Service Oriented Architecture

An Approach to SOA

Marcel Baumann

Version 1.0.3

Table of Contents

1 Introduction

Modern client server strategic applications manipulate complex domain models. One approach to manage the inherent complexity of such solutions is to define a service oriented architecture. This architecture clearly separate the domain model from the business logic.

This document describes mechanisms to more easily implement a SOA solution for classical N tiers applications. The server offers mechanisms to solve the following problems

- Locate components and subsystems
- Startup and shutdown of the application
- Persistence layer supporting optimistic locking and remote edition of graph of data objects
- Logging of actions
- Audit of business data modification

Project experience has taught us the power of patterns in a SOA based architecture. The highest return was achieved with

- Object oriented approach and systematic use of information hiding
- Visitor pattern as a swiss army knife to implement various algorithms on business models
- Locator approach to develop loosely coupled components
- Data transfer object pattern to exchange information between layers

The intended audience is architects and senior designers in charge of complex mission critical applications.

2 Development Patterns

Client server application have often a structure similar to the one depicted in the diagram below. The blue color identifies components both used in the server and the clients. Yellow systems are glue code used to delegate client requests to the server. Red packages contains persistent objects.

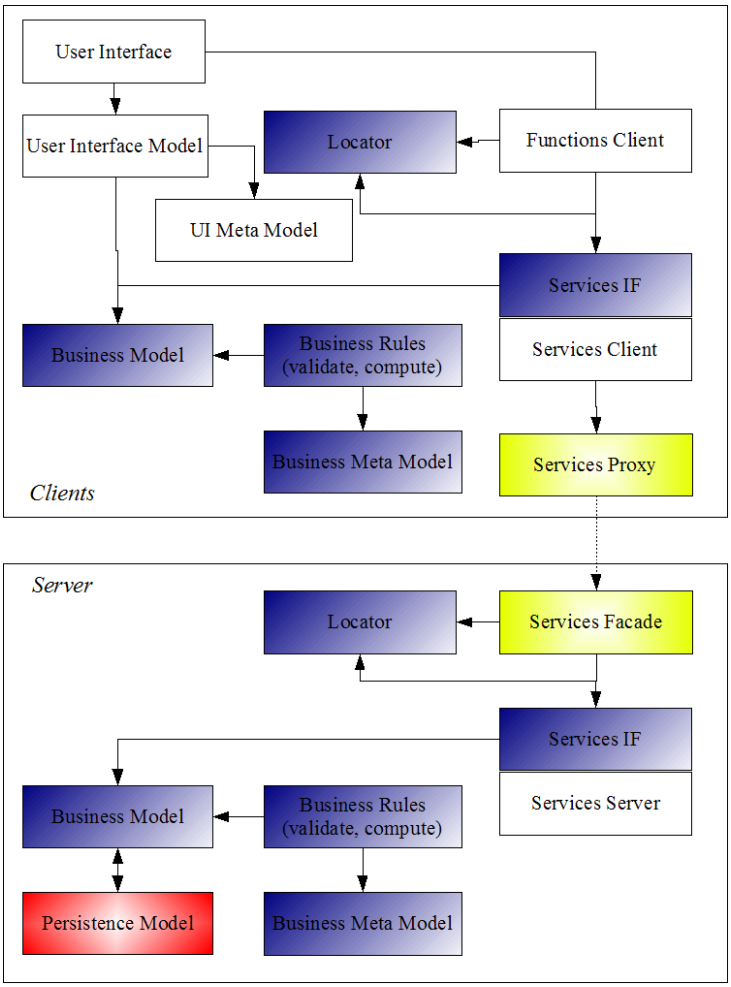


Illustration 1client-server structure

2.1 Business Logic

Business logic, in a very broad sense, is the set of procedures or methods used to manage a specific business function. Taking the object-oriented approach enables the developer to decompose a business function into a set of components or elements called *business objects*. Like other objects, business objects have identity, state and behavior. To manage a business problem you must provide the desired functionality. The set of business-specific rules that help identify the structure and behavior of the business objects, along with the pre- and post-conditions that must be met when an object exposes its behavior to other objects in the system is known as *business logic*.

2.1.1 Business Object

Business objects are domain entities describing the abstractions necessary to describe the business logic an application should implement. Commercial applications manipulate graphs of connected business objects. To simplify the definition of the model a taxonomy can be introduced.

- *Business Object*: An abstraction representing an entity in the domain model. A business object is modeled through classes.
- *Attribute:* An attribute is a value of a business object. An attribute has a type, a range of values and an optional initial value.
- *Relationship*: A business object can own other business objects. Such dependencies are specified as relationships. The only difference between an attribute and a relationship is that the type of attribute is never a business object type. But the type of a relationship is always a business object type.
- *Reference Code*: An enumeration of values defining a domain relevant type and its values. For example the set of currencies defined in the related ISO standard is a reference code.
- *Hierarchical Reference Code*: A reference code with a hierarchical structure. For example the department structure of a worldwide company can be represented through a hierarchical reference code.

2.1.2 Service Approach

Complex business logic often change when the market conditions change or the company expand in new market segment. It is highly recommended to separate application specific volatile business logic from the stable domain model. Such an approach is define in the service oriented architecture SOA.

