The generic query builder allows the user to construct ad hoc queries to the database. It can be an applet, an SQL prompt, or any commercial query-building tool.
- a session manager: The session manager is a Java servlet[2] that implements workflow and session management. It accepts the user's input, makes invocations to the service components located in the BOs, and then issues a response to the client.

Application components can communicate with BOs using BSRs, non-standard component application programming interfaces (APIs), or some combination of both. Communication using BSRs has the advantage of maintaining a greater degree of independence between applications and BOs. In particular, the use of non-standard APIs can create a dependency on a particular implementation of a BO within the application component. However, the granularity and performance of BSR-based communication may not be sufficient to support user interactions. As a result, the OAGIS permits the use of non-BSR communication in this situa-

[2] Session management can also be implemented as Java Server Pages (JSPs). JSPs basically provide an HTML-like interface for programming Java servlets.

tion. The data architecture uses remote method invocation (RMI) to communicate between application components and BOs.

Application components access data through the BOs, except in the case of reports and ad hoc queries: in reports, access is performed through a reporting layer that accesses the database; in ad hoc queries, access to the database is direct.

3.1 Business Objects (BOs)

Business objects (BOs) encapsulate the business logic of a single business entity and data particular to that entity. BOs communicate with each other through BSRs containing BODs and with the database layer using SQL. Oracle is the RDBMS.

For each BO one or more *service components* and one or more *data components* are defined. A service component (SC) represents a piece of functionality to be provided by a BO. A data component (DC) provides encapsulation and communicates with the database to obtain and update information.

Service components within the business object must provide a BSR interface for communication with other BOs, as described in Section 1.4. In addition, these service components may provide other external interfaces that may be invoked by application components or other service components within the BO.

Service components also need to communicate with external systems. *Wrapper components* are used for communication with legacy systems that do not have a BSR interface.

Given the loosely coupled nature of the integration scheme proposed by the OAGIS, it is tempting to decompose the system into as many business objects as possible to take advantage of being able to replace components easily. Nevertheless, the partitioning of a system into BOs is a critical issue due to constraints imposed by the message-based communication required by BSRs and their impact on performance and transaction management.[3]

Having a large number of BOs is not recommended if there is a high coupling between the defined BOs that would cause large traffic due to BSR-based communication. This decomposition could potentially degrade performance, require the construction of adapters, and require the implementation/adaptation/construction of a transaction-management system.

Choosing to decompose a system in more than one BO requires a thorough analysis of scenarios versus BOs to determine which BOs are highly coupled. Decreasing the number of

[3] Seacord, Robert C., et al. *Modernizing Legacy Systems*. Boston, MA: Addison-Wesley, to be published.

BOs does not eliminate BSR-based communication, but does reduce it. Taking this approach requires manual transaction-management mechanisms because the transaction context cannot be maintained easily with MQSeries as the message-oriented middleware package.

Building the system as one BO eliminates internal BSR-based communication. Communication with BSRs is required only for communication with external systems that have a BSR interface. The problem with this approach is that the advantages granted by loose coupling are limited because it reduces the potential for interchanging components and adopting commercial components.

The proposed data architecture defines the system as one BO with a BSR interface to communicate with external systems or commercial components. Adapters will have to be constructed if the commercial product does not have a BSR-based interface. This could be a momentous task depending on the underlying technology.

The OAGIS model of XML-based communication is mainly for business-to-business (B2B) and application-to-application (A2A) communication between applications built by different vendors. The parsing, construction, and communication using BSRs adds a great overhead when used in a tightly coupled application like an RSS. The BSR interface should be used only for communication with external OAGIS-compliant applications.

3.1.1 Service Components

A service component represents business logic, the functionality to be provided by a BO. It communicates with one or more data components to obtain information, to respond to a call from another service component or an application, to respond to a BSR, or to generate a BSR.

Service components do not access the database directly; they access it through data components (see Section 3.1.2 on page 15). Service components perform bulk operations on data components, on behalf of Java servlets in the application components.

A related concept is found in the Session Entity Façade pattern [J2EE 01a]. The service component acts as a façade—a unified, simple interface to all of the service's clients. Those clients use the service component as "one-stop shopping" for functionality and data access.

Service components are implemented as EJB session beans. Session beans act as agents for clients and are typically instantiated for each client session. Order placement is a good example of functionality that would be implemented as a session bean.

The access of data components through service components also simplifies transaction management. For example, if "Required" is specified as the transaction attribute on the session bean implementing the service component, all entity-bean accesses run in the session

bean implementing the service component, all entity-bean accesses run in the session bean's transaction.

EJBs define stateless and stateful session beans:

- stateless session beans: These are components that model business processes performed in a single method call. They hold no conversational state on behalf of clients, which means that they are free of a client-specific state after each method call. For a stateless session bean to be useful for a client, the client must pass all client data that the bean needs as parameters to business logic methods.
- stateful session beans: These are conversational beans because they hold conversations with clients that span multiple method invocations. Stateful session beans store a conversational state within the bean, and this state must be available for that same client's next method request.

When choosing between a stateful and a stateless session bean, the question to ask is, "What type of business process is the session bean attempting to emulate?" If the business process spans multiple invocations, the stateful model fits well because client-specific conversations are part of the bean state. If the business process is being performed on behalf of a specific client (an order placement, for example), a stateful session bean is appropriate.

A generic service is modeled as a stateless session bean. A service that manipulates multiple rows in a database and represents a shared view of the data is a common stateless session bean. An example of such a service is a catalog that presents a list of various products and categories. Since all users are interested in this information, the stateless session bean that represents it could be shared easily. When implementing behavior to visit multiple rows in a database and present a read-only view of data, stateless session beans are the best choice. They are designed to provide generic services to multiple clients.

Because session beans encapsulate a business task, service components typically match use cases—there is a strong correlation between service components and use cases identified in the system. Examples of use cases in an RSS are order placement, requisition, and delivery. Since these are services performed on behalf of a client, they should be modeled as stateful session beans.

When defining use cases, it is common to encounter steps that are common to other use cases (e.g., obtaining a list of items). These steps are usually extracted from the use case and modeled as use cases themselves. They are associated to the original use case with the *include relationship*. Use cases that provide a service should be modeled as stateless session beans.

3.1.2 Data Components

The data components provide encapsulation and communicate with the database to obtain or update information. In this architecture, data components are implemented as EJB entity beans.

Data components represent data in a database and add behavior to that data. Instead of writing database logic in an application, the application simply uses the remote interface to the entity bean to access its data.

Entity beans can use either bean-managed or container-managed persistence. With bean-managed persistence, database calls are implemented within the bean, for example, using JDBC. With container-managed persistence, there is no persistence logic inside the bean: the EJB container manages data persistence based on information provided in the deployment descriptor.

Even though having container-managed persistence has benefits, such as smaller bean size and no need to write data-access logic, bean-managed persistence proves to be better in large and complex systems for three reasons:

1. There is usually the need to write some logic, especially for finder methods.
2. Complex data fields may not be directly mappable to underlying storage.
3. There may be relationships between entity beans that must be specified.

Entity beans should represent coarse-grained objects, such as those that provide complex behavior beyond simply getting and setting field values. These coarse-grained objects typically have dependent objects, which are objects that have real domain meaning only when they are associated with their coarse-grained parents. Data components are modeled by coarse-grained entity beans and have Java-class subcomponents that represent the finer-grained, dependent-data elements. This is represented in the Aggregate Entity Pattern which says to use an aggregate entity bean to model, represent, and manage a set of interrelated persistent objects rather than representing them as individual fine-grained entity beans [Larman 00]. An aggregate entity bean represents a tree of objects.

For example, an invoice can be represented as a coarse-grained entity bean. The lines in the invoice are dependent objects. This representation avoids the following problems:

- an interentity-bean communication bottleneck, if the entity-bean schema were to match the relational schema (each table represented by an entity bean). There would be an entity bean for each invoice and for each line in the invoice.
- overhead due to the presence of a large number of entity beans
- fine-grained management for fine-grained components

- numerous dynamically established connections between entity beans
- distributed debugging

Figure 9 shows the aggregate entity pattern in which entity beans represent independent objects with associated sets of dependent objects (subcomponents), thus providing a coarse-grained, entity-bean schema. The aggregate entity pattern states:

> *"In general, an entity bean should represent an independent business object that has an independent identity and life cycle, and is referenced by multiple enterprise beans and/or clients. A dependent object should not be implemented as an entity bean. Instead, a dependent object is better implemented as a Java class (or several classes) and included as part of the entity bean on which it depends" [Larman 00].*

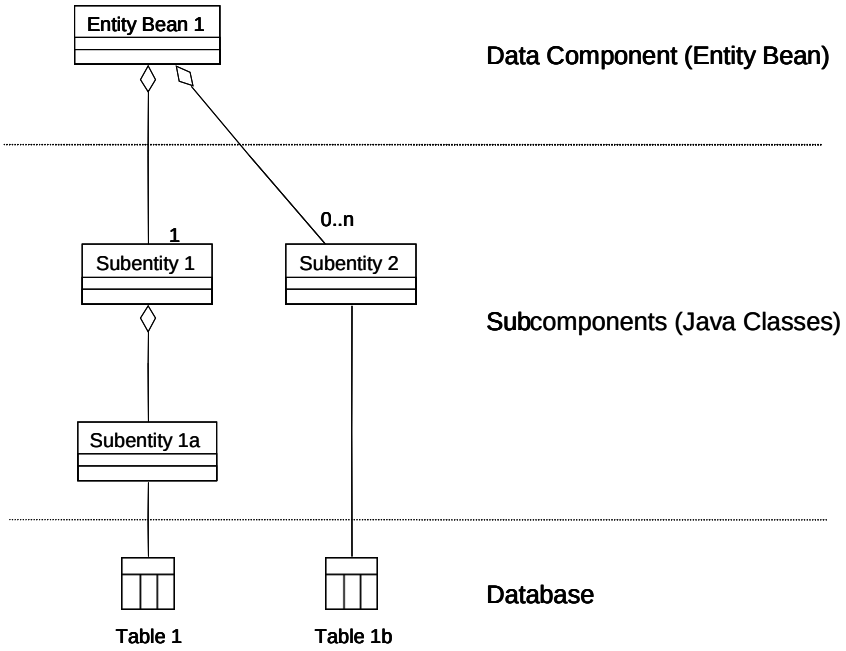


Figure 9. Data Component Internals – Aggregate Entity Pattern

This form of aggregation requires access to subcomponents to be performed always through the entity bean. One way to reduce the overhead of having to go through the entity bean is to have *group operations* when necessary. This means that in the above figure, for example, *EntityBean1* could have a method *SetSubentity1(…)* that sets all the attributes in *Subentity1* without having to call an individual set method for every attribute.

Elements identified as data components have common qualities. In particular, they

- are referenced by more than one component or client
- have an independent life cycle that is not bound or managed by the life cycle of another element
- require a unique identity
- provide complex behavior beyond simply getting and setting field values
- usually have dependent objects, which are objects that have no real domain meaning when standing alone

3.1.3 Wrapper Components

A wrapper component is an adapter that allows a service component to communicate with an external system.

There are two situations in which a wrapper should be used, as shown in Figure 10. A BSR-to-API wrapper is used to communicate with an external legacy system that will eventually become a BO of its own. An API-to-Native-API wrapper is used when a component communicates with an external legacy system that is accessed only by components in the same BO or that will eventually become a part of the same BO when modernized.

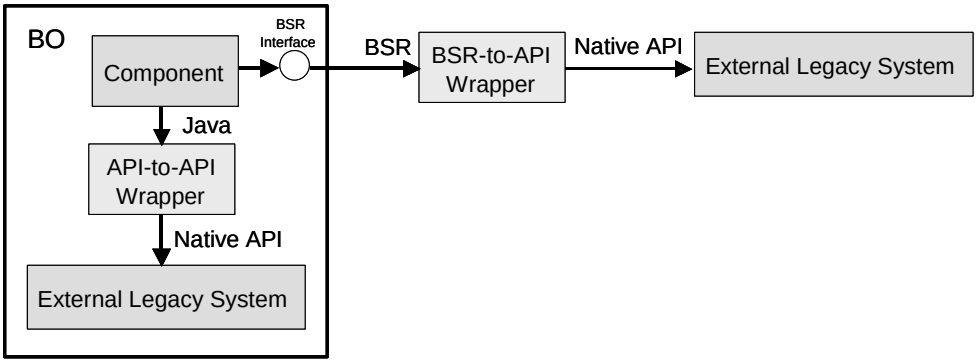


Figure 10. Types of Wrapper Components

3.1.4 BSR interface

The BSR interface provides service for communication between BOs. The layout for most OAGIS-compliant applications is

1. a layer that receives information in an OAGIS XML BOD from a messaging system or messaging framework (e.g., MOM, JMS, RMI, SOAP). A requirement for this system is that the messaging system be MQSeries.
2. a layer that interprets the message and typically includes an XML parser. Newer applications interface with the parser to retrieve and provide information. In older (legacy) ap-

plications, this is done via an adapter layer. These adapters can be thin or thick depending on the amount of processing to be done.

