Broker Revisited

Michael Kircher, Klaus Jank, Christa Schwanninger, Michael Stal

{Michael.Kircher,Klaus.Jank,Christa.Schwanninger, Michael.Stal}@siemens.com

Markus Voelter

voelter@acm.org

Copyright © 2004, Kircher, Voelter, Jank, Schwanninger, Stal

After having written the Remoting Patterns book [VKZ04], we felt that it was necessary to take a look at fundamental pattern in that context: Broker of [POSA1]. This revised pattern description reflects the current state of discussion. Main changes are in the responsibilities as well as the participants area of the original Broker pattern:

- Finding remote objects has been separated out. Possible solutions therefore are documented as Lookup pattern in [POSA3].
- The original Broker pattern has a Broker participant, which responsibility is the transmission of requests from clients to server, as well as the transmission of responses and exceptions back to the client. This participant has been replaced by a client-side part called Requestor and a server-side part called Invoker.
- The original Broker pattern contains a client-side and server-side proxy participant, which encapsulate system-specific functionality such as marshaling of requests and responses and mediation between client and requestor, and invoker and servant, respectively. Broker Revisited replaces these participants by encapsulating their responsibilities within other participants.
  Proxies are only needed when remote invocations are expected to be invoked as if they were local invocations. As this is not in every known use the case, we made them optional. They are not part of the core responsibility of the Broker Revisited pattern.
- Since proxies when needed should be able to be created at runtime, they shouldn't contain any system-specific functionality. Another reason why client proxies have been made optional is that location transparency is not always necessary.
- Broker Revisited focuses on remote communication, host-local communication is only an exception/special-case.
- The Bridge participant of the original Broker pattern has been separated out, because interoperability issues such as type system transparency are handled by an appropriate network and marshaling protocol.

Broker Revisited

The Broker Revisited pattern connects clients with remote objects by mediating invocations from clients to remote objects while encapsulating the details of network communication.

Example Suppose, you are going to design an innovative framework for home automation where a central network allows to connect different actuators and sensors. The framework should be capable of integrating heterogeneous devices such as VCRs, TVs, notebooks, lighting systems, PDAs, refrigerators, coolers, or security sensors. It provides connectivity using fixed network cables as well as wireless communication lines. External access to the E-Home system is possible via secured Internet access. Central command and control services help to monitor and modify the system's behavior or malfunctioning. Users and administrators might deploy custom services that leverage multiple devices to allow emerging behavior such as triggering events on one device when other events on different devices occur. Examples could be a recording service where the recording of a television broadcast is triggered by a central clock or the activation of the cooling system by an external telephone call. Services will be implemented by the different vendors of the devices.

In order to make all this possible, the services have to collaborate. But the services are located on various controllers in the house, connected through a network between them and software running on those devices. It should be possible to change the collaborations easily and services should not be expected to always run on the same controller. To make the framework successful it is necessary to make it easy for vendors to provide new services quickly, without much overhead in learning how to develope software for the framework.

Context A system that consists of multiple distributed objects that need to interact with each other synchronously or asynchronously.

Problem Distributed software systems face many challenges that do not arise in single-process software. One major challenge is the communication across unreliable networks. Other challenges are the integration of heterogeneous components into coherent applications, as well as the efficient usage of networking resources. If developers of distributed systems must master all these challenges within their application code, they likely will loose their primary focus: developing application code that resolves their domain-specific responsibilities well.

Communication across networks is more complex than local communication, because remote communication concerns have to be considered, e.g. connections need to be established, invocation parameters have to be converted into a network-capable format and transmitted, and a new set of possible errors has to be coped with. This requires handling invocations to local objects differently than invocations of remote objects. Additionally, the tangling of remote communication concerns with the overall application structure complicates the application logic.

The following **forces** must be addressed:

- *Location Independence*—The location of the remote objects should not be hard-wired into client applications. It should be possible to run remote objects on different machines without adaptation of the client's program code.

- *Separation of Concerns*—Application logic and remote communication concerns should be well separated to manage complexity and allow for evolution of each concern independent of the other.
- *Resource Management*—Network and other communication resources have to be managed efficiently to minimize footprint and overhead of distributed systems.
- *Type System Transparency*—Differences between type systems should be coped with transparently.
- *Portability*—Platform dependencies should be encapsulated and separated from the application logic.