The business object graph framework implies that some business object types are handled as graph roots. Roots always need well-known services to select a graph, update a graph, and delete one. The service providing them is specific to this root and its name is derived from the name of the business object type.

Roots are always the start point for validation rules covering sets of owned objects.

Services need to traverse graph of objects to accomplish their responsibilities. The traversal is provided through the iterator pattern. A depth first algorithm is used to limit resource consumption.

Experience: The business model must provide an efficient and versatile iterator implementation. It is a lot easier to provide clean services if they have access to powerful iterators. One nice extension is to provide iterator class using functor to define the action of the nodes of the graph.

2.1.3 Validation & Computation Framework

The complexity of business applications are hidden in the validation rules of the business model and the computation formula of derived data.

Validation

The validation rules should not be programmed inside the business model. They are defined as rules and executed through the services of a validation component.

Visual assistants should be provided to simplify the development of complex sets of rules. Often the final number of rules is in the hundreds or thousands.

Derived Attributes

Derived attributes are attributes, which value is defined through a formula. They are computed each time the value of an attribute involved in the formula changes.

A scalable approach is described in the article “Business Object Derived Attributes” [mb-attributes-2003].

Experience: Validation and derived attribute computation should be orthogonal to the business object model. Modification in the validation or computation should never require changes in the business model classes.

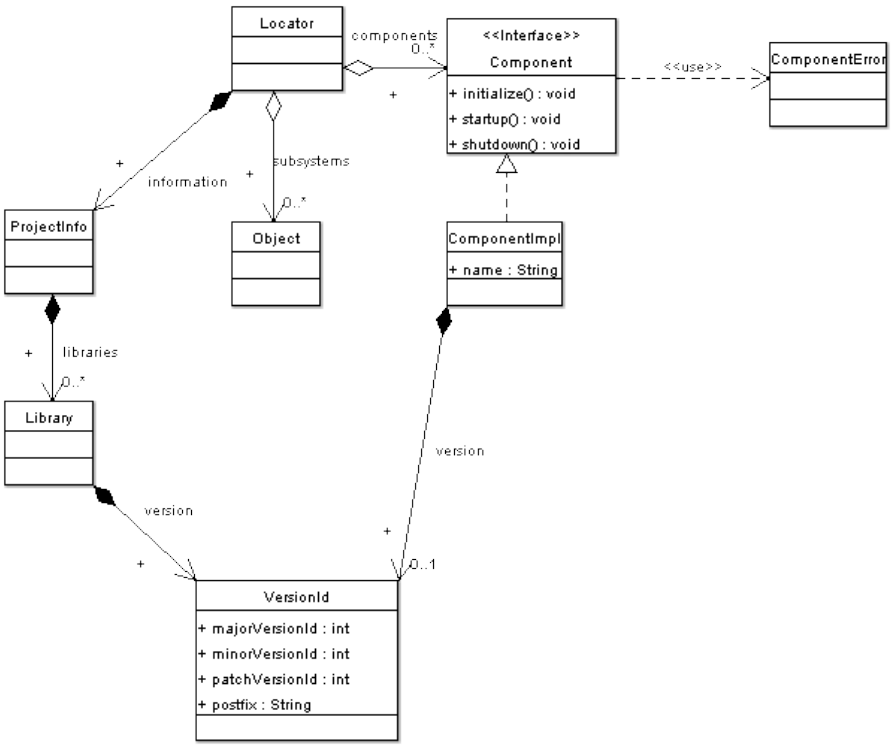
2.1.4 Default Initialization

Business objects and their attributes must be initialized upon creation. The initial value is often dependent from the user creating the objects and use cases being executed.

A scalable approach is described in the article “Business Object Derived Attributes”.

2.2 Service Locator

The service locator pattern is documented as one of the J2EE blueprints defined in the Sun white papers. The location framework streamlines how components are found and accessed in the applications.



The service registry is a directory service that contains available services. It is an entity that accepts and stores contracts from service providers and provides those contracts to interested service consumers. A contract is the interface of the addressed component.

2.3 Application Life Cycle

The application life cycle patterns are documented and implemented in the Apache foundation projects *Phoenix* and *Avalon*. It corresponds to the service factory concept in the service locator pattern as documented in Sun J2EE blueprints.

The life-cycle pattern separate the order of declaration of components from their initialization, startup, and shutdown.

The life-cycle details are as follow.

1. Creation of components and subsystems. It is guaranteed that basis services such as persistence, configuration, preferences and logging are available. The components should not access over components.
2. Initialization of components. It is guaranteed that all components and subsystems are registered. The components should not try to call services of over components.
3. Start of components. All services are available and can be called.

2.4 Access Rights

To be written.

2.5 Exception Propagation

2.5.1 Java Rules

Exceptions propagate information about exceptional conditions. Each application should define its exception languages to describe its exceptional and rather unexpected conditions. Enough information should be propagated to the interested client to allow him to react optimally.

The following guidelines are enforced.

1. Internal exceptions should be unchecked errors. The application developer is not interested to handle internal error condition or propagate them.
2. Framework exceptions informing the application developer what his request encountered a known request related problem should be checked exceptions.
3. Use existing exception classes. Before defining a new class of exceptions verify that the JDK is not providing a similar class.
4. If you wrap an exception propagate the context of the original exception.