3. an application layer that is the actual application that performs the value-added processing

The BSR interface package must be able to receive BSRs, parse the BODs contained in the BSRs, and translate the BODs into calls to the service components. It also has to generate BODs and send BSRs when requested from a service component or as a confirmation/reply to a BSR.

The parts of a BSR interface as shown in Figure 11 include

- BSR interpreter: The BSR interpreter receives XML messages, parses them, and generates the appropriate calls to the service components inside the BO.
- BSR constructor: The BSR constructor receives requests from service components to construct BSRs to send to other BOs. The BSR constructor can also be called from the BSR interpreter if the incoming BSR requires another BSR to be generated as a confirmation or reply.
- XML schema parser: This is any third-party XML schema parser. Java, for example, has XML parser and builder classes in the com.sun.xml.parser and com.sun.xml.tree packages.
- DTD (data type definition) repository: The DTD repository contains the DTDs for all the BODs defined by the OAGIS. This repository resides in the database or on any file system.

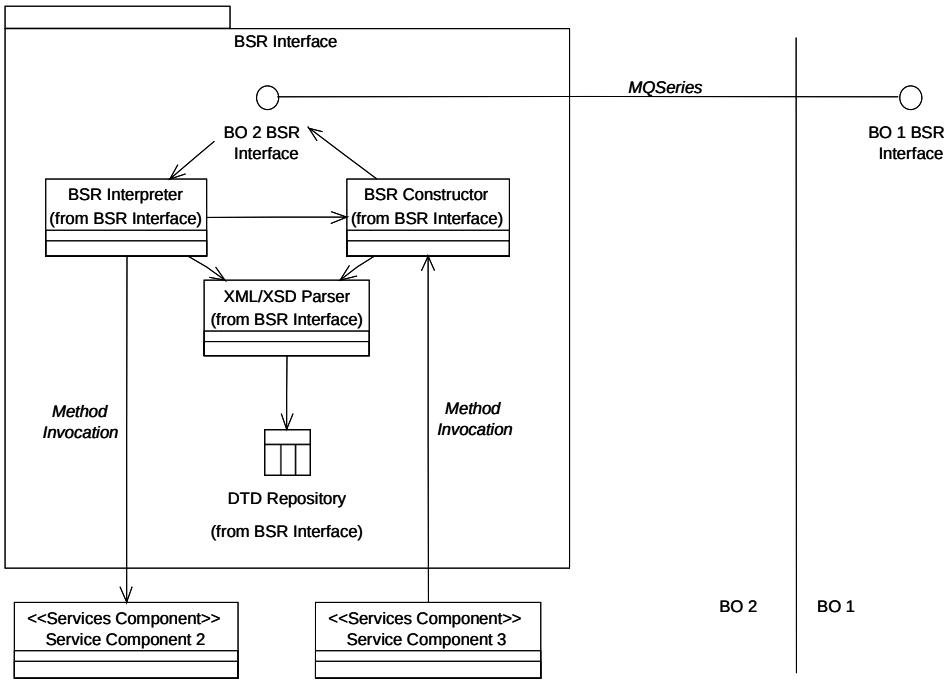


Figure 11. Elements of a BSR Interface Package

As shown in Figure 12, the following operations take place when the BSR interface receives a BSR.

1. The BSR interface of BO 1 sends a BSR over MQSeries to the BSR interface of BO 2.
2. The BSR interface of BO 2 invokes the BSR interpreter that knows how to handle the incoming BSR.
3. The BSR interpreter invokes the XML schema parser to extract the information from the BOD associated with the incoming BSR.
4. The XML parser obtains the DTD from the DTD repository.
5. The BSR interpreter invokes the service component that knows how to process the incoming BSR and passes it the information that was extracted from the BOD.

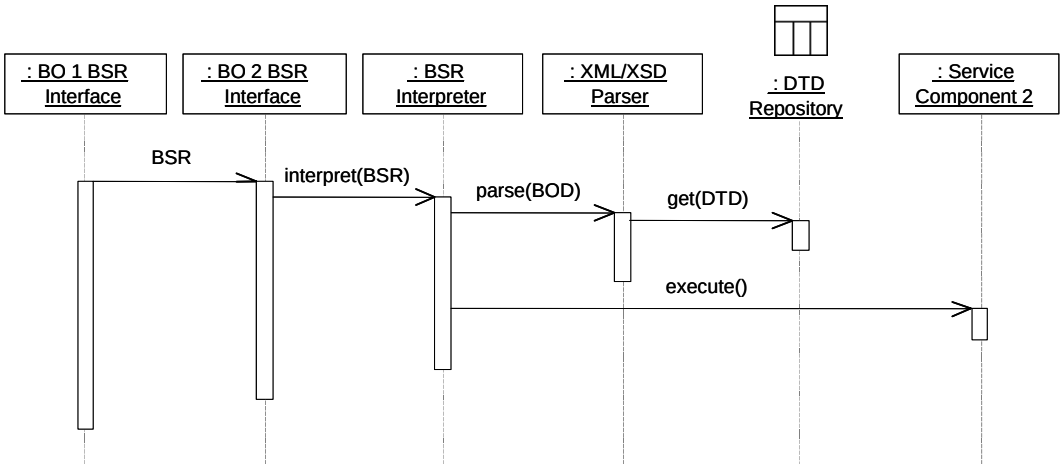


Figure 12. Processing of an Incoming BSR

As shown in Figure 13, the following operations take place when a service component invokes the service of a BSR interface for communication with another BO.

1. The service component in BO 2 that needs to communicate with BO 1 sends a request for BSR construction to the BSR constructor.
2. The BSR constructor invokes the XML schema parser to construct the BOD to be sent in the BSR.
3. The XML parser obtains the DTD from the DTD repository.
4. The BSR constructor requests the BSR interface of BO 2 to send the BSR to BO 1.
5. The BSR interface of BO 2 sends the BSR to BO 1.

If a BSR is required as a reply to an incoming BSR, the BSR interpreter requests the services of the BSR constructor, and the operations in steps 2 through 5 above take place.

Another possibility is to have the BSR constructor communicate with the BSR interface of BO 1 directly. The problem with this is that you have two components that interact through MQSeries (or are aware of how to do it) instead of having just one. The BSR interfaces know how to receive a message through MQSeries and send the BSR (or BOD) to the interpreter, and they know how to take a BSR (or BOD) and bundle it up into a message for MQSeries.

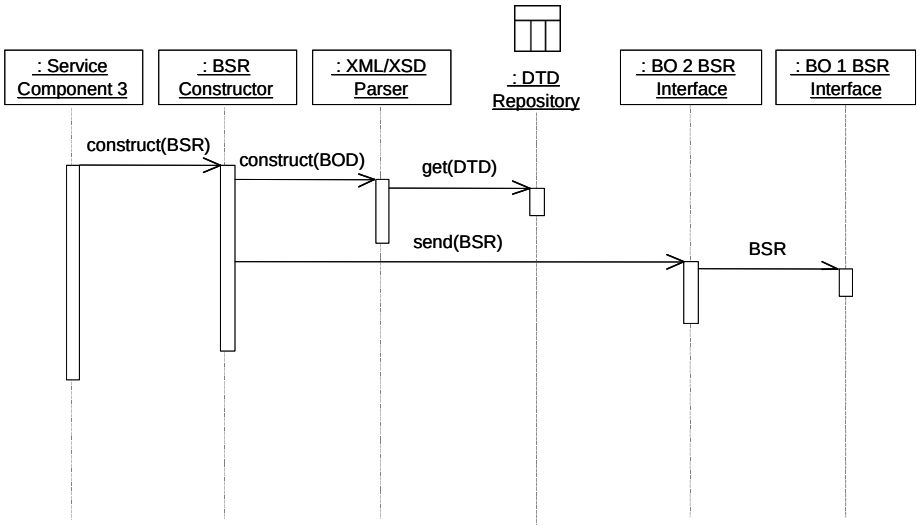


Figure 13. Processing of an Outgoing BSR

4 Architectural Patterns

Architectural patterns are templates that represent generic functions required by the system. They should be used as a guide for component developers.

The architectural patterns that have been identified in this data architecture are

- access operation involving one BO
- access operation involving more than one BO
- report
- ad hoc query
- batch roll up
- continuously updated roll up
- transaction

Each architectural pattern describes the motivation for using the pattern and a Unified Modeling Language (UML) sequence diagram, followed by explanatory steps.

4.1 Access Operation Involving One Business Object

4.1.1 Motivation

This pattern is used when an operation accesses data that is fully contained within the system. It should not be used if the operation communicates with an external system using BSRs.

The sequence of operations for a simple example that accesses data from only one data component is shown in Figure 14. An operation requiring access to multiple tables would include calls to other data components that have access to the tables.

4.1.2 Sequence Diagram

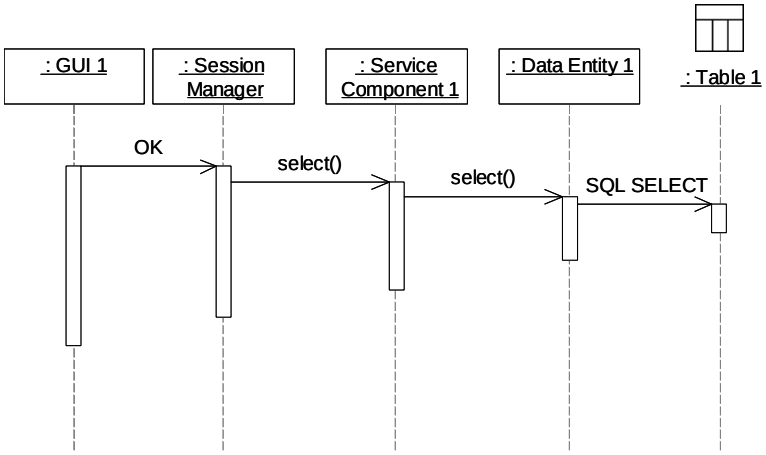


Figure 14. Sequence Diagram for Access Involving One Business Object

4.1.3 Explanatory Steps

The sequence of operations for access involving one BO is as follows:

1. The GUI confirms the operation to the session manager.
2. The session manager invokes the service component that performs the operation.
3. The service component communicates with the data component that has access to the data.
4. The data component executes the necessary SQL command to select the data.

4.2 Access Operation Involving More Than One Business Object

4.2.1 Motivation

This pattern is used when an operation accesses data that is not fully contained within the system and access to an external system is needed to complete the operation.

The sequence of operations for a simple example that communicates with an external system is shown in Figure 15. An operation requiring access to multiple tables would include calls to other data components with the appropriate access.

This pattern assumes that the BSR that returns the data requested by the system exists. If not, a BSR for this purpose needs to be defined.