Solution Separate the communication functionality of a distributed system from the application functionality by isolating all communication related concerns. Introduce a client-side requestor and a server-side invoker to mediate invocations between client and remote object. The client-side requestor provides an interface to construct and forward invocations to the server-side invoker. The invoker dispatches incoming requests to the remote object and returns potential results of remote object invocations to the requestor.

A marshaler on each side of the communication path handles the transformation of requests and responses from programming-language native data types into a byte array that can be sent across the wire, and vice versa.

The details of communication and management of communication resources is hidden from the client and the remote object by the requestor and invoker. Further, the invoker provides a registration interfaces. This allows remote object implementations, so called servants, to be registered and made accessible to clients.

Structure The following participants form the structure of the Broker Revisited pattern:

A *client* interacts with a remote object.

A *requestor* forwards requests to an invoker across the network.

An *invoker* invokes requests on a servant. It adapts the interface of the servant, so that the other participants can stay independent of any remote object specifics. Depending on the implementation, this invoker-internal Adapter [GoF] can be decoupled and large parts of the invoker can serve multiple remote objects.

A *servant* implements a remote object the clients wants to use, whereas the remote object may represent an actual object, a component, or a service.

A *marshaler* transforms between invocation parameters/results and a serialized format.

The following CRC cards describe the responsibilities and collaborations of the participants

Class Client	***Collaborator***
Responsibility	• Requestor
• Invokes remote objects on the requestor • Collects all invocation information, such as the remote object reference and the invocation parameters, in a request object.	

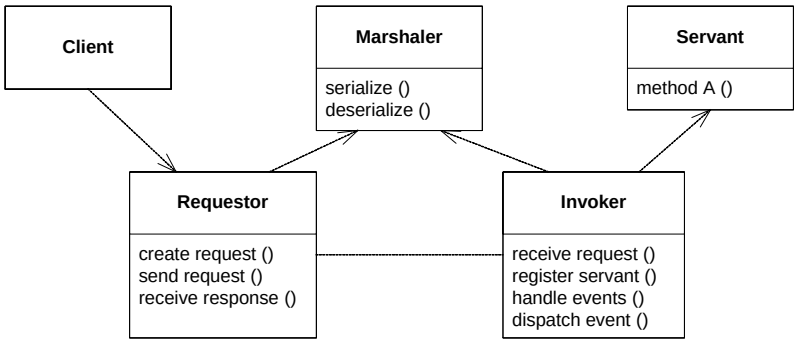
Class Requestor	***Collaborator***
Responsibility	• Invoker • Marshaler
• Provides an invocation interface, where clients can hand over the request object. • Marshals the invocation parameters on invocation, and de-marshal the returned results using the marshaler. • Uses communication resources, such as network connections, threads, and possibly manages those. • Sends invocations to remote objects, respectively the invoker instances they are registered with. • Hands over the request to the transport layer. • Waits for responses from remote objects and blocks clients until the result can be returned.	

Class Invoker	***Collaborator***
Responsibility	• Marshaler • Requestor • Servant
• Allows servants to be registered. • Receives requests from the transport layer. • Demarshals request parameters using the marshaler, and marshals results of the invocation in a response using the marshaler. • Invokes operations on servants using the request parameters. • Uses the communication resources and possibly manages those. • Sends responses back to the requestor.	

Class Servant	***Collaborator***
Responsibility	
• Implements the remote object's functionality.	

Class Marshaler	***Collaborator***
Responsibility	
• De-/Marshals invocation parameters and results.	

The dependencies between the participants are illustrated by the following class diagram.