These principles are applied to the interface of remote services.

- The Java architecture requires that each remote method can throw a remote exception. Therefore all remote methods support this checked exception.

- Application internal exceptions on server side are handled as checked or unchecked error. The server side code does not need to add special handling code for exceptions. The same code can be executed server or client side.
- The facade of the server wraps application errors in an server error or in a server exception as specified in the remote method invocation standard. The chain of exception must be broken to avoid the propagation of exception defined in packages only available in the server.
- The client has the freedom to propagate the original exception or unpack the application specific exception. The handling of the exception is either done in the wrapper communicating with the server or in the client application.

Exception cascading is only lost if the communication between the client and the server does not support native Java polymorphism. For example if CORBA is the used middleware the exception must transformed into a simple CORBA exception. The application programmer has the responsibility to define meaningful code and text. The client application has only error code and text to implement a legible exception handling for errors occurring on server side.

2.5.2 Client Server Rules

Client applications should not include libraries only used in the server such as JDBC drivers. The drawback is that the client cannot marshal the RMI representation of an exception containing a reference to a JDBC exception. To avoid low level RMI or CORBA exceptions the client should only receive exceptions which classes it knows. Therefore chained exceptions cannot be propagated from the client to the server.

The business facade is responsible to catch all exceptions in the server and transform them to a common set of exceptions known from the client. The architect is responsible to define this set and ensure it is used. Compilers cannot detect such errors. They are only detected as runtime client side errors.

2.5.3 Design

The architect should define an exception architecture orthogonal to the class structure of the application. The exceptions thrown in the service interface provides information why a service was not successful. The exceptions should never contain information related to the implementation of the service. As much as possible use exceptions defined in the Java standard packages.

Here a set of standard exceptions.

- *Access Control Exception*: Access to a resource is denied. The client does not have the privileges to access the resource.
- *Arithmetic Exception*: An error occurred when evaluating a mathematical expression.
- *Illegal Argument Exception:* The service call contains illegal parameters.
- *Missing Resource Exception*: A resource could not be found. The comment should document the missing resource with its name and type.
- *Unsupported Operation Exception*: The requested operation is not available.

Here a set of useful exceptions.

- *Persistence Exception*: An error occurred when accessing the database. The comment is the error message of the database exception.
- *External System Exception*: An error occurred when accessing an external system. The comment contains the name of the external system and a hint about the reason of the problem.
- *Programming Exception*: An error due to a programming error. This error should never occurred in a running system. During development assertions should caught such errors. After deployment the assertion errors should be caught and transformed before being returned to the clients.

Here a set of internal exceptions thrown in the framework.

- *Component Exception*: A component exception is thrown when the location and startup patterns encounter a problem. A description of the problem is available.

3 Mechanisms

3.1 Persistence

The persistence mechanism is implemented through the business object graph framework and standard persistence layer implementing industrial standards such as ODMG 3.0 or JDO. The business object graph framework provides mechanisms to manipulate graphs of domain objects in the server and in the clients.

3.2 Common Services

A set of blueprints document how a client server application should be designed.

- The domain model is represented as a set of connected business objects. The same model is used on the client and on the server.
- Data transfer objects are transmitted as graphs of domain objects. The business logic has only access to the domain model. They have never access to data transfer objects.
- The application is a composition of components. Components are located through the locator pattern.
- Application components are defined as interfaces. This interface is implemented in the server and client application. Because all components are accessed through the locator business logic does not know if it is executed on the server, on the client, or both.
- The application is started and stopped through the factory startup and shutdown pattern.

The service design follows the recommendations of Bjarne Stroustrup. Do not try to build a library before who have implemented particular solutions and understand the problem. After developing specific solutions for various applications a first set of services were abstracted and defined as frameworks available to the community.

- Business object graph framework with a common part for the business model, a server part for the persistence, and a client part for the visualization.

3.3 Client Services

The locator and client startup patterns provide the bricks to compose a client application from a set of components and to start it in a controlled manner.

3.4 Server Services

The locator and server startup patterns provide the bricks to compose a server application from a set of components and to start it in a controlled manner.

3.5 Business Logic

The business logic should follow a service oriented architecture approach.

4 Server Push

4.1 Purpose

Complex client/server applications often communicate with other servers or are part of a company wide workflow. For example the accounting system can modify administrative information of a contract in the contract and document management system.

Some modifications have impact on the clients currently working. The concerned client applications should be informed when such changes are committed. So the user will not try to edit obsolete data or perform operations no more legal in the new context. The server should push selected information to all interested clients. The amount of data and the number of occurrences should stay at a reasonable level. Otherwise the system will not scale when the number of concurrent clients increases.

4.2 Analysis

4.2.1 Business Model

The following entities were identified in the domain.