4.2.2 Sequence Diagram

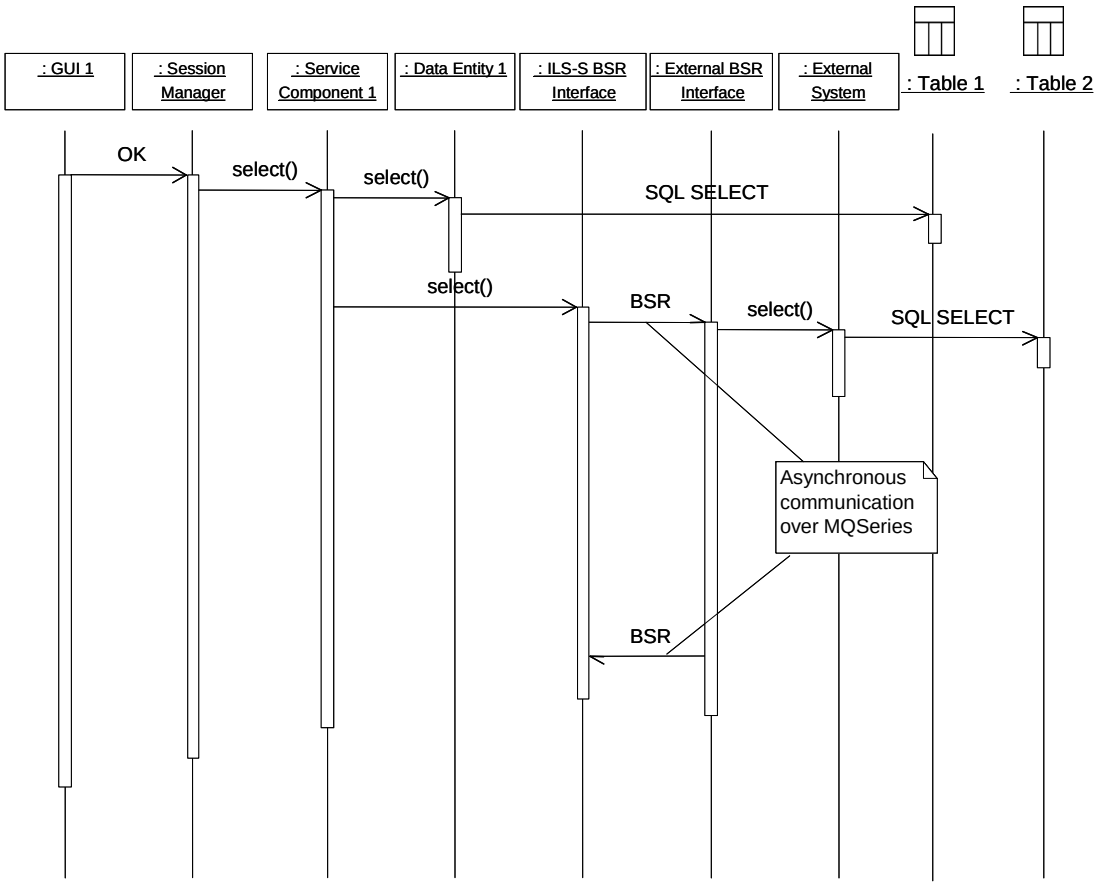


Figure 15. Sequence Diagram for an Access Operation Involving More than One Business Object

4.2.3 Explanatory Steps

The sequence of operations for an access operation involving more than one BO is as follows:

1. The GUI confirms the select operation to the session manager.
2. The session manager invokes the service component that performs the operation.
3. The service component communicates with the data component that has access to the data.
4. The data component executes the necessary SQL command to obtain the data.
5. The service component sends a request to the BSR interface, because it has to obtain data from another BO.
6. The BSR interface constructs a BSR and sends it to the BSR interface of the second BO.

7. The BSR interface of the second BO invokes the service component that performs the operation.
8. The service component communicates with the data component that has access to the data.
9. The data component executes the necessary SQL command to obtain the data.
10. The BSR interface constructs a BSR with the obtained data and sends it to the BSR interface of the first BO.

4.3 Report

4.3.1 Motivation

A report is a formatted and organized presentation of data. The report pattern is used when a report needs to be produced in the system. The output for a report can be printed on paper, written to a file, or sent to an external system, and can be generated from the application component or as part of an operation in a service component.

The report is an exception to the rule of always going through BOs to access data, because SQL is recognized as the best way to handle reports. An example of this is the J2EE Bimodal Data Access pattern that states

> *"Under certain conditions, the Bimodal Data Access pattern allows designers to trade off data consistency for access efficiency. JDBC provides read-only, potentially dirty reads of lists of objects, bypassing both the functionality and overhead of entity enterprise beans. At the same time, entity enterprise beans can still be used for transactional access to enterprise data. The mechanism to select depends on the requirements of the application" [J2EE 01b].*

Reports use the JDBC-based reporting layer for access to data used only for display, since in this case transactional support is unnecessary. (When the application needs to update the database transactionally, it uses enterprise beans.) The sequences of operations are shown in Figure 16 and Figure 17. The report output is not represented in the sequence diagrams but should be defined in the report.

Another option that could be considered for reports is to have stored procedures in the database. These procedures can be written in the SQL programming language as either PL/SQL Stored procedures or Java Stored procedures. The problem with this option is that it makes the reports database dependent. If the database changes, all the procedures stored in the database would have to be ported to the new database. An additional problem is that even though most databases support standard SQL, some have enhancements or additional features that, for example, work only for that particular database and can be used only inside PL/SQL. If this is the case, the procedures will not work when ported to the new database. The new database may not support Java Stored procedures.

4.3.2 Sequence Diagram 1

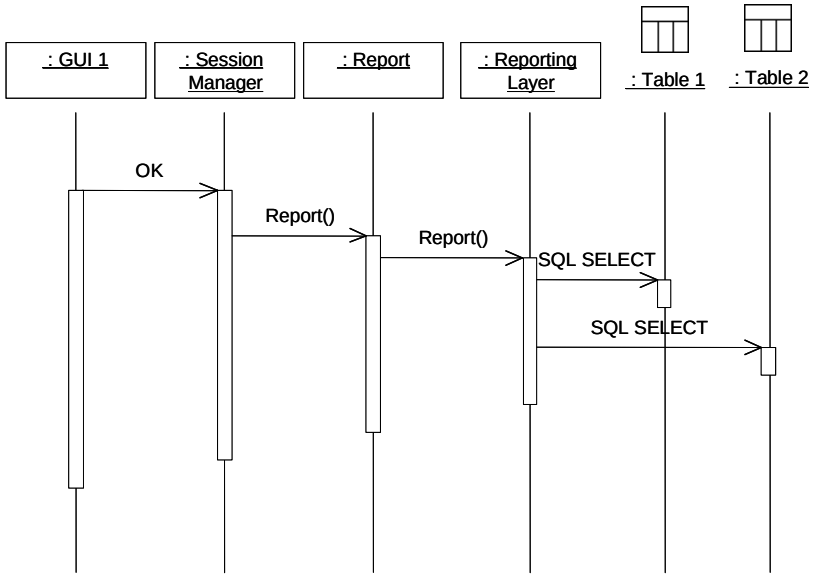


Figure 16. Sequence Diagram for a Report Executed from an Application Component

4.3.3 Explanatory Steps for Sequence Diagram 1

The sequence of operations for a report executed from an application component is as follows:

1. The GUI confirms the report operation to the session manager.
2. The session manager invokes the report script.
3. The report script invokes the Java program (which uses JDBC to access the database) for the report in the reporting layer.
4. The reporting layer obtains the information from the tables in the database.

4.3.4 Sequence Diagram 2

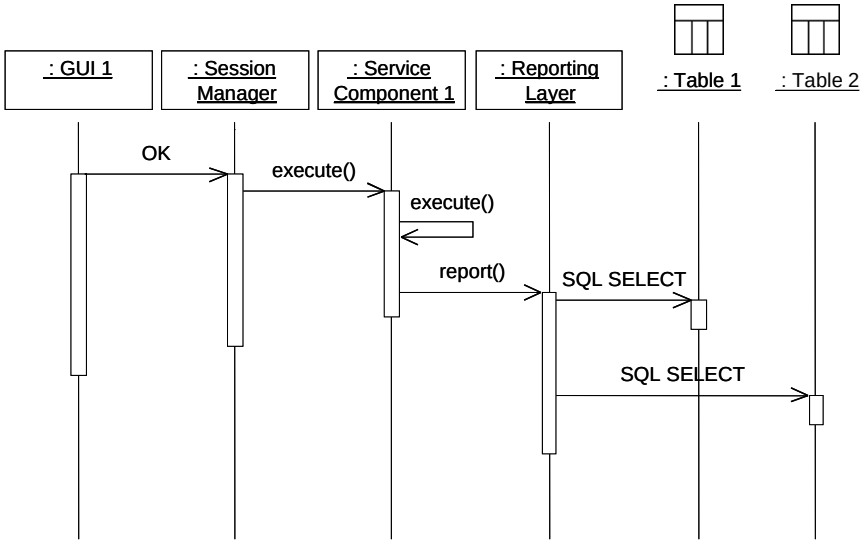


Figure 17. Sequence Diagram for a Report Executed from a Service Component

4.3.5 Explanatory Steps for Sequence Diagram 2

The sequence of operations for a report executed from a service component is as follows:

1. The GUI confirms the operation to the session manager.
2. The session manager invokes the service component that performs the operation.
3. The service component executes the operation.
4. The service component invokes the Java program (which uses JDBC to access the database) for the report in the reporting layer.
5. The reporting layer obtains the information from the tables in the database.

4.4 Ad Hoc Query

4.4.1 Motivation

This pattern is used when a user needs to obtain information from the database and there is no predefined report or operation that returns the data in the desired form. Ad hoc queries are also an exception to the constraint of going through BOs to access data. The sequence of operations for an ad hoc query using a generic, query-builder tool is shown in Figure 18. Another option for entering ad hoc queries is using SQL directly.

4.4.2 Sequence Diagram

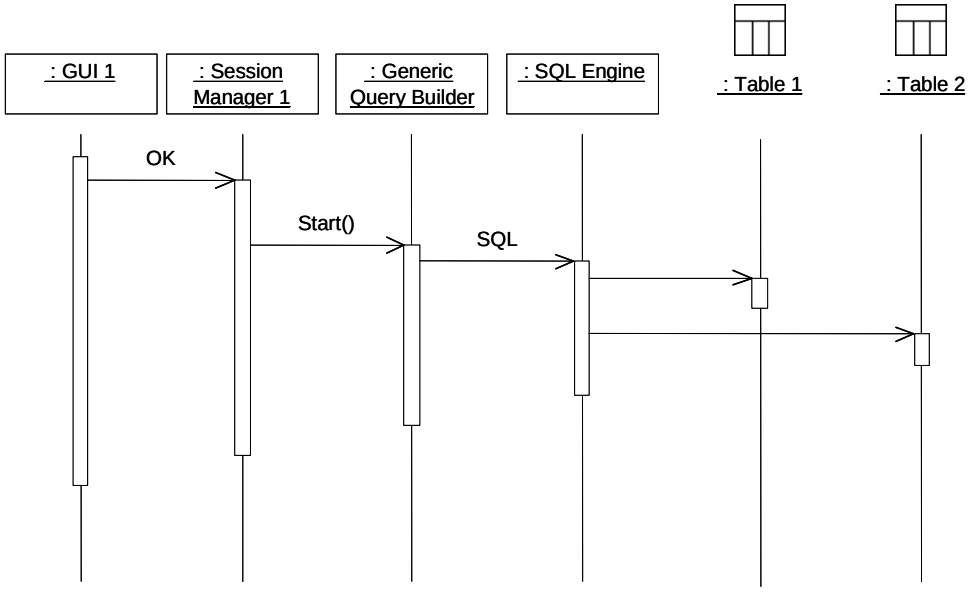


Figure 18. Sequence Diagram for an Ad Hoc Query

4.4.3 Explanatory Steps

The sequence of operations for an ad hoc query is as follows:

1. The GUI confirms the ad hoc query to the session manager.
2. The session manager invokes the generic query builder.
3. The generic query builder sends the query entered by the user to the SQL engine.
4. The SQL engine retrieves the results from the tables in the database.

4.5 Roll Ups

Roll ups are persistent reports (also called summaries) that require the consolidation of data from one or more tables, potentially located on different machines or even different sites. There are two categories of roll ups:

1. on the fly: These are generated immediately and therefore treated as reports.
2. persistent: These require roll-up tables that must be updated to maintain their data in sync with operational data. Roll-up tables have an associated data component, as shown in Figure 19, that is located inside a BO.

The roll-up tables are either batch updated or continuously updated, but the actual generation of the roll up is a report and is treated as such.