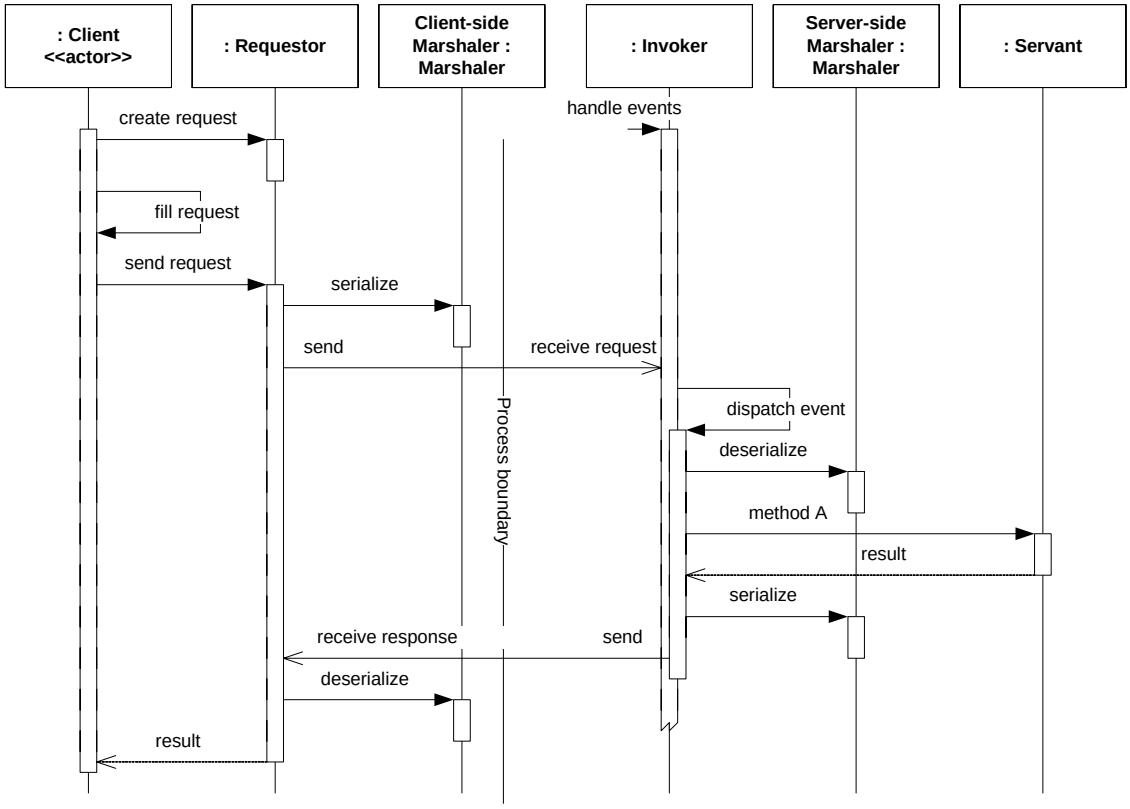
Dynamics For clients to interact with a remote object it is necessary that they first identify the remote object. This is done via an object reference. The object reference is different from a reference to a local object, because it must contain information about the remote host, where the remote object is located, as well as an indication of how it can be reached, for example the network protocol and endpoint information.

The client needs a reference to the remote object. One solution is that the client obtains an object reference to a remote object via Lookup [POSA3]. The client only needs to use the Lookup the first time it invokes this object, after that it could store the reference. To perform an invocation on the remote object, it constructs a request, handing over the object reference and all invocation parameters to the requestor.

The requestor marshals the invocation parameters using the marshaler, identifies the remote host the request has to be sent to, and sends the request to the remote host. For a discussion of whether connections are needed or not, refer to the Implementation section.

On the server-side the invoker, waiting for requests to arrive, receives the request. The invoker demarshals the invocation parameters, again using a marshaler, looks up the servant, and invokes the servant with the parameters. In order to invoke an operation on the server it has to adapt to the servants interface. When the invocation returns, the results are marshaled by the invoker and sent back to the requestor.

The just mentioned scenario, but without the lookup of the reference to the remote object, is illustrated in the following figure.



Implementation There are several steps involved in implementing the Broker Revisited pattern.

1 *Define an invocation interface.* The invocation interface of the requestor must allow clients to construct and send requests. An invocation request comprises several parameters including at least the reference of the servant, the method identifier and optional method parameters. A possible solution could be to provide a Factory method [GoF] for the construction of request objects:

```
Request r = requestor.createRequest (servantReference,
                                     "someOperation" );
r.addParameter( xyz );
r.addParameter( "12" );
requestor.send(r);
```

If it is desirable to completely hide the details of request creation, another possible solution could be to provide an invocation method with the request information as parameters.

```
requestor.invoke (servantReference,
                  "someOperation",
                  args[]);
```

Keep in mind that all request parameters must be capable to be serialized into a byte array that can be sent across the wire, and vice versa.

2 *Select marshaling protocol and implement the marshaler*. Define the mapping of invocations (request-response) and errors to the underlying communication protocol. Choose an appropriate marshaling protocol. For example, support for heterogeneity can be established by using a platform independent marshaling protocol such as XML.
Define the marshaling mechanism. A possible solution is to use reflection to introspect the

structure of the type that has to be serialized, at runtime. Many interpreted languages provide this generic, built-in facility.
If it is desirable to provide for the developer the flexibility to implement a custom serialization algorithm at type level, another possible solution is to provide a suitable interface that defines operations for serialization and de-serialization.

```
ISerializable {
    writeObject (byte[] stream);
    readObject (byte[] stream);
}
```

Both solution approaches can be combined in the way, that a type superclass implements the interface with a default mechanism based on reflection. Changes, for example to optimze the serialization process, can be done by the developer for each type individually.