- *Event*: An event is an information of interest signaling a change in the application. An event has a source, an identifier, a type and a load.
  - *Event Send:* An event is sent to all known listeners. The action of sending an event is characterized as “fire and forget”.
  - *Event Reception*: All interested and available listeners receive a copy of the event.
  - *Event Dependency:* A causality chain can exist between events. The listeners are responsible to handle the chain adequately. The framework does not provide support for manipulating or traversing the causality chain.
  - *Event Cancellation:* An event cannot be canceled. Once sent no method exits to revoke it. The application is free to send a new event indicating that previous ones are obsolete.
  - *Reliability:* The approach of event dispatching is “fire and forget”. An event can be lost in the transmission layer or a listener could not be available to process the event. This situation is acceptable. An event can only be lost if the transmission middleware is not reliable.
- *Data load:* The data load contains application specific data. The application is responsible to ensure type safety.
- *Source:* The source of events sent through a channel.
- *Listener:* A component interested to events. A listener can set a filter on the event it wants to receive. The filter criteria are often based on the source and the type of the event. Application specific filter can be applied on the data load.
- *Filter*: A filter selects the events relevant for a listener or a group of listeners. Only the events fulfilling the conditions of the filter will be delivered to the listener.

- *Listener Context:* Event are asynchronous objects. No context or history is provided to the listeners.

The analysis of the application needs identified the following requirements.

1. A channel has at most one source. A source can sent events in multiple channels.
2. The middleware used to transmit the messages to the listeners could be changed later. The solution should provide mechanisms to support multiple middleware[1].

4.2.2 Non Functional Requirements

1. The number of listeners is not limited in the application. The maximum number of concurrent listeners must be at least 1000 listeners.
2. Events are seldom sent. The medium value is 10 events per hour. The maximum value is 1000 events per hour.
3. The medium size of an event is 200 bytes. The maxim size of an event is limited to 1 KB.

In general the filters are so configured that less than 1 % of the events are relevant for a specific listener.

4.3 Architecture

The overall architecture should be middleware independent. The implementation will decide if the server pushes to the clients or if the clients have to poll the server. The architecture decisions allow the use of standard middleware such as CORBA notification service, JMS or JMX notification service.

The design decisions follow the rules of event notification in the Java environment.

- The provider always pushes its events.
- The listeners receive their events through a push channel.

4.3.1 Register

The client registers itself on the local event manager. The registration specifies the source of the events, the channel used to retrieve them and the filters. The event manager implementation can propagate the information to the server. It must perform adequate operations to enable the reception of the expected events.

4.3.2 Event Propagation

The source register itself on the local event manager. The registration specifies the channels to use. The source is free to add filters on each channel.

4.3.3 Unregister

To be written.

---

1 The application could later be integrated in an office environment or a company wide workflow tool. The events could then be transmitted using JMS, email or RMI, or a propriatary method.

4.4 Detailed Design

4.4.1 JMS Reference Implementation

A reference implementation using JMS as middleware is provided. The listener filters are only applied to messages received on client side. To handle situation where too many messages are sent, a runtime option enables filtering of messages being sent. A message is sent if at least one listener is interested in.

The transmission mechanism for the JMS mechanism uses the publish/subscribe approach. The number of listeners is not limited by the implementation. The underlying used JMS middleware is the only constraint.

4.4.2 Java Mail Reference Implementation

A reference implementation using email – SMTP and POP/IMAP protocols - as middleware is provided.

To be written.

5 Selected Solutions

5.1 Data History

5.1.1 Concepts

Often financial applications are interested in the history of changes concerning business objects. Three strategies compete to implement change tracking.

- Each change is stored. This approach is typical for accounting applications. Each time money is checked in or withdrawn the associated transaction is stored in the database. This solution is the most elaborate one and tracks each change. But the domain processes must be adapted to support this approach.
- Each new version of a set of business object is stored. This approach is typical for document management applications. Each time a document reaches a consistent and relevant state, a copy is stored in the database with a new version number. This solution is optimal when the domain process seldom update complex documents or set of related business objects.
- A trace of a set of business objects is stored before being deleted. This approach is typical for configurations used in command and control applications or in file systems. The document is marked as deleted and either hidden from the normal user or copied in a scratch area. The solution is the one demanding the less resources. But the business processes must define the concept of a set of related business objects, which can be deleted and restored. Deleted objects can only belong to logical groups with no subgroups.

The most adequate variant for an application should be derived from the requirements and not be a technical decision. All three approaches can be realized with reasonable resources.

5.1.2 Realization

The first solution is implemented through the business logic definition. No additional feature needs to be realized. The effort is in the specification of transaction oriented business process.

The second solution requires an extension of the identity concept. The business identifier of the business object is a pair composed of an identifier unique to the type of business objects and a version number. These two values are unique for any object of a given type. Services should be provided to enable applications to either display the document currently under edition, the latest release of a document, or to view all versions of a document.

The application can decide to store the current instance and all variants with a version number in the same table or to store all variants into a shadow database to enhance responsiveness.

The third solution can be implemented using two approaches.

- Each business object carries a flag indicating if it is deleted or not. The search and retrieval queries must be adapt to remove all deleted objects before sending them to

the user.

- Each deleted object is copied to a shadow database. The concept of deletion insures that at most one instance of a given business object, identified through its identifier is stored in the shadow database. The copy operation of the deleted tree of business objects is already provided in the business object graph framework.
  The use cases to undelete a set of related business objects must also be defined.
  The delete concept is only meaningful if the undelete is defined.