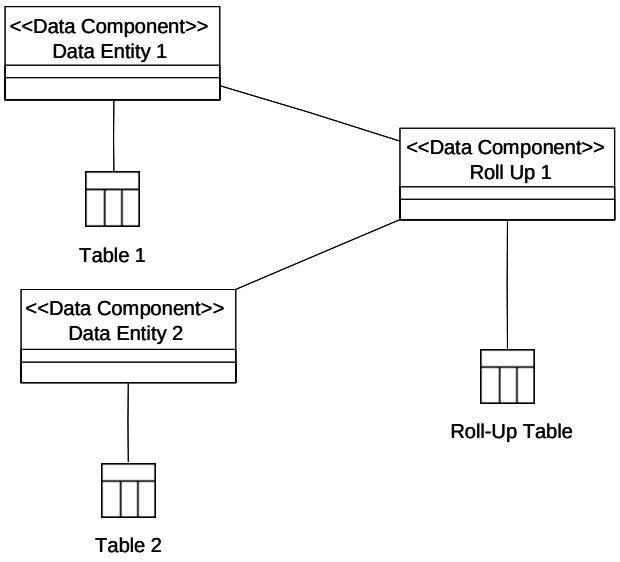


Figure 19. Roll-Up Data Component

4.5.1 Batch Roll Up

The batch roll-up pattern is used when an operation requires data to be extracted from different sources and consolidated in a persistent table, and this table does not need to be immediately synchronized with the tables from which it obtains its data. A procedure is executed on demand to synchronize the roll-up table with its related tables. Figure 20 shows a sequence diagram for a roll-up table that needs to synchronize with only one other table. If the number of related tables is greater, a select operation has to be performed against each of the tables to obtain the necessary data.

4.5.1.1 Sequence Diagram

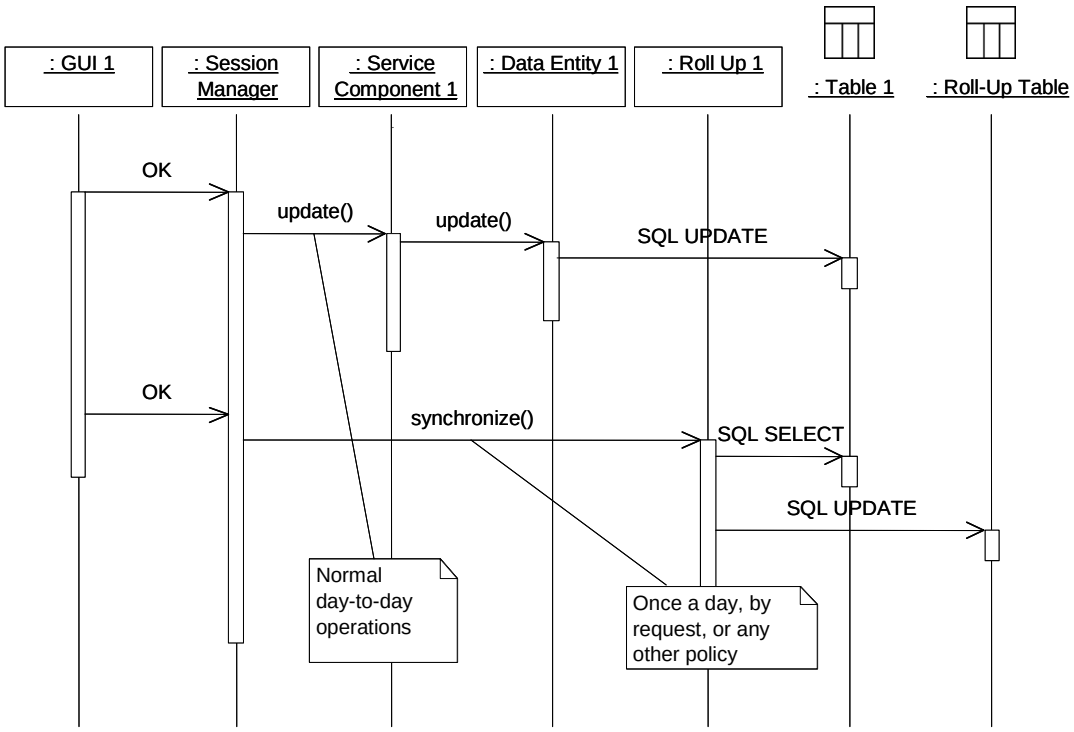


Figure 20. Sequence Diagram for a Batch Roll Up

4.5.1.2 Explanatory Steps

The sequence of operations during online transaction processing (day-to-day operations) is as follows:

1. The GUI confirms the update operation to the session manager.
2. The session manager invokes the service component that performs the operation.
3. The service component communicates with the data component that has access to the data.
4. The data component executes the necessary SQL command to update the data.

The sequence of operations during processing once a day, by request, or any other policy is as follows:

1. The GUI confirms the synchronization operation to the session manager.
2. The session manager sends a synchronization request to the roll-up data component.
3. The roll-up data component obtains the necessary data from the table associated with the roll-up table.
4. The roll-up data component updates the roll-up table.

4.5.2 Continuously Updated Roll Up

4.5.2.1 Motivation

This pattern is used when an operation requires data to be extracted from different sources and consolidated in a persistent table, and this table needs to be immediately synchronized with the tables from which it obtains its data.

In a continuously updated roll up, the roll-up table has to be updated every time any of its associated tables is updated. This operation is based on the Subject-Observer pattern [Gamma 95]. In this case, the roll-up table acts as the *observer* by registering an interest in knowing when its associated tables have been updated. The associated tables are the *subject*. After this, every time an update occurs, the roll-up table is notified.

Figure 21 shows a sequence diagram for a roll-up table that synchronizes with only one table. If the number of related tables is greater, a select operation has to be performed against each of the tables to obtain the necessary data.

4.5.2.2 Sequence Diagram

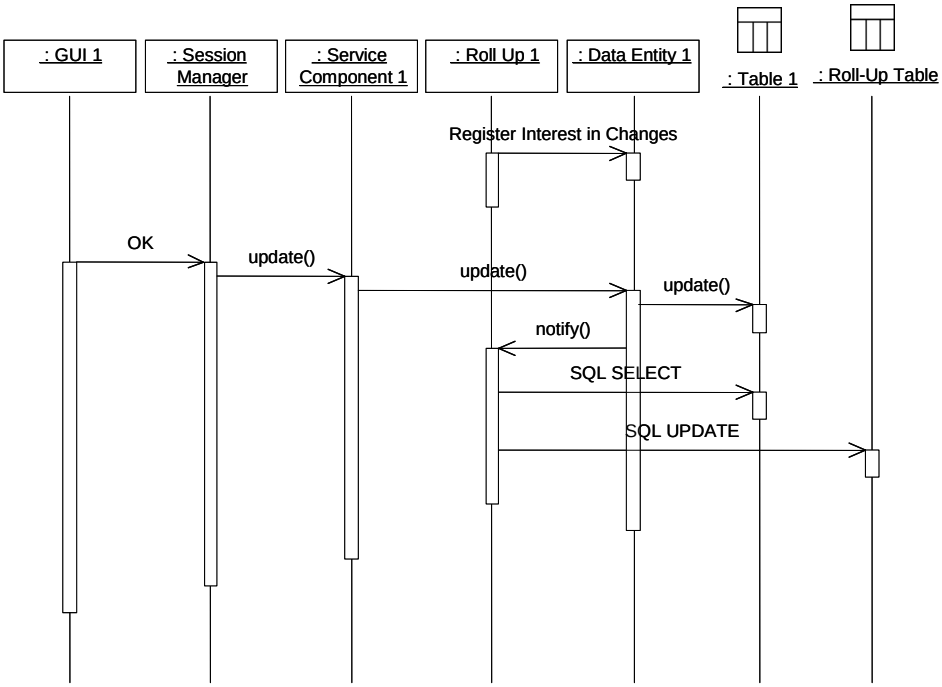


Figure 21. Sequence Diagram for Continuously Updated Roll Up Using the Subject-Observer Pattern

4.5.2.3 Explanatory Steps

During normal day-to-day operations, once the roll-up data component registers an interest in changes made to Table 1 by sending a registration message to the data component associated to the table, the sequence of operations is as follows:

1. The GUI confirms the update operation to the session manager.
2. The session manager invokes the service component that performs the operation.
3. The service component communicates with the data component that has access to the data.
4. The data component executes the necessary SQL command to update the data.
5. The data component notifies the roll-up data component that there has been an update.
6. The roll-up data component obtains the necessary data from the table associated with the roll-up table.
7. The roll-up data component updates the roll-up table.

The operations that take place after the notification can be considered part of the transaction or can be executed outside of the transaction if the response time is inadequate.

4.6 Transactions

4.6.1 Motivation

This pattern is used when an operation updates data that is entirely contained within the system. Transactions that span over two BOs are not covered in this pattern because the transaction context might not be able to be maintained between components that use message-based communication. Also, because of the asynchronous nature of message-based communication, the length of time required to perform the transaction across components can vary. This is the reason why loosely coupled applications are often limited to transient data exchange and messaging between systems across organization boundaries.

A transaction is an *atomic* operation that follows the ACID properties for transaction-processing systems.[4] A transaction involves two or more operations on the database, where

[4] ACID is an acronym for the four properties of transaction-processing systems: Atomicity, Consistency, Isolation, and Durability. Atomicity refers to the principle that the update operations done by a transaction on a database are *atomic* (i.e., either all or none of the operations are done). Consistency is the property of the application that requires any execution of transactions to take the database from one consistent state to another. Isolation pertains to the extent to which operations done upon data by one transaction are *seen* by or protected from a different transaction or query running concurrently. Durability is the property of a system to preserve the effects of committed transactions and ensure database consistency after recovery from certain types of system failures.

either all or none of the operations are done. A commit operation (where all changes are kept) or a rollback operation (where all changes are removed) ends a transaction.

Figure 22 shows a sequence diagram for a transaction that requires the update of two tables as part of the same transaction. The number of tables updated as part of the transaction is irrelevant, as the EJB framework manages all updates within the same transaction context.

4.6.2 Sequence Diagram

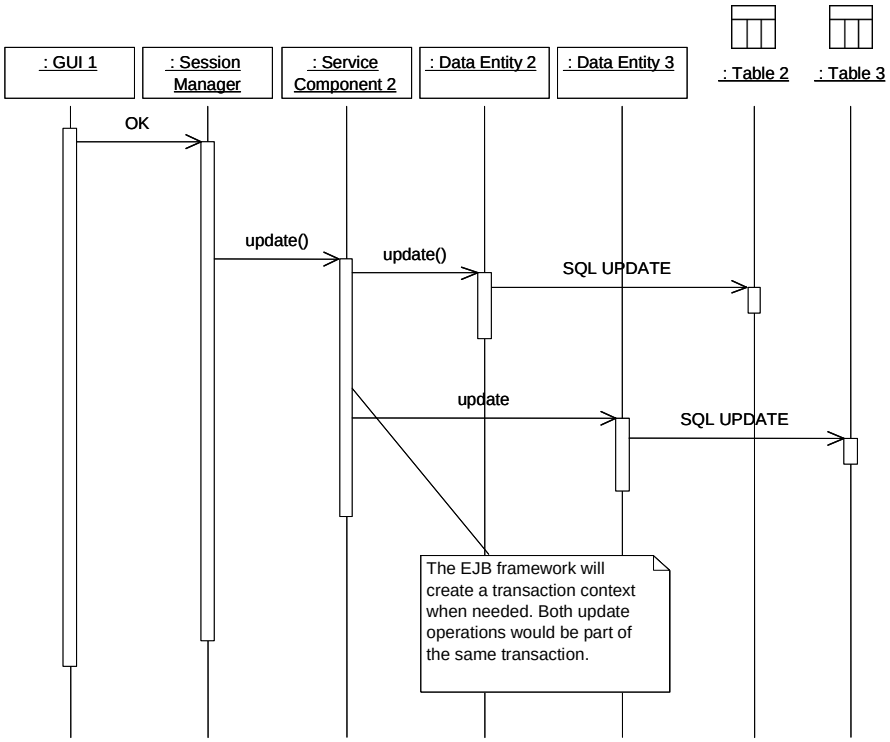


Figure 22. Sequence Diagram for a Transaction

4.6.3 Explanatory Steps

The sequence of operations for a transaction is as follows:

1. The GUI confirms the operation to the session manager.
2. The session manager invokes the service component that performs the operation.
3. The EJB framework creates a transaction context.
4. The service component communicates with the first data component participating in the transaction.
5. The first data component executes the necessary SQL command to update the data.
6. The service component communicates with the second data component participating in the transaction.

7. The second data component executes the necessary SQL command to update the data.
8. The EJB framework commits or rolls back the transaction.