In situations when complete objects need to be transferred, either of the two patterns, Value Object [Fowl03] or Data Transfer Object [Fowl03], are recommended. Value Object is useful for small objects, that you eventually want to compare. Data Transfer Object is useful to make communication more coarse grained, grouping related data as one object in a request or a response. In cases when complex data types have to be marshaled, it is often useful to delegate the marshaling to the object representing the complex data type.

To allow for easy customizing and extension, a design according to Pipes and Filter [POSA1] is advisable, where the filters are implemented as Interceptors [POSA2]. Using this design specific marshalers or additional processing steps, such as the encryption of invocation messages are easy to introduce. Below a possible solution of an interface for a custom marshaler is shown:

```
IMarshaler {
    byte[] serialize (Object o);
    Object deserialize (byte[] stream);
}
```

3 *Select communication protocol*. Depending on the quality of service that needs to be provided, either a connection-oriented or connection-less protocol is used. Additionally, it might be important to verify the Quality-of-Service (QoS) properties, supported by the communication protocol.

4 *Implement the network communication.* Use the Acceptor/Connector [POSA2] pattern for establishing connections between requestor and invoker. For dispatching incoming requests and asynchronously arriving responses use the Reactor [POSA2] pattern.

5 *Implement resource management*. The participants in the Broker Revisited pattern have to manage multiple kinds of resources, such as connections, threads, and memory. To ensure the fulfillment of non-functional properties, such as scalability and stability, it is important to manage those connections effectively and efficiently. Connections between requestors and invokers can be reused and shared using the Caching [POSA3] and Pooling [POSA3] pattern, respectively. For efficient self-management of threads that use shared resources, the Leader/Followers [POSA2] pattern provides effective means.

6 *Define an registration interface.* Define a registration interface on the invoker for the registration and unregistration of servants. Registered servants are referenced by the invoker which is then able to dispatch client requests to the corresponding servant instance. A possible solution could be that the developer creates the desired servant instance and passes it to the invoker:

```
IInvoker{
    registerServant (servantInstance instance);
    unregisterServant (servantInstance instance);
}
```

If it is desirable to completely hide the details of servant creation for example to enable a flexible lifecycle and resource management, another possible solution could be to pass the required information the invoker needs to create the servant instance itself. If Reflection

[POSA1] is available the servant implementation class identifier would be a sufficient imformation candidat.

```
IInvoker{
     registerServant(servantClass class);
     unregisterServant(servantClass class);
}
```

Possible resource management strategies to define the point of time for the servant instance creation are Lazy Acquisition [POSA3] and Eager Acquisition [POSA3]. Static Instance [VKZ04], Per-Call Instance [VKZ04] and Client-Dependent Instance [VKZ04] are strategies for lifecycle management.

7 *Provide a mechanism to reference servants*. In order to perform requests on remote objects, represented by servants, the clients have to obtain references to those remote objects. Therefore, let the invoker provide so-called object references that identify a remote object on its remote host, where it is located. Besides the ID of the remote object, they must also contain how the remote host can be reached, for example its network protocol and endpoint information. The object references are created by the invoker, but the actual distribution of those references to clients is outside the scope of this pattern. For more details about possible solutions, see the Lookup [POSA3] pattern.

```
Request r = (Request) marshaller.deserialize (requestStream);
String methodName = r.getOpName();
  // retuns an array of the types of the parameters
Class paramTypes = r.getParameterTypeArray();
  // retuns the parameters as an array
Object params = r.getParameterArray ();
  // tries to find a suitable method using reflection
Method m = servant.getMethod (methodName, paramTypes);
Response response = null;
if ( m != null ) {
       // invokes the operation on the servant
       // (error handling is omitted)
     Object result = m.invoke( servant, params );
     response = new Response (r.getOpName ());
     response.setResult (result);
} else {
     response = new ErrorResponse ("Operation not available");
}
byte[] responseStream = marshaller.serialize (response);
```

8 *Implement the mechanism to transform request messages into invocations*. Inside the invoker the operation identifier contained in request messages must be translated into an actual invocations. If Reflection [POSA1] is available, the invoker can adapt invocations to the servants' methods at run-time (so called dynamic dispatch). For programming languages that do not support reflection, adaptation code must be provided at compile time to adapt to the interface of the servant (so called static dispatch).