Which variant is selected for an application is a requirement decision and not a design aspect. Often applications have no requirements for history of data.

5.2 Reference Code Loading

5.2.1 Concepts

Reference codes defines the vocabulary of the domain and are intensively used in the user interface layer. Client server applications have often network latencies. If the reference code values are each time requested from the server the reaction time would be too slow for the user. Therefore it is necessary to store the values of all often used reference codes on client side.

The approach to load the codes at application startup is often bad. The size of all code values is easily hundreds of kilobytes, slowing down the startup of the application. A local persistent representation of the reference codes should be available. This copy must be updated each time the values are updated on server side.

The advantages of this approach are the following.

- Faster startup time of the client applications
- Increased responsiveness of the user interface, in particular combo boxes containing reference codes.
- Automatic update of code values each time they are modified on server side. No new deployment of the client is necessary.

5.2.2 Realization

The set of locally stored reference codes are marked as preloaded in the domain model. Each reference code type is declared in a management structure with its current version number. This structure is a list of pairs containing the qualified name of the reference code type and its version number.

The server provides the following services.

- An implicit feature is that each reference code type which can be loaded and stored locally as a version number.
- Retrieves all preloaded codes available on server side. The result is a list of reference code values. Each type is stored in its own list. The result contains also the management structure. This call is useful for the initial retrieval of a client instance or if too many types have been updated.
- Retrieve all preloaded code values belonging to a given type. This call is useful to retrieve a single or few updated reference code types.

- Requests the list of updated reference types based on the list of reference codes stored locally and their version number. The call is useful to identify the changes between a client instance and the server application.

5.3 Reference Code Update

5.3.1 Concepts

The reference codes of any deployed application evolve over time. Some values are no more active – meaning they are displayed but cannot anymore be selected -, new ones are introduced; in the worst case the structure of hierarchical reference codes is modified.

Values stored locally on the client platform must be updated each time reference codes are changed. These new versions of codes must be sent to client installations without redeploying the client application because such operations are cumbersome and expensive.

Parallel to the new version codes, the persistent instances stored in the database must be updated to reflect the modified hierarchical reference codes.

5.3.2 Realization

The realization of efficient reference code updates as an implementation component and an administrative aspect.

The implementation of the reference code manager must provide the services described in the previous chapter about reference code loading. No additional services need to be realized. In other words the mechanisms used to store the values locally are the ones needed to update the stored reference codes upon a change of their values. Additionally a mechanism must provide the current version for each reference code type. A persistent data structure must be defined containing for each reference code type the following information.

- An identifier of the reference code type. Usually one good candidate is the qualified name of the class representing the reference code type.
- A version number of the code. We suggest to use the conventions defined in the Apache project. These conventions define the structure and compatibility rules. A version number is composed of a major number, a minor number and a patch number.

The administrative procedure defines the steps necessary to synchronize the system when modifying reference code values.

- The changes are documented in a change request form and are formally released. The update of the codes is planned and the users are informed about the downtime of the application servers.
- Stop the servers to avoid an inconsistent database when users are trying to modify reference code values with the old definition.
- Update the reference code values for each type. An existing value can be activated or inactivated. New values can be added. It is prohibited to delete any existing

value.

- The version number of each modified reference code type must be increased. This new version triggers the automatic update mechanism described in the reference code loading chapter.
- If the structure of hierarchical reference code types was modified the next steps are mandatory. Part of these steps should be performed during the first step of this procedure to minimize downtime.
    - Defines a mapping between the old hierarchy and the one one. The mapping is a business domain mapping and not a technical one. If the hierarchical code type is a control attribute, the translation rules can become quite complex and tedious to realize[2].
    - Writes the conversion jobs to transform the old code values to the ones for all involved instances in the database. Check with the business users if the updates are legal ones. If the updates cannot be performed, the changes must be canceled.
    - Updates all rows in the database. The timestamp of the instance is not changed in the database.
    - Update version number of the modified reference code types.
    - Verify the changes. The servers should be restarted in the quality checks are successful.
- Start the servers. All users should restart their users to retrieve the new reference code values. No mechanism exits to force t he users to restart their application upon a change. If such a change is needed the version number of the application must be increased. This solution implies a new deployment of the client application. The size of the update could be minimized if web start technologies are used for the distribution of the updates.

5.4 Inversion of Control

5.4.1 Concepts

The inversion of control pattern is the recommended approach to build a complex application. The pattern is also nicknamed as Hollywood principle

> “Don't call us we'll call you”

Inversion of control addresses a component's dependency resolution, configuration and lifecycle. The lifecycle is hard coded in the component interface.

5.4.2 Realization

The pattern is implemented in two classes.

- The component manager implements the lifecycle for the application and the locator pattern. The locator is used to find a component through its unique name.

2 This step is the most complex and the one consuming the most resources.

- The component interface defines the methods and lifecycle of any component plugged in the application.

One major strength of the implementation is the support of multiple level of initialization.

- *Construction*: Components are instantiated. In this step components have only access to basic services and do not have the right to access over components.
- *Initialization*: Components are initialized. In this step internal initialization is executed. Other components can be accessed but no guarantees are given that they are fully functional.
- *Startup*: Components are started. All components are fully functional and their services are accessible.
- *Running*: The application is running.
- *Shutdown*: The application is shutting down. Components should release all resources.
- *Finalize*: The finalize methods are executed and the virtual machine exited.