4.7 Data Warehouses

This pattern is used to interface to data warehouses and data marts. Because data marts are usually subsets of data warehouses, the data warehouses communicate with the operational database, and then the data marts populate themselves from data in the data warehouse. This pattern is dependent on the specific tool that is used for data-warehouse population.

There are two possibilities to populate data warehouses:

1. pull mechanism: The data-warehouse-population application pulls the data from the operational database either by request or through automated update procedures. In this case an interface to the operational database has to be implemented.
2. push mechanism: The operational database pushes data to the data warehouse either by request or through automated update procedures. In this case an interface to the data-warehouse-population application has to be implemented.

If a pull mechanism is implemented, the functionality for data-warehouse population resides in a third-party data-warehouse-population tool. This tool can be treated as a BO where population operations use BSRs, given that the tool has a BSR-based communication interface, or it can be treated as an application-specific component where population operations use the reporting layer. These two possibilities are shown in Figure 23 and Figure 24.

The population operations that take place inside the BOs are omitted in the pull option for data-warehouse population using BSRs. The sequence of steps in this option is as follows:

1. The data-warehouse-population application communicates with its BSR interface to request a population operation.
2. The BSR interface constructs a BSR and sends it to a BO.
3. The BO obtains the information and uses its BSR interface to construct a BSR that sends the requested information to the data-warehouse-population application.
4. The data-warehouse-population application populates the data warehouse with the obtained data.

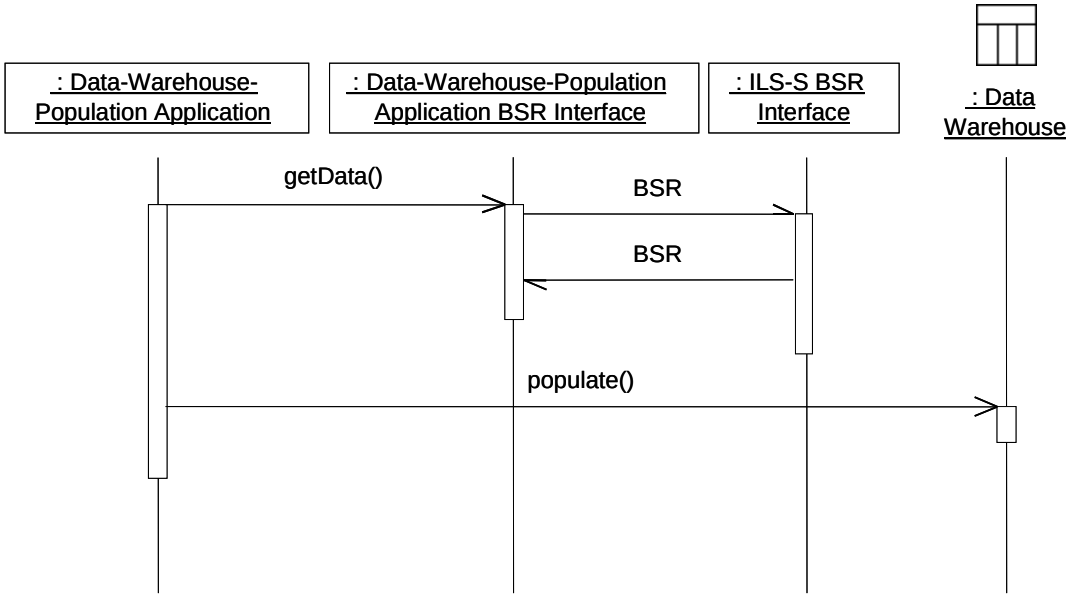


Figure 23. Pull Option for Data-Warehouse Population Using BSRs

The sequence of steps in the pull option for data-warehouse population using the reporting layer is as follows:

1. The data-warehouse-population application communicates with the reporting layer to obtain the data.
2. The reporting layer performs the necessary SQL select operations to obtain the data.
3. The data-warehouse-population application populates the data warehouse with the obtained data.

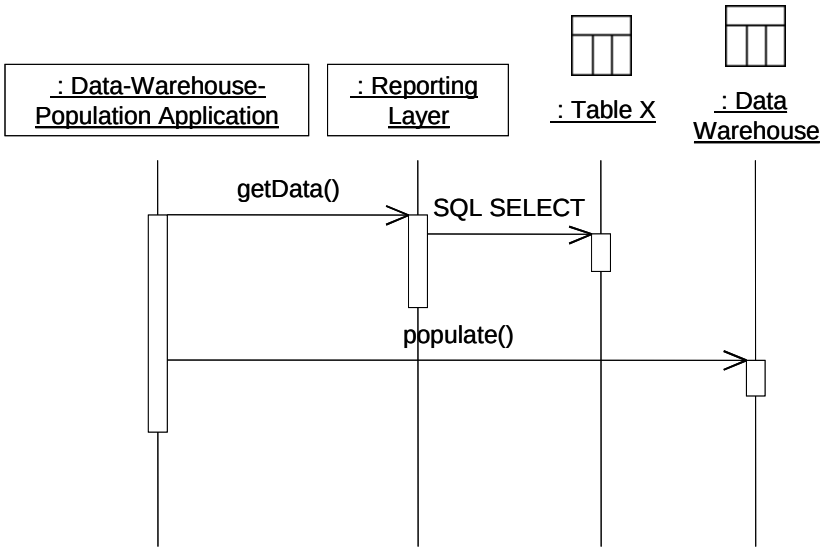


Figure 24. Pull Option for Data-Warehouse Population Using the Reporting Layer

If a push mechanism is implemented, the functionality for data-warehouse population resides within the system. As in the previous option, if the tool has a BSR-based communication interface, communication can take place through BSRs. If this is not a possibility, an adapter for communication with the tool is to be implemented as a wrapper component. These two possibilities are shown in Figure 25 and Figure 26.

The operations that take place inside the BOs are omitted in the push option for data-warehouse population using BSRs. The sequence of steps in this option is as follows:

1. The system uses its BSR interface to construct a BSR that sends information to the data-warehouse-population application.
2. The BSR interface sends the data-warehouse-population application the incoming data.
3. The data-warehouse-population application populates the data warehouse with the incoming data.
4. (Optional) The BSR interface constructs a confirmation BSR and sends it back to the system.

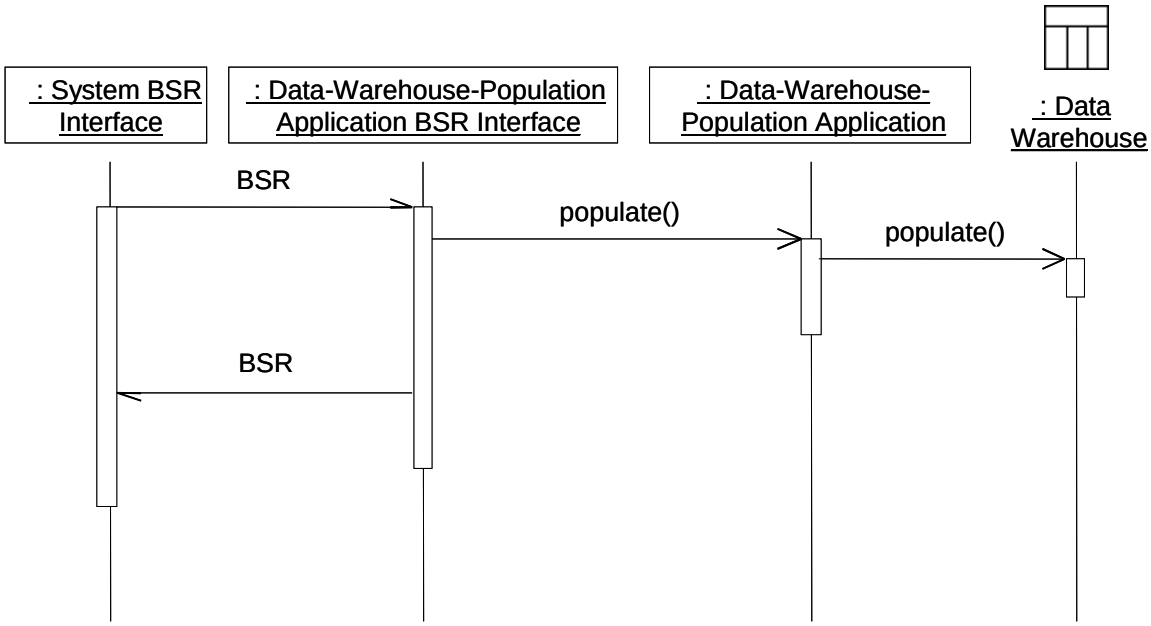


Figure 25. Push Option for Data-Warehouse Population Using BSRs

The operations that obtain the data from the data components in the push option for data-warehouse population using a wrapper component are omitted. For details see Section 4.1 on page 23. The sequence of steps in this option is as follows:

1. A service component sends a request to a wrapper component to populate the data warehouse. Data obtained from the operational database is attached to this request.
2. The wrapper component sends the request and data to the data-warehouse-population application.
3. The data-warehouse-population application populates the data warehouse with the incoming data.

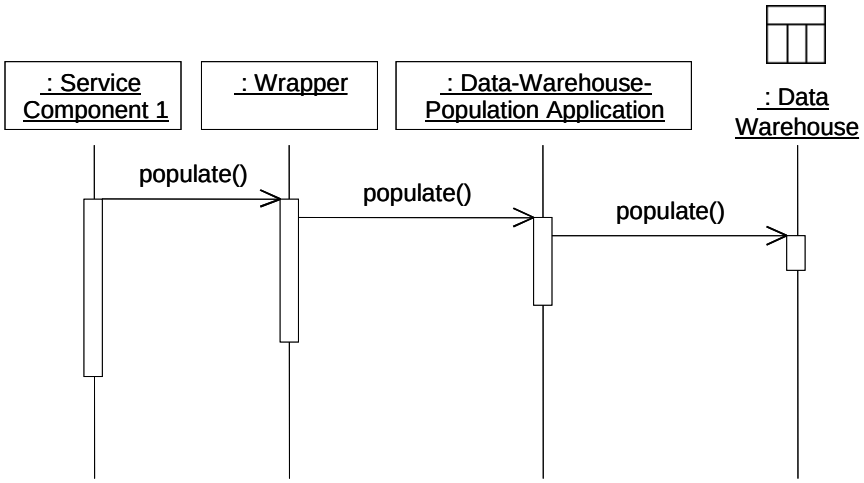


Figure 26. Push Option for Data-Warehouse Population Using a Wrapper Component

5 Examples

This section contains examples taken from RSS use cases. The instantiations of the architectural patterns do not correspond completely to use cases because only data-interaction activities are considered. Aspects like notifications to other users and communication between users are not represented.

The assumption for all of the patterns is that the application component is responsible for validating users, displaying forms, and handling all communication between the user and the system. For simplicity, details on the actual tables managed by the data components and subcomponents are not included.

5.1 Decomposition of a Use Case into Service and Data Components

This example corresponds to the Order Consumable Item use case. This use case's sequence of steps is summarized below and illustrated in Figure 27.

1. The user accesses the order-management application and selects the ordering option for a consumable item.
2. The user is presented with an order form.
3. The user fills out the order form and can obtain a list of items in the catalog if desired.
4. The user submits the order.
5. The system validates the order and communicates with the financial system to check funds as part of the process. If the order is not valid, the user is informed.
6. If the item is available: it is reserved; the Pull Item from Warehouse Location use case is initiated; and the user is presented with an order confirmation.
7. If an item is not available, a list of alternate items is presented. If this is not satisfactory, the Backorder use case is initiated.