9 *Decide on supporting asynchrony (Optional)*. Asynchrony can be supported purely in the client (client-side asynchrony) or purely in the server (server-side asynchrony).

Client-side asynchrony is implemented by having the requestor return after having sent the request. Responses to an invocation are received by the requestor independent of the client waiting for it. Clients can receive the results by a Callback [VKZ04] or Polling [VKZ04] model.

Server-side asynchrony decouples returning results of an invocation from sending the response. Servants run in a different thread than the thread receiving requests. When a servant finishes a method execution, it will inform the invoker by a callback. The invoker in turn will return the response to the client, the requestor respectively.

10 *Optimize local invocations (Optional).* Client invocations of operations on a remote object result in an overhead due to the marshaling and interprocess communication. This is different from invocations of local objects. Therefore remote communication should only be used if the remote object really resides on a remote host. As it is transparent to the client on which host the remote object is running, a client would have to check for every object, whether it is remote or actually local. For local invocations, the client might call the object direcly and not use requestor and invoker. To avoid the explicit checks in the client's application logic, a client proxy can be used, as in the case of the Transparent Broker, see the next implementation step and the variants section.

Optimizations in the communication between clients and servers residing in the same process can also be handled in the requestor. The requestor does not have to marshal invocations but can short-circuit the invocation. Different levels of such short-circuiting are possible. To find the right short-circuit concurrency issues have to be considered. For example, if a client proxy would directly invoke methods on the servant, the servant code would be executed in the client's thread and not in a separate thread, as expected. The multithreading behavior can become undesired or even dangerous with respect to deadlocks and race conditions. Further, be aware that if the used client proxy invokes the remote object operation directly, the requestor, nor the invoker, can perform any security, transaction, or other checks.

11 *Determine if transparency is needed (Optional).* If remote objects should be represented as local objects, use the Proxy [POSA1] pattern. Client proxies will make it transparent to clients whether local or remote objects are invoked. Nevertheless, because communication to remote objects is unreliable, clients have to cope with errors specific to remote communication—this cannot be made transparent without losing the guarantee of invocation delivery. For more details refer to the Transparent Broker variant.

Example Resolved Back to the automation framework: To integrate the services, the Broker Revisited pattern is introduced. This allows the services to collaborate on higher level services. The connections between them are abstracted by the pattern, so that the services can address them logically and are not required to deal with controller or network details. The developers of services do not have to understand the communication concerns implemented in the framework. This is necessary to keep the implementation, reconfiguration and maintenance overhead low. The responsibilities of service logic and communication logic is properly separated. Services can even be collocated on the same controller in order to reduce hardware cost, while the overall collaboration scenario is not influenced.

Variants *Transparent Broker.* From an application-developers perspective it would be ideal, if the remote objects could be invoked as if they were local objects. The usage of client proxies hides the explicit creation of requests. A client proxy provides the same interface as the remote object, respectively the servant. It is transparent to the clients whether they access the client proxy or the actual remote object. As a special case, remote objects might actually be local to the client. In this case the implementation of the requestor might notice this and perform optimizations, such as avoiding overhead by not marshaling and sending the request across a network connection, but invoking the object directly or by a short-cut through the invoker.

Client-side Asynchrony. Client sends the request and receives the response via Callback or Polling.

Server-side Asynchrony. Decouples the execution of a method invocation from sending of a response. They may run in separate threads.

Consequences There are several **benefits** of using the Broker Revisited pattern:

- *Location Independence*—Clients do not have to care where an object is located, though for remote objects, they always have to use the more complex interface, unless a Transparent Broker is used.
- *Type System Transparency*—Differences in type systems are coped with by a intermediate network protocol. The marshaler translates between programming language specific types and the common network protocol.
- *Separation of Concerns*—The communication and marshaling concerns are properly encapsulated in the requestor, invoker, and marshaler.
- *Resource Management*—The management of network and other communication resources such as connections, transfer buffers and threads is encapsulated within the Broker Participants and therefore seperated from the application logic.
- *Portability*—Platform dependencies which typically arise from low level I/O and IP communication are encapsulated within the Broker Participants and therefore seperated from the application logic.