The implementation assumes that basic services are available before any component is started.

- Logging service is configured and running.
- Middleware naming services are running.

5.5 Business Identifiers

5.5.1 Concepts

Applications often need to generate unique identifiers used in the business domain to identify business objects. These identifiers have no relation with the technical identifiers used as primary keys in the database. A mechanism must allow client applications to generate unique streams of identifiers.

A possibility would be to use the database features to generate these identifiers. The drawback of this approach is that new objects must first be stored in the database before receiving an identifier. Often business rules require manipulation of the identifiers at the initialization of business objects. If we use the database features we must store two times business objects, first to make them persistent, second to store the modifications done in the business rules. This approach is therefore inefficient.

A better approach is to use the database to insure uniqueness of the Identifiers but delegate their creation to the application.

5.5.2 Realization

The identifier generator provides stream of unique identifiers. Each stream is identified through a unique name. This approach allows the application to generate identifiers for objects belonging to different types. The database insures that the identifiers are unique. The most efficient approach is to use low and high intervals and to generate individual keys in the application. Therefore an update is only performed

after all identifiers of an interval were generated.

The identifier generator provides the following services.

- generate key for named key stream.

To guarantee unique keys over multiple server instances the streams must exists in the database and are protected through optimistic locking.

5.6 External Systems

5.6.1 Concepts

Major enterprise wide applications connect to external systems and exchange information with systems such as accounting, purchase, management information systems.

External systems are hidden behind connectors are never directly visible to client applications. The services should be defined as a set of interfaces. All classes defining the interfaces should be defined in a small set of packages. The programmatic interface should be provided as a standalone library.

5.6.2 Realization

To be written.

5.7 Services Logging

5.7.1 Concepts

The detection of the source of errors in productive environment is quite difficult. Exhaustive logging often helps the maintenance to trace the reason why a given type of errors occurs.

Logging code is cumbersome to write and the majority of developers do not like to perform this task.

5.7.2 Realization

Application services are more and more generated using MDA approach and code generators. The generators add the needed logging statements in the services.

6 Distributed Computing

- *Caching:* When an application repeatedly distributes the same data, a significant gain in performance can be obtained by caching the data, thus changing some distributed requests to local ones.
- *Compression:* If the volume of data being transferred is large or causes multiple chunks to be transferred, then compressing the transferred data can improve performance by reducing transfer times.
- *Reducing messages*: Most distributed applications have their performance limited by the latency of the connections. Each distributed message incurs the connection latency overhead, and so the greater the number of messages, the greater the cumulative performance delay due to latency. Reducing the number of messages transferred by a distributed application can produce a large improvement in the application performance.
- *Asynchronous activities*: Distributed systems should make maximum use of asynchronous activities wherever possible. No part of the application should be blocked while waiting for other parts of the application to respond, unless the application logic absolutely requires such blocked activities.

There is some evidence what CORBA scales significantly better than RMI as applications grow in any dimension. RMI was designed as a relatively simple distributed application communication layer for Java whereas CORBA has a much more complex architecture, aimed specifically at supporting large enterprise systems.

Many client server projects over the years have shown that if your application can put up with the increased latency, asynchronous messaging maximizes the throughput of a system. Requiring synchronous processing over the Internet is a heavy overhead. Consequently, supporting asynchronous requests, especially for large, complicated services, is a good design option. You can do this using an underlying messaging protocol, such as JMS, or independently of the transport protocol using the design of the service.

7 MDA Approach

The model driven architecture approach allows the architecture to think at a higher level of abstraction. They can concentrate on the domain model and ignore the details of the target platform and used programming languages.

The code generation approach is very powerful when changes are frequent and must be mapped to multiple physical models such as database schema, object-relational mappings, business domain model, CORBA communication layer, validation rules, and user interface mappings.

MDA is used in our projects to supports multiple architectures through their platform specific models *PSM*. This approach to support CORBA and J2EE middleware for different versions of the same application.

8 Guidelines

8.1 Programming Rules

The following patterns are considered as very useful and must be used. No exceptions are tolerated.

- The visitor pattern is used to traverse graphs of objects. A functor variant of the visitor interface is provided to minimize coding effort for standard cases.
- The locator pattern is used to find components. Singleton approaches are prohibited.
- Object/relation mapping approach is used to persist business object in the database.
- Business rules and processes are realized against the business model instances and never directly against the database.

8.2 Application Rules

The application is implemented with the help of the business object graph and the explorer frameworks.

- The application has exactly one service locator. The service provides access to subsystems and basic services.
  - The business object handler provides the interface to the underlying layer for retrieving and storing business objects and reference codes. The handler is a singleton accessible through the locator.
  - The lock manager provides synchronization mechanisms on the business model for all threads in the application. The lock manager is a singleton accessible through the locator.
- The application has a mechanism to group the following components. Multiple groups can exist in the same application. A group is generally an instance of business object manager.
  - The business object types defining a business model.
  - The lightweight object handler responsible to transform business model objects into lightweight representations.
  - The business object handler responsible to retrieve and store business objects from and to the underlying layer.
  - The lock manager containing the set of existing graphs being locked build with the above types. Business object locking is the responsibility of the application and not of the database.

The business object manager can be used on client and server side. Access to services is always done through facades objects. There are three basic principles that apply when designing session facades.

- They don't do work themselves; they delegate to other objects to do the real work. This means that each method in a session facade should be small.

- They provide a simple interface. This means that the number of facade methods should be relatively small.
- They are the client interface to the underlying system. They should encapsulate the subsystem specific knowledge an not unnecessarily expose it.

Four kinds of objects that are generally find in most of EJB designs.

- *Value Object*: Serialized Java beans that contain data requested by a client. They contain a subset of the data contained the Entity beans and other data sources. They are the return types for session EJB methods. Value objects are also called data transfer objects.
- *Object Factory*: Factories are responsible for building value objects. They know about the different data sources, create instances of the value objects, fill in the instances of the value objects, and so on. Each factory can retrieve and update data to and from multiple data sources. There should be a factory for every root object in your object model. In a way, an object factory is acting as a facade onto the JDBC or Entity bean persistence subsystem, implementing the layering principle.
- *Entity EJB*: EJB should be standard data sources that can be globally useful across the enterprise. Entity beans should not contain application specific domain logic, nor should they be constrained to only work within a single application. Note that entity beans are optional and are not a required part of this architecture. A factory could just as simply obtain data directly form a data source like a JMS queue or a JDBC connection.
- *Action Object*: An action object represents a unique business process that a session bean may invoke. Action objects are required to handle business processes that are not related to simply creating, reading, updating, or deleting data. Like object factories, actions also act as inner-layer facades.

9 Tools

10 Process

10.1 Unit Tests

To be written.

10.2 Nightly Builds

Nightly builds are an efficient approach to regularly test a complex application. Each day the application is built, the database is newly created and loaded with test data and all unit tests are executed.

To inform the internal users about changes in the nightly build a summary of the changes is provided. The following informations are provided on a per application base.

- The list of implemented change requests with their number and a short description. The change request management tool should be used to automatically generate this list.

- The list of implemented functional requirements.
- Additional text can be provided to explain more complex available functionality.

11 Lessons Learned

- Initially we used the database identifier scheme to generate application specific unique identifiers. As long as the model identifiers have no rules this approach is very efficient. But often business rules applies on the identifiers defined in the business model. These rules are cumbersome to implement with the database generation approach. We recommend to use the database to generate unique identifiers through all clients. But the generation of keys is provided as a regular service. Therefore complex rules can be implemented as application specific functions realized in one pass. A key stream should exist for each type of business identifiers.
- Model identifiers manipulated by end users seem to be a difficult concept for many business analysts. An identifier is an identifier is an identifier. Therefore it is unique for all instances of the same type.
- Logging is done using the log4J package from the Apache foundation. Errors encountered during development and integration are all available in the logs for further analysis.

12 Remaining Problems

12.1 Documentation

12.1.1 Design Description

Documentation has always been a weak link in the software development process. It is often done as an afterthought. Most developers feel their main task is to produce code. Writing documentation during development costs time and slows down the process. It does not support the developer's main task. The availability of documentation supports the task of those that come later. So writing documentation feels like doing something for the sake of prosperity, not for your own sake. There is no incentive to write documentation other than your manager, who tells you that you must.

The developers are wrong, of course. Their task is to develop systems that can be changed and maintained afterwards. Despite the feelings of many developers, writing documentation is one of their essential tasks.

12.1.2 Traceability

To be written.

12.2 Roles

12.2.1 Architect

Becoming a good software architect is a difficult job requiring a large breadth of knowledge and many years of experience. It requires becoming an expert in software

dev3elopment. It also means learning many social skills. It requires becoming an expert in the domain of the project. The following are some characteristics of great software architects.

- Well versed in software analysis and design techniques, as well as architectural and design patterns
- Fluent in several programming languages
- Excellent verbal communication and writing skills
- Excellent at critical thinking and knowledge acquisition
- Ten or more years of experience in software development

With the exception of experience, the skills required to become an architect can be taught. People can learn new languages, read about design, and take classes to improve communication skills.

12.2.2 Integration Manager

To be written.

12.2.3 Toolsmith

To be written.

12.2.4 Coach

To be written.

12.3 Unit Tests

To be written.

13 References

To be written.

14

15 Cookbook

15.1 Create a Service

15.1.1 Rules

Software architecture has been emerging as a discipline over the last decade. A system's architecture describes its coarse-grained structures and its properties at a high level.

Services are a key component of a service oriented architecture *SOA*. Each system's software reflects the different principles and set of trade-offs used by the designers. Service oriented architecture has these characteristics.

- Services are discoverable and dynamically bound. See the locate pattern to learn how services are registered and queried.
- Services are self-contained and modular. Services references only the business model and other service interfaces. They never creates over service instances or static or singleton instances.
- Services stress interoperability
- Services are loosely coupled
- Services have coarse-grained interfaces
- Services are location transparent. The application programmers should not know if their code is executed on the client or on the server.
- Services can be composed

Services are entry points for business logic in service oriented architecture. All complex logic should be defined in service. The next steps create a new service.