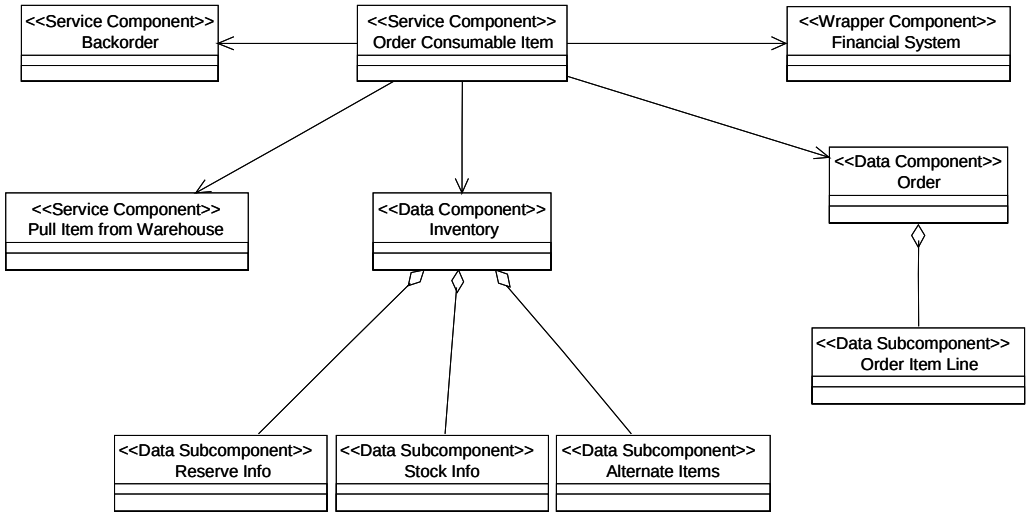


Figure 27. Decomposition of the Order Consumable Item Use Case

In this use case

- Order Consumable Item, Pull Item from Warehouse, and Backorder are service components implemented as stateful session beans, because they require knowledge of the user.
- The Order Consumable Item service component contains the logic for validating an order, creating an order, and reserving an item. If other use cases require reserving an item, a Reserve Item service component can be created.
- Inventory and Order are data components implemented as entity beans and can be used by any service component in the system.
- Reserve Information, Stock Information, and Alternate Items are data subcomponents implemented as Java classes and are subcomponents of the Inventory data component.
- Order Item Line is a data subcomponent implemented as a Java class and is a subcomponent of the Order data component.
- There is a wrapper component for communication with the financial system. If the financial system has a BSR interface, the communication would take place through the BSR interface, making the wrapper component unnecessary.

The Catalog List service component is not part of the Order Consumable Item use case. It can be invoked from the application component and should be implemented as a stateless session bean, because it only displays information and does not require knowledge of the user.

5.2 Access to Information in One Business Object

The Obtaining a Catalog List operation is an example of access to information in one BO. The sequence of steps in this operation is summarized below and illustrated in Figure 28.

1. A user submits a request for a catalog list.

2. The session manager invokes the Catalog List service component.
3. The Catalog List service component requests the catalog information from the Item data component.
4. The Item data component obtains the catalog information from the Item table.
5. The session manager sends the data to the user screen to be displayed.

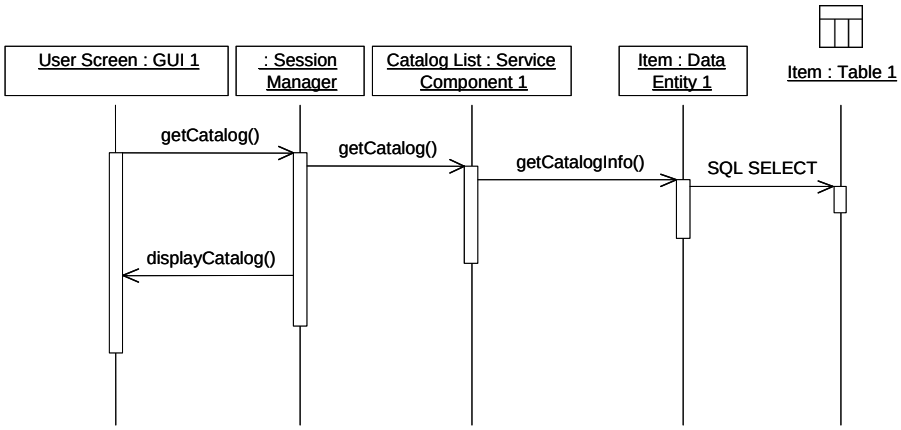


Figure 28. Catalog List Use Case as an Example of Access to Information in One Business Object

5.3 Access to Information in Two Business Objects

This example corresponds to the Manually Prepared Requisition use case. The sequence of steps in this use case is summarized below and illustrated in Figure 29.

1. An authorized supply clerk accesses the order-management application and selects the option for manually entering a requisition.
2. The user is presented with an order form.
3. The user fills out the order form and can obtain a list of items in the catalog if desired.
4. The user submits the order.
5. The system validates the order and communicates with the financial system to check funds as part of the process. If the order is not valid, the user is informed.
6. If the item is available: it is reserved; the Pull Item from Warehouse Location service component is initiated; and the user is presented with an order confirmation.
7. If an item is not available, a list of alternate items is presented. If this is not satisfactory, the Backorder use case is initiated.

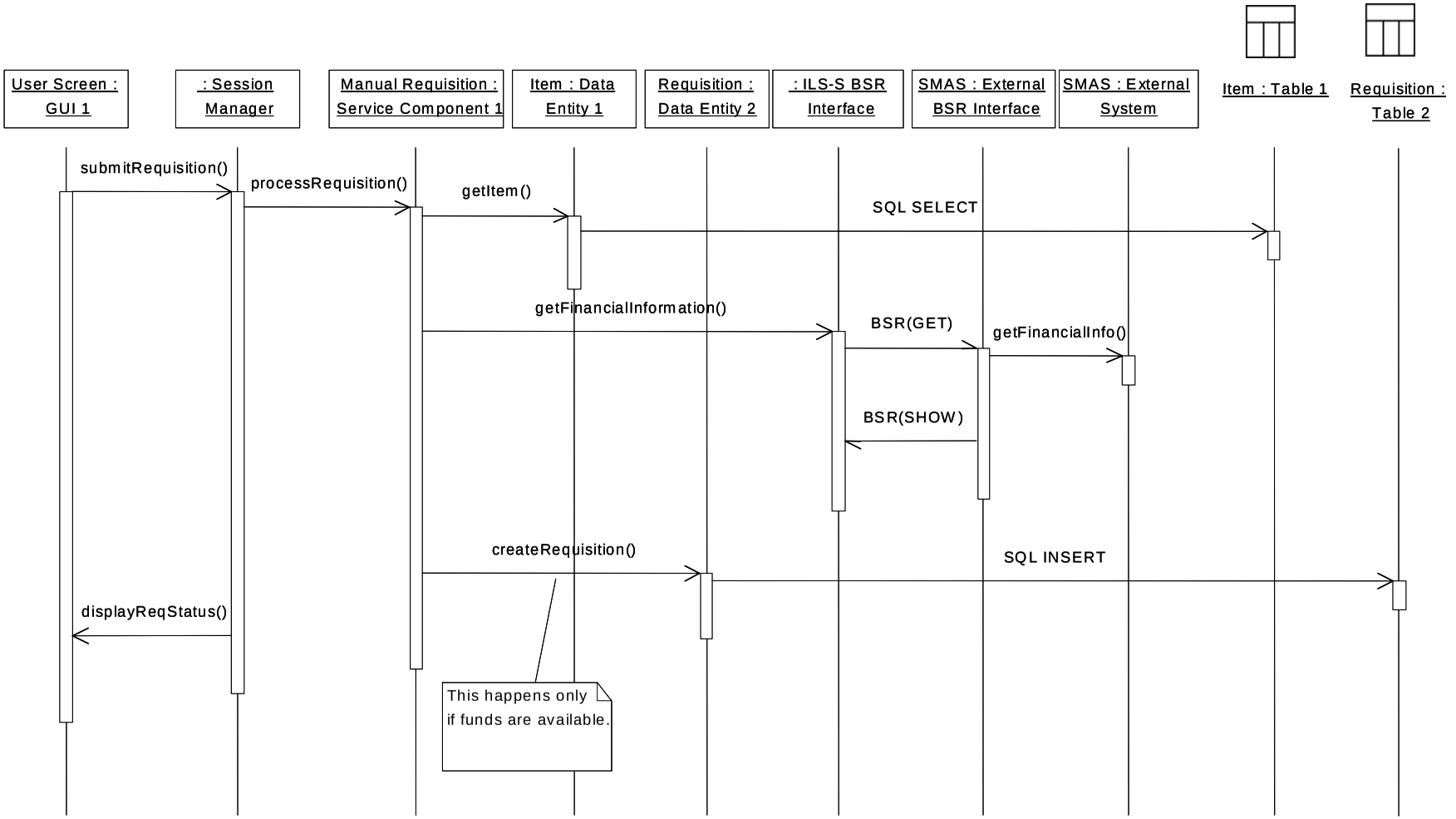


Figure 29. Manual Requisition Use Case as an Example of Access to Information in Two Business Objects

5.4 Report Generated from the User Interface

According to the Manually Prepared Requisition use case, supply customers are allowed to query the status of their organizations' backorders and linked requisitions at any time before the final receipt of the order. This example corresponds to the Backorder Status Report use case. The sequence of steps in this use case is summarized below and illustrated in Figure 30.

1. The supply clerk submits a request for a Backorder Status Report.
2. The session manager invokes the Backorder Status Report script.
3. The script assembles a request to the reporting layer with the user parameters.
4. The Backorder Status Report program in the reporting layer converts the request into SQL statements using JDBC and obtains the data from the database.

   The report can be sent to a printer, a file, or an external system.

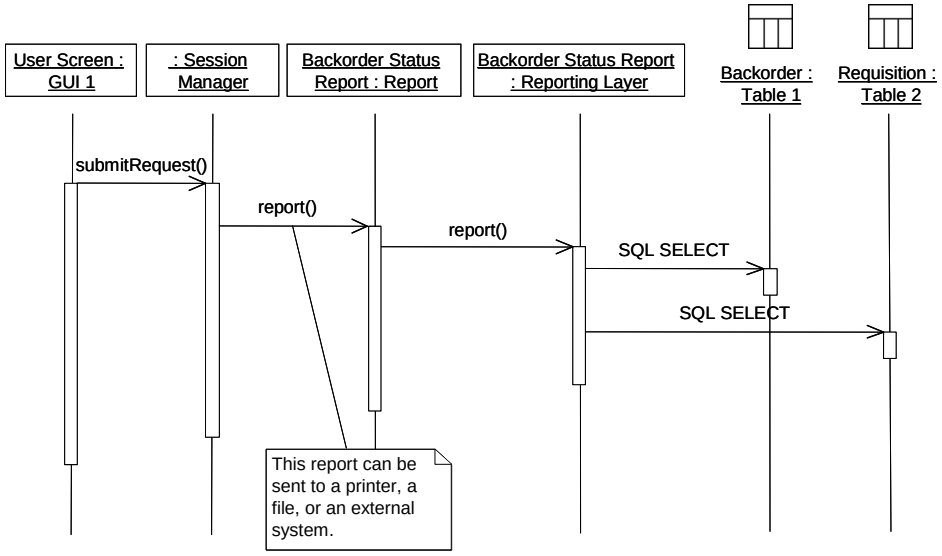


Figure 30. Backorder Status Report as an Example of a Report Generated from the User Interface

5.5 Report Generated from a Service Component

The Ship Exchangeable Item use case is an example of a report that is generated from a service component. After the shipment suspense is created, a hard copy of the shipment documentation is generated and attached to the property. The sequence of steps in this use case is summarized below and illustrated in Figure 31.

1. An authorized supply clerk accesses the order-management application and selects the option for shipping an exchangeable item.
2. The supply clerk is presented with a shipping information form.
3. The supply clerk fills out the form.
4. The supply clerk submits the form.
5. The session manager invokes the Ship Exchangeable Item service component.
6. The Ship Exchangeable Item service component sends a request to obtain the shipping information for the item to the Item data component.
7. The Item data component obtains the shipping information for the item from the Item table.
8. The Ship Exchangeable Item service component sends a request to decrease the stock for the item to the Inventory data component.
9. The Inventory data component updates the stock information for that item in the Inventory table.
10. The Ship Exchangeable Item service component sends a request to create a shipment suspense to the Shipment Suspense data component.
11. The Shipment Suspense data component creates the shipment suspense in the Shipment Suspense table.
12. The Ship Exchangeable Item service component sends a request for a Shipment Documentation Report to the reporting layer.
13. The Shipment Documentation Report program in the reporting layer converts the request into SQL statements using JDBC and obtains the data from the database.