There are also some **liabilities** using the Broker Revisited pattern:

- *Error Handling*—Clients have to cope with the inherent unreliability and the associated errors of network communication.
- *Overhead*—Developers can easily forget about the location of objects, which can cause overhead if the expenses of remote communication are not considered.

Known Uses. **CORBA**. CORBA is the old man amongst the middleware technologies used in today's IT world. CORBA stands for Common Object Request Broker Architecture and is defined by its interfaces, their semantics and protocols used for communication. CORBA supports the basic Broker Revisited pattern, as well as the Transparent Broker. For the basic functionality CORBA supports the so called Dynamic Invocation Interface (DII) on the client-side. The invoker is separated between Object Request Broker (ORB) core, Portable Object Adapter (POA), and skeleton. The server-side skeleton is generated from IDL, as the client-side stub is. Various ORB extensions support a wide variety of advanced features. CORBA supports client-side asynchrony via standardized interface. Server-side asynchrony is only supported proprietarily.

RMI. Sun's Java Remote Method Invocation (RMI) is based on the Transparent Broker variant pattern. The client-side proxy (so called stub) and the server-side invoker (so called skeleton) have to be created manually by an additional compilation step. In contrast to CORBA the servant interfaces are not written in an abstract IDL, but in Java. Consequently RMI is limited to the usage of Java. To establish interoperability RMI-IIOP is provided. RMI doesn't support client-side or server-side asynchrony out of the box. Lifecycle management strategies are implemented in form of the activation concept of remote objects. A central naming service (so called RMI registry) allows clients to look up servant identifiers.

.NET Remoting. Microsoft's .NET Remoting platform implements the Transparent Broker variant pattern to handle remote communication. Since the .NET platform supports reflection to acquire type information, the client proxy is created automatically at runtime behind the scene, completely transparent for the application developer. There is no separate source code generation or compilation step required. The interface description for the client proxy can be provided by MSIL-Code or by a WSDL-Description of the interface itself. The client proxy is responsible of creating the invocation request, but is not in charge of any communication related aspects.

The remote communication functionality of .NET Remoting is encapsulated within a framework consisting of marshalers (so called Formatters in .NET Remoting) and

Transport Channels, which abstract from the underlying transport layer. This framework is designed in a very flexible manner, allowing any custom extensions to fulfil for example QoS requirements.

Furthermore .NET Remoting supports the client-side asynchrony broker variants. Lifecycle management strategies for servants are also included within the framework. .NET Remoting doesn't have a central naming or lookup system. Clients have to know the object reference of the servant in advance. However different strategies avoid the hardcoding of the servants destination inside the client application code.

Indigo. Microsoft's operating system code-named Longhorn comprises Indigo as an integrated middleware infrastructure. It combines .NET Remoting, ASP.NET Web Services, .NET Enterprise Services, and MSMQ under a common framework. In addition, it offers a unified programming model for building service-oriented connected systems. The Connector subsystem of Indigo is structured using the Broker Revisited pattern. Ports are introduced that represent communication endpoints under which services are available. Inbound and outbound communication with services is provided by channels which closely resemble end extend the notion of .NET Remoting channels. Besides other styles of communication Indigo provides a remoting paradigm that is built upon the core communication framework. Message encoders take the role of marshalers. Proxies are available to provide requestor and invoker functionality. In contrast to .NET Remoting Indigo comes with built-in activation and security support.

Additional parts of Indigo include messaging services, system services (transactions, federation), a service model, and different hosting environments for these services.

See Also *Lookup* [POSA3] describes how to register and find remote objects. It also gives guidance on how to bootstrap a distributed system, when no initial references are available.

Smart Proxies [HoWo03] allow to transparently integrate ‘smart’ services, such as transaction handling, authentication/authorization, or even Caching [POSA3].

References

[Fowl03] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, 2003

[GoF] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[HoWo03] G. Hophe and E. Woolf, Enterprise Integration Patterns, Pearson Education, 2003

[POSA1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, Pattern-Oriented Software Architecture—A System of Patterns, John Wiley and Sons, 1996

[POSA2] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, Pattern-Oriented Software Architecture—Patterns for Concurrent and Distributed Objects, John Wiley and Sons, 2000

[POSA3] M. Kircher and P. Jain, Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management, John Wiley and Sons, 2004

[VKZ04] M. Voelter, M. Kircher, and Uwe Zdun, Remoting Patterns— Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware, John Wiley and Sons, 2004