15.1.2 Common

- Create a new interface for the new service. Defines the services as method in the interface. Document the interface using Java documentation[3]. Return value and parameter types must be declared in the common packages.
  The activities to design a maintainable and legible application are concentrated in the definition of elegant and lean service interfaces. The implementation of the service is mainly a component design. If the interface is stable a component can be integrated to new technologies without disrupting the clients.
- The interface must implement the component interface. All services are managed through the locate and application startup patterns.
- If some methods can be implemented independently from the server or client, implement them in a common class. To minimize the amount of code to write the class should inherit from the component default implementation class.

3 A check with documentation checker of Sun should detect no error or warning. Otherwise update the documentation.

15.1.3 Server

- Create a new class implementing the service interface. If a common class exists, you must inherit from it otherwise we suggest to inherit for the component default class.
- Implements the remaining methods using server side services.
- For each method implemented in the server you must provide a function the clients can call to implement their responsibilities. It is up to the architect to decide if a one to one mapping or a more compact one should be used[4].
- Update the startup sequence to register the component in the locator instance.

15.1.4 Client

- Create a new class implementing the service interface. If a common class exists, you must inherit from it otherwise we suggest to inherit for the component default class.
- Implements the remaining methods using the functions the server provides.
- Update the startup sequence to register the component in the locator.

15.1.5 Unit Tests

- Write unit tests calling each method of the component interface.

15.1.6 Hints

The following rules must be follow in the implementation.

- Access other components always through the locator services.
- Do not cache components locally.
- The semantic of the methods are specified in the interface of the component. If possible use pre- and post-conditions. The client and server implementation should respect this semantic.

15.2 Modify a Service

New functions can be added or old ones removed.

The next steps update a service.

15.2.1 Common

- Update the interface to reflect the new component signature. Update the documentation. Use the deprecated feature to mark methods which are obsolete.
- If a common implementation exists, update the implementation of the modified methods.

4 Avoid using collections in the server interface. Instead use arrays as documented in J2EE guidelines. Only arrays are guaranteed to be compatible with CORBA IDL interfaces.

15.2.2 Server

- Update the implementation of the modified methods.

15.2.3 Client

- Update the implementation of the modified methods.

15.2.4 Unit Tests

- Update the unit tests to call the new methods.

15.3 Make a Class persistent

Business models must be persisted in the database. The next steps make a class persistent.

- Declare the persistent class as persistent. The definition of its attributes and their database representation must be written in the MDA model. The class will inherits from the data object interface from the data object graph framework. Verify that all reference codes used in the class are declared as persistent.
- Generate the code artifacts with the code generator.
- Execute the DDL of the table containing the new class and the associated reference integrity rules. Now the database can store the class. If persistent data already exists, you must migrate it manually into the new table.
- Compile the source code artifacts. Now the application can manipulate the persistent class. The artifacts also contain the transformation operations for the transport layer.
- Copy the new OJB mapping declaration file in the class path of the application. Now OJB can automatically retrieve and save the persistent representation of the class.

Do not forget to write or extend a unit test to verify that the new persistent class is correctly handled in the server, the OJB layer and the database.

15.4 Interface External System

Complex business applications must communicate with external systems. The interface to such a system must be a clean and lean communication path. The next steps crate an interface to an external system.

- Component
  - Create a package containing all interface classes for the external system.
  - Define a manager as entry point to the subsystem. The manager must implement the component interface. All interfaces are managed through the locate and application startup patterns.
  - Implement the interface to and from the external system. If the external system needs to communicate with us a concurrent operation concept must be written

down[5].

  - Update the startup sequence to register the component in the locator.
- Interface
  - Document the interface to the system if duplex communication is necessary. The exchanged data must formally be documented. The scenarios must described with object scenarios.
  - The interface must be packaged as a separate full contained library file. This file will be distributed to the clients which need to communicate with our system.
- Unit Tests
  - Write unit tests verifying the described communication scenarios.

The major work when integrating an external system is not the implementation of the interface but its documentation. The technical users read this document to find out how they can integrate our application in their product and how company wide workflow is implemented.

5 The architects suggest that asynchronous communication using messages is used where possible. This approach scales when more load is applied on all involved applications.

16 Abbreviations

CORBA	*C*ommon *O*bject *R*equest *B*roker *A*rchitecture
DOG	*D*ata *O*bject *G*raph
DTO	*D*ata *T*ransfer *O*bject – the older terminology for this pattern was value object – The pattern is sometimes called Transfer Object EJB *E*nterprise *J*ava *B*eans
IDE	*I*ntegrated *D*evelopment *E*nvironment
J2EE	*J*ava 2 *E*nterprise *E*dition
J2SE	*J*ava 2 *S*tandard *E*dition
JDO	*J*ava *D*ata *O*bject
MDA	*M*odel *D*riven *A*rchitecture
OJB	*O*bject *J*ava *B*ridge
ODMG	*O*bject *D*ata *M*anagement *G*roup
OMG	*O*bject *M*anagement *G*roup
O/R	Object / *R*elational
POJO	*P*lain *O*ld *J*ava *O*bject