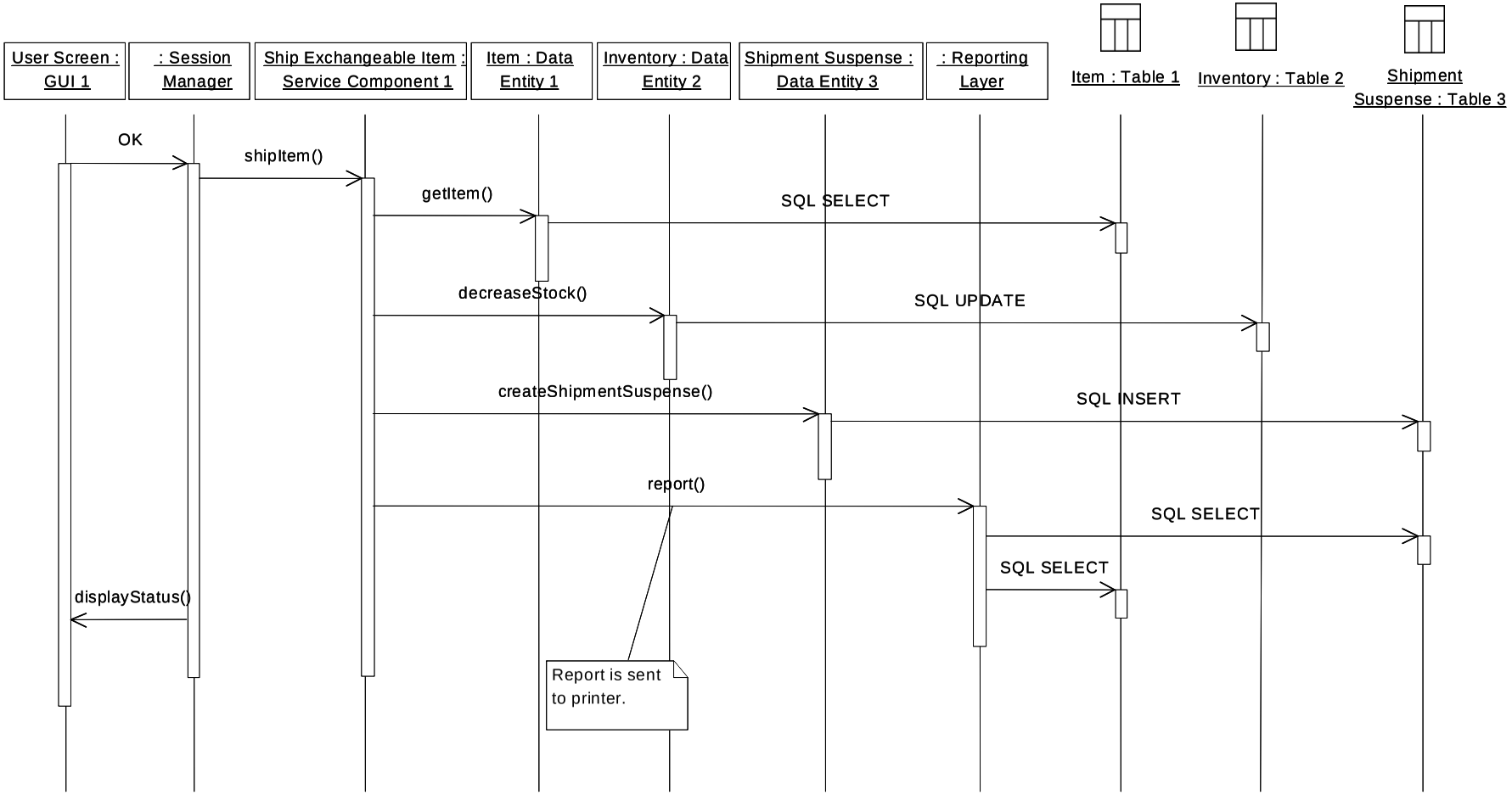


Figure 31. Ship Exchangeable Item as an Example of a Report Generated from a Service Component

5.6 Ad Hoc Query

An ad hoc query tool provides users with "on-demand" query and reporting capabilities. An example of an ad hoc query could be "I need to know how many units of item X have been transferred to Office Y in the past Z weeks." The sequence of steps in this operation is summarized below and illustrated in Figure 32.

1. The user submits a request for an ad hoc query to the session manager.
2. The session manager invokes the ad hoc query tool.
3. The ad hoc query tool allows the user to assemble the ad hoc query and submits it to the database SQL engine.
4. The SQL engine retrieves the necessary data from the tables involved.

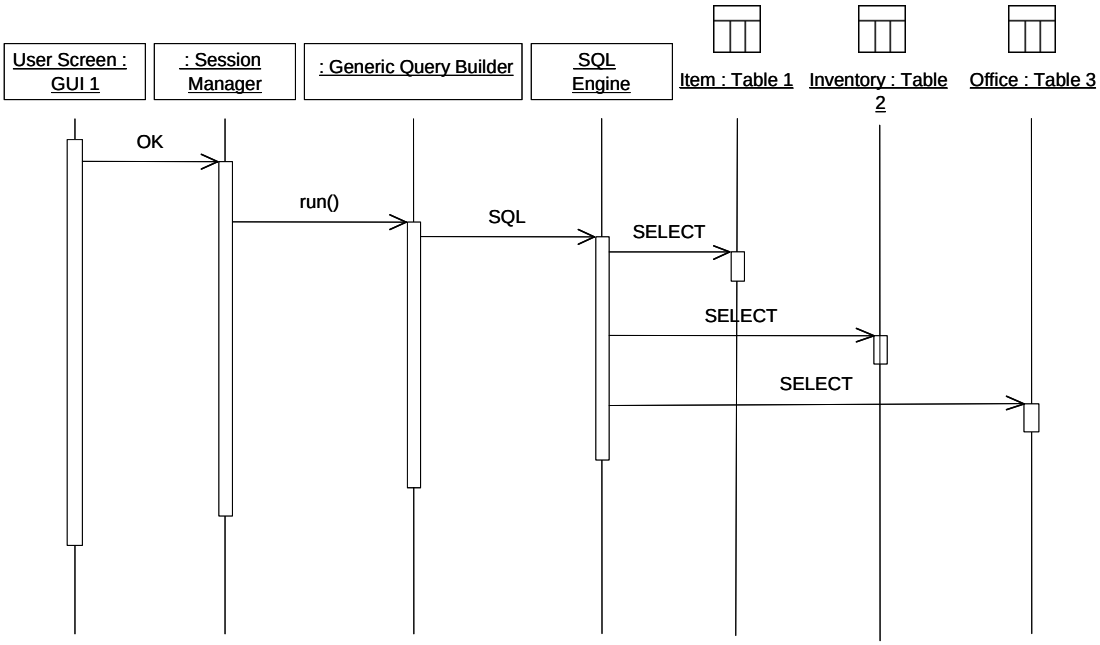


Figure 32. Ad Hoc Query Example

To avoid the problems of not being able to find data or having a tool that is too difficult for end users, ad hoc query tools (such as Oracle Discoverer) provide an End User Layer™ (EUL), which is a metadata repository and query management engine. The EUL presents users with a business view of their data and shields the complexity of database structures. End users interact with their data using familiar business terminology, not cryptic database terminology. If this is the case, the metadata repository becomes an element of this architectural pattern.

5.7 Batch Roll Up

Batch roll ups are typical in the current system. An example of a batch roll up is Outstanding Orders. Orders that are, for example, seven days old and have not been delivered can be rolled up into an Outstanding Order Roll-Up data component. This data can then be used in reports, queried by users of other systems, transferred to other systems, or kept for long-range demand planning. The sequence of steps for the generation of an Outstanding Orders batch roll up is summarized below and illustrated in Figure 33. It is combined with the Order Consumable Item use case to show the relationship between the roll-up table and its associated table.

During online transaction processing (day-to-day operations), these steps are taken:

1. The user accesses the order-management application and selects the ordering option for a consumable item.
2. The user is presented with an order form.
3. The user fills out the form.
4. The user submits the form.
5. The session manager invokes the Order Consumable Item service component.
6. As part of its operations, the Order Consumable Item service component requests the creation of an order from the Order data component.
7. The Order data component creates the order in the database.

Once a week, these steps are taken:

1. The user submits a request for Outstanding Orders.
2. The session manager invokes the synchronization procedure in the Outstanding Orders Roll-Up data component.
3. The Roll-Up data component obtains the necessary data from the Order table and creates/updates entries in the Outstanding Orders table.

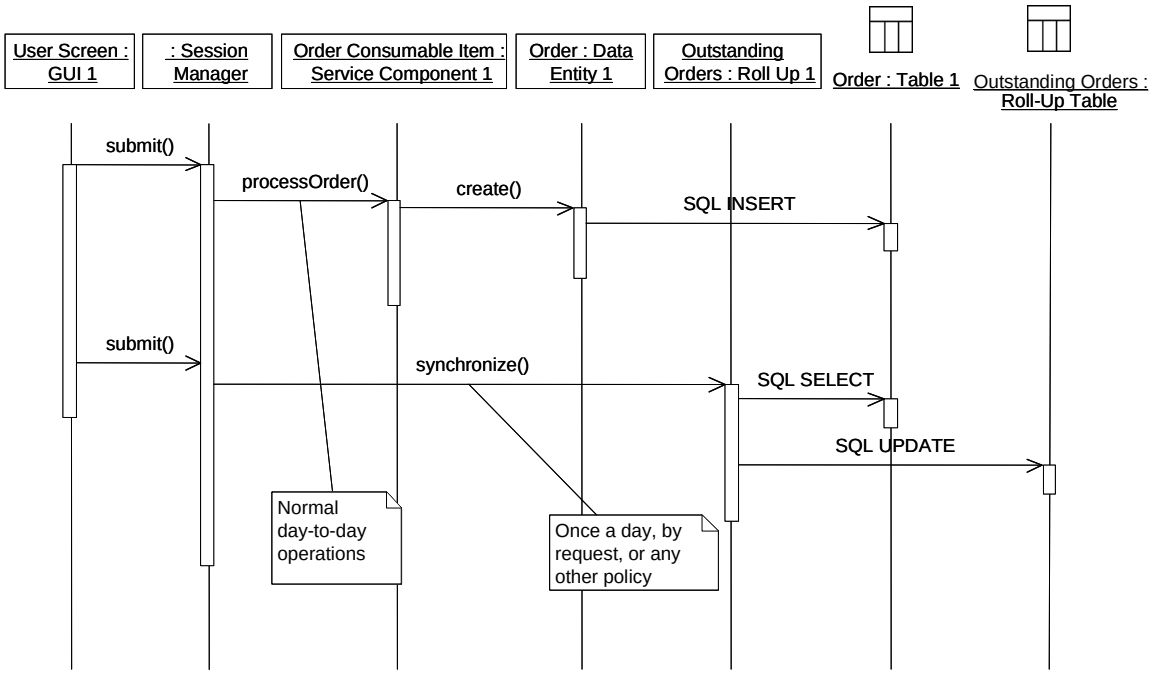


Figure 33. Example of a Batch Roll Up – Outstanding Orders

5.8 Continuously Updated Roll Up

Continuously updated roll ups differ from batch roll ups in that the former must be updated as operations occur in the system, as opposed to waiting for a synchronization operation to occur. An example of a continuously updated roll up is Average Cost of Inventory. The average cost of inventory in the supply system varies with every change in the Inventory table. The average cost of inventory data can be used in reports, queried by users of other systems, transferred to other systems, or kept for long-range demand planning. The sequence of steps for the update of an Average Cost of Inventory continuously updated roll up is summarized below and illustrated in Figure 34. It is combined with the Ship Exchangeable Item use case to show the relationship between the roll-up table and its associated table.

The following step is taken once: the Average Cost of Inventory Roll-Up data component registers an interest in changes made to the Inventory table, by sending a registration message to the Inventory data component associated with the table.

During normal day-to-day operations, these steps are taken:

1. An authorized supply clerk accesses the order-management application and selects the option for shipping an exchangeable item.
2. The supply clerk is presented with a shipping information form.
3. The supply clerk fills out the form.
4. The supply clerk submits the form.

5. The session manager invokes the Ship Exchangeable Item service component.
6. As part of its operations, the Ship Exchangeable Item service component sends a request to decrease the stock for the item to the Inventory data component.
7. The Inventory data component updates the stock information for that item in the Inventory table.
8. The Inventory data component notifies the Average Cost of Inventory Roll-Up data component that there has been an update.
9. The Average Cost of Inventory Roll-Up data component obtains data from the Inventory table (if necessary).
10. The Average Cost of Inventory Roll-Up data component updates the Average Cost of Inventory Roll-Up table.

Steps 8 to 10 can be considered part of the transaction or can be executed outside of the transaction if the response time is inadequate. The component developer can decide which will be done based on performance requirements.

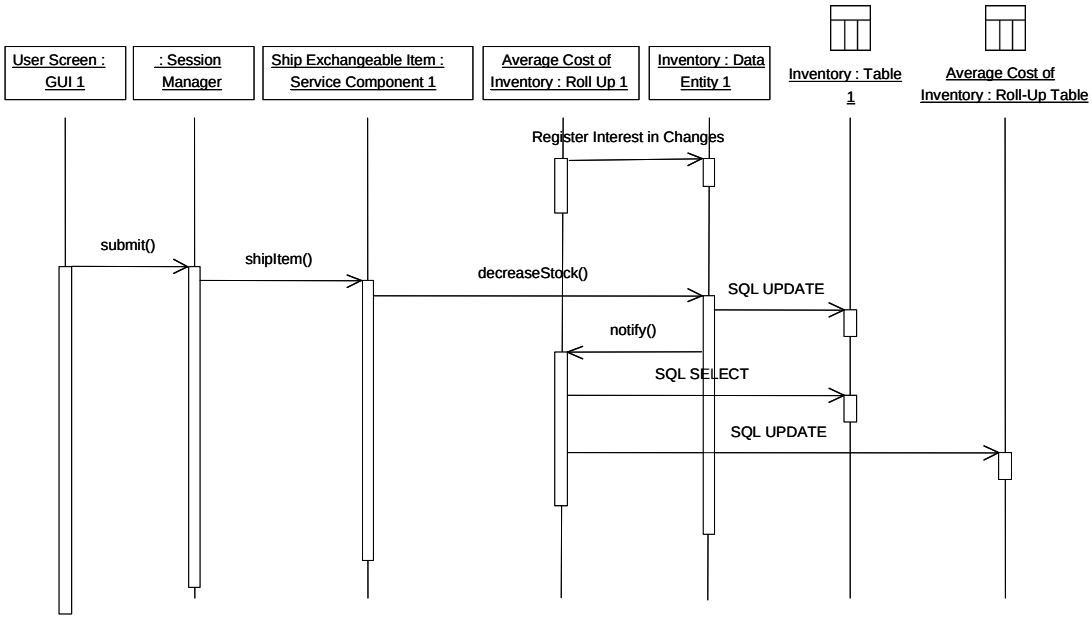


Figure 34. Example of a Continuously Updated Roll Up – Average Cost of Inventory

5.9 Transaction

This example corresponds to the Delivery – Customer Pickup use case. The sequence of steps in this use case is summarized below and illustrated in Figure 35.

1. An authorized delivery clerk accesses the order-management application and selects the option for delivering an item to a customer.
2. The delivery clerk is presented with a delivery information form.
3. The delivery clerk fills out the form.
4. The delivery clerk submits the form.
5. The session manager invokes the Delivery for Customer Pickup service component.
6. The EJB framework creates a transaction context.
7. The Delivery for Customer Pickup service component sends a request to update the warehouse-tracking information to the Warehouse Tracking data component.
8. The Warehouse Tracking data component updates the warehouse-tracking information in the Warehouse Tracking table.
9. The Delivery for Customer Pickup service component sends a request to decrease the stock for the delivered item to the Inventory data component.
10. The Inventory data component updates the stock information for that item in the Inventory table.
11. The EJB framework commits or rolls back the transaction.
12. The delivery clerk is informed of the transaction's status.

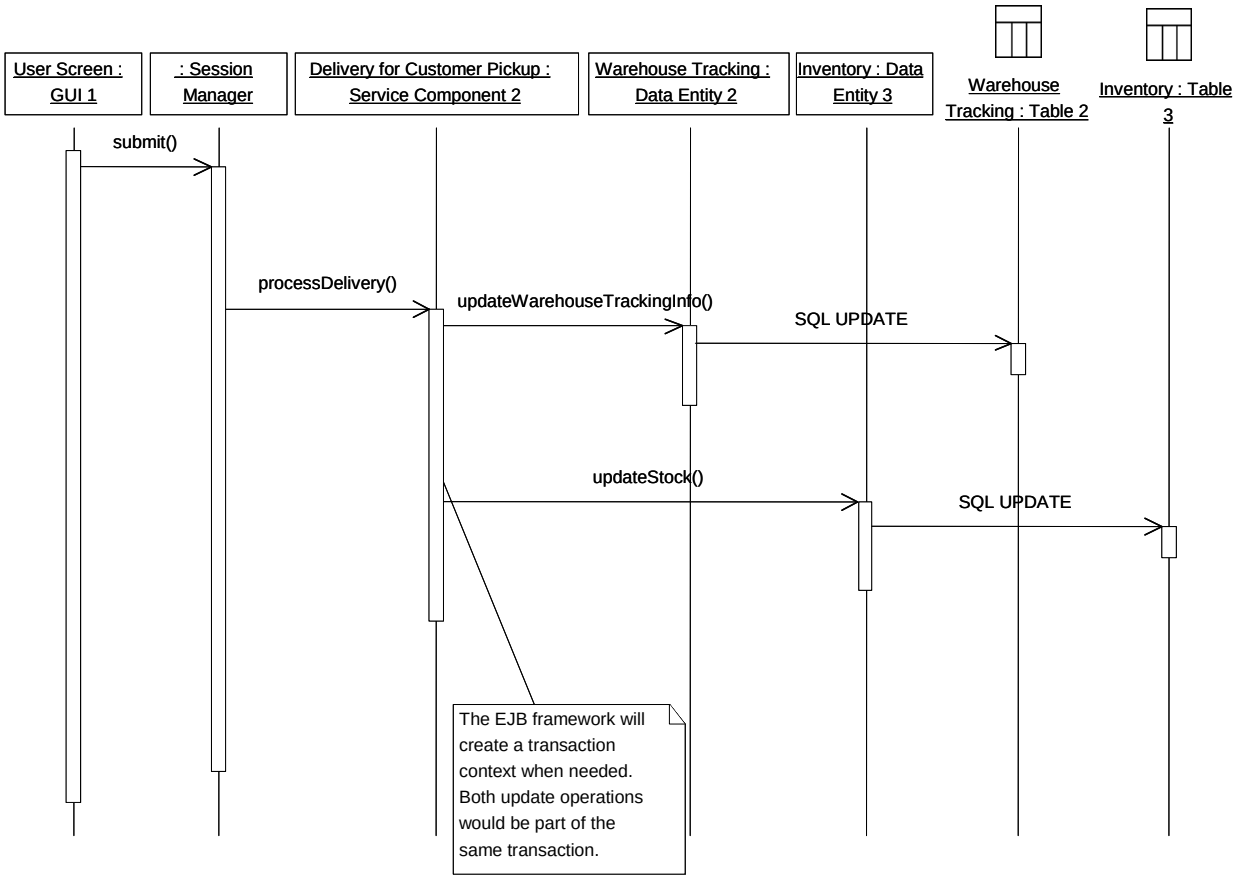


Figure 35. Delivery for Customer Pickup as an Example of a Transaction

5.10 Data Warehousing

A specific example of data warehousing will not be included here, because it depends on the tool that will be used for data-warehouse population. Refer to Section 4.7 on page 35 for examples of communication between the system and a data-warehousing application.

6 Conclusions

The architectural patterns included in this guide provide templates to create a data architecture for a system based on the J2EE platform and the OAGIS. This guide can also be used to identify and resolve potential design risks resulting from inconsistent or contradictory requirements.

The application of this guide should result in a system with the following characteristics:

- fulfillment of the data requirements
- capability of communication with other BOs through a BSR interface
- compliance with the given technical requirements
- decoupling of data components from data representation
- conservation of ACID properties for transactions provided by the EJB framework
- use of J2EE design patterns representing best practices

As always, sound engineering judgment should be used in applying the architectural patterns included in this guide, since it is difficult (if not impossible) to predict all possible scenarios. This guide should be maintained and updated to reflect lessons learned during early iterations of the development process.

7 Acronyms

A2A	application to application
API	application programming interface
ASP	active server pages
B2B	business to business
BO	business object
BOD	business object document
BSR	business service request
DC	data component
DTD	data type definition
EJBs	Enterprise JavaBeans
J2EE	Java 2 Enterprise Edition
JDBC	Java Database Connectivity
JMS	Java Message Services
JSP	Java Server Pages
MOM	message-oriented middleware
OAGI	Open Applications Group, Inc.
OAGIS	Open Applications Group Integration Specification
RMI	remote method invocation

SC	service component
SOAP	simple object access protocol
SQL	standard query language
UML	unified modeling language
XML	extensible markup language
W3C	WorldWide Web Consortium

References

Comella 00 Comella-Dorda, Santiago; Wallnau, Kurt; Seacord, Robert C.; & Robert, John. *A Survey of Legacy System Modernization Approaches* (CMU/SEI-2000-TN-003, ADA377453). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. Available <http://www.sei.cmu.edu/publications/documents/00.reports/00tn003.html>.

Gamma 95 Gamma, E.; Helm, R.; Johnson, R.; & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley Publishing Company, January 1995.

IBM International Business Machines. “Fundamental Information Aggregate Concepts” [online]. Available <http://www-106.ibm.com/developerworks/patterns/bi/concepts.html> (2001).

J2EE Sun Microsystems, Inc. Java 2 Platform, Enterprise Edition. <http://java.sun.com/j2ee>.

J2EE 01a Sun Microsystems, Inc. “Session Façade Design Pattern” [online]. Available <http://java.sun.com/j2ee/blueprints/design_patterns/session_entity_facade/index.html> (2001).

J2EE 01b Sun Microsystems, Inc. “Fast-Lane Reader Design Pattern” [online]. Available <http://java.sun.com/j2ee/blueprints/design_patterns/bimodal_data_access/index.html> (2001).

Larman 00 Larman, Craig. “Enterprise JavaBeans 201: The Aggregate Entity Pattern.” *Software Development Magazine*, April 2000. Available <http://www.sdmagazine.com/documents/s=745/sdm0004c/0004c.htm>.

OAG 99 Open Applications Group. “Plug and Play Business Software Integration: The Compelling Value of the Open Applications Group.” Atlanta, GA: Open Applications Group, Inc., 1999.

Ozsu 99 Ozsu, M. T. & Valduriez P. *Principles of Distributed Database Systems*. Upper Saddle River, NJ: Prentice Hall, January 1999.

Thomas 98 Thomas, A. “Enterprise JavaBeans Technology – Server Component Model for the Java Platform”[online]. Available <http://java.sun.com/products/ejb/white/white_paper.html> (1998).

Appendix Representation of the Data Architecture in Rational Rose

Use Case View

There should be a mapping between use cases and service components.

Logical View

BO Realization (Package)

- for each service component: class of stereotype <<service component>>
- for each data component: class of stereotype <<data component>>
- for each wrapper component: class of stereotype <<external system wrapper>>
- class diagram including
  - "uses" relationships between service component and wrapper components
  - <<BSR>> relationships between service component and the BSR interface
  - <<Uses>> relationships between service components and data components
- class diagram mapping data component to subcomponents and tables, and relationships between data components
- sequence diagrams for operations performed by service components, if needed

Application Components

- application component 1 (package)
  - Applets
  - Reports/Summaries
  - Global View (layered diagram showing the interaction between all components)
- application component 2 (package)

:

- application component N (package)

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	October 2001	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
An Enterprise Information System Data Architecture Guide	F19628-00-C-0003

6. AUTHOR(S)

Grace Alexandra Lewis, Santiago Comella-Dorda, Pat Place, Daniel Plakosh, Robert C. Seacord

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2001-TR-018

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	ESC-TR-2001-018

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Data architecture defines how data is stored, managed, and used in a system. It establishes common guidelines for data operations that make it impossible to predict, model, gauge, or control the flow of data in the system. This is even more important when system components are developed by or acquired from different contractors or vendors.

This report describes a sample data architecture in terms of a collection of generic architectural patterns that both define and constrain how data is managed in a system that uses the Java™ 2 Enterprise Edition (J2EE) platform and the Open Applications Group Integration Specification (OAGIS). Each of these data architectural patterns illustrates a common data operation and how it is implemented in a system.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Data architecture, J2EE, OAGIS, component-based design, enterprise information systems, architectural patterns	72

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL