

SOA Reference Model – Part 2 – the Flexible Service Runtime¹

By John Cheesman, Georgios Ntinolazos

Abstract

This article builds on the SOA reference model foundation described in the March 2004 CBDI Journal. It examines further the SOA concepts with a particular focus on the patterns and software roles needed for flexible coupling in the service runtime. We compare these patterns with existing technology-focused approaches for EJB, .NET and Web Services, and provide a systems integration scenario to support the flexible coupling requirement.

Distributed systems requirements

SOA is the current state of evolution of best practice in the organization and architecture of enterprise systems. As the information systems industry moves forward we have progressed from monolithic systems, to client-server, to more generalized distributed n-tier systems using object request brokers (ORBs). Within this context we have also needed to provide solutions to infrastructure requirements such as transaction support, security and so on – enter the container model, supported by both the J2EE architecture and .NET, where business logic is kept separate from technology platform provision of infrastructure services. For example, the container takes care of managing the transaction and you take care of the business functionality.

Where SOA moves the agenda on is in shifting the emphasis from tightly-coupled, fine-grain distribution of application design elements to more loosely-coupled, coarser-grain distribution of software services supporting business processes. This increases software provision flexibility and creates a more direct link to business process change. Although many of the underlying themes and principles are common, the method and level of applying them is significantly different.

Loose-coupling has many connotations, so in this context we should perhaps use the term “flexible-coupling”. Specifically, when connecting service consumers and providers together we want to be able to provide strong, well-defined and stable connections, but also flexible, readily reconfigurable connections. This means that existing service consumers and providers can be assembled together *unchanged* to obtain business solutions in different technical environments and with different operational constraints.

Service Dependence, Technology Independence

What do we mean by flexible coupling? The heart of the requirement is that we enable functional software to be written so that its dependencies on external services can be made totally independent of the specific technical and operational environments in which those services operate. This independence enables service reuse and moves the software solution mindset on from a development focus to an integration and assembly focus. How do we achieve this?

The following elements and dimensions summarize the requirement:

¹ This article first appeared in the CBDI Journal, June 2004 and is copyright © CBDI Forum Limited.

- **Functional Interface:** the functional aspects of the service should be defined (in any language) independently of the implementation language and technology used. This covers both the operations or messages sent, and the data formats used.
- **Operational requirements:** operational (or non-functional) requirements of a service should be defined separately from the functional interface. These constitute the service-level agreement (SLA).
- **Communication protocol:** a provider's services should be accessible using a variety of communication protocols, and the choice of protocol should not affect the provider's functional behaviour (although it might affect the operational behaviour of the system)
- **Consumer insulation:** the service consumer is insulated from the technology choices made both within the service provider and in communicating with that provider
- **Infrastructure services:** the provision of infrastructure services is separated from the functional software. Further, infrastructure services should be "composable" meaning that they can be independently added and removed, and incrementally consumed (see [1]).
- **Life-time independence:** service consumers and providers should be unaware of each other's life-cycle and hence independent of any service "instantiation" concerns.

Before we look at an architecture pattern for achieving such flexible coupling let's first recap (from [2]) on the runtime concepts of service consumer and provider.

Components and Services revisited

In the March edition we introduced the development-time concept of a Functional Component (FC) and its runtime equivalent the Functional Component Object (FCO). In particular, we stated that a runtime software service was provided by an FCO through its runtime functional interfaces. This article follows this terminology, using service as the runtime concept and service definition and interface as design-time concepts.

Let's revisit these concepts and the flexible-coupling requirement through an example. Consider a hypothetical insurance company offering insurances services to customers. It has separate Car insurance and House insurance services and each of these makes use of Customer Management services. There is also an internal marketing function which makes uses some of those services for marketing purposes. The high-level functional component specification architecture is shown in Figure 1². This is a design-time model.

² The <<FCSpec>> stereotype classifies the classes as Functional Component Specifications

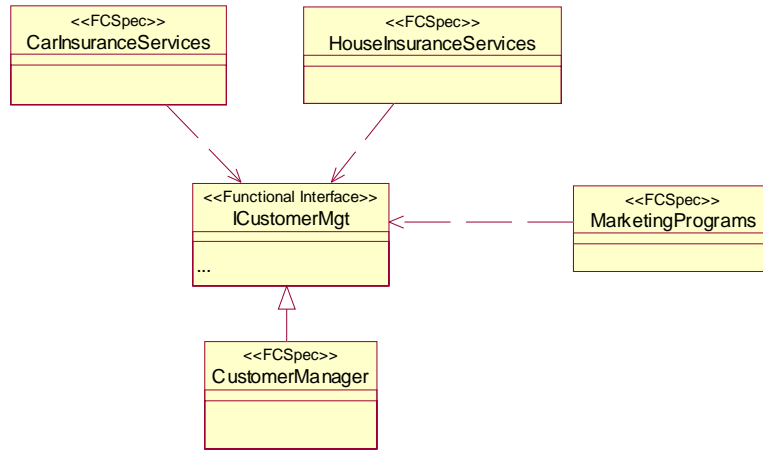


Figure 1 – Functional Component Specification Architecture

For the moment let's assume that the run-time architecture reflects the design-time architecture, that is, that we have one instance of each FC. So, for example, we will have a shared InsuranceCustomers instance of CustomerManager. However, later we'll run through some different run-time scenarios with two instances of CustomerManager (two FCOs) to demonstrate the benefits of flexible coupling.

The Pattern Elements

Based on the requirements listed previously, Figure 2 identifies the pattern elements within the solution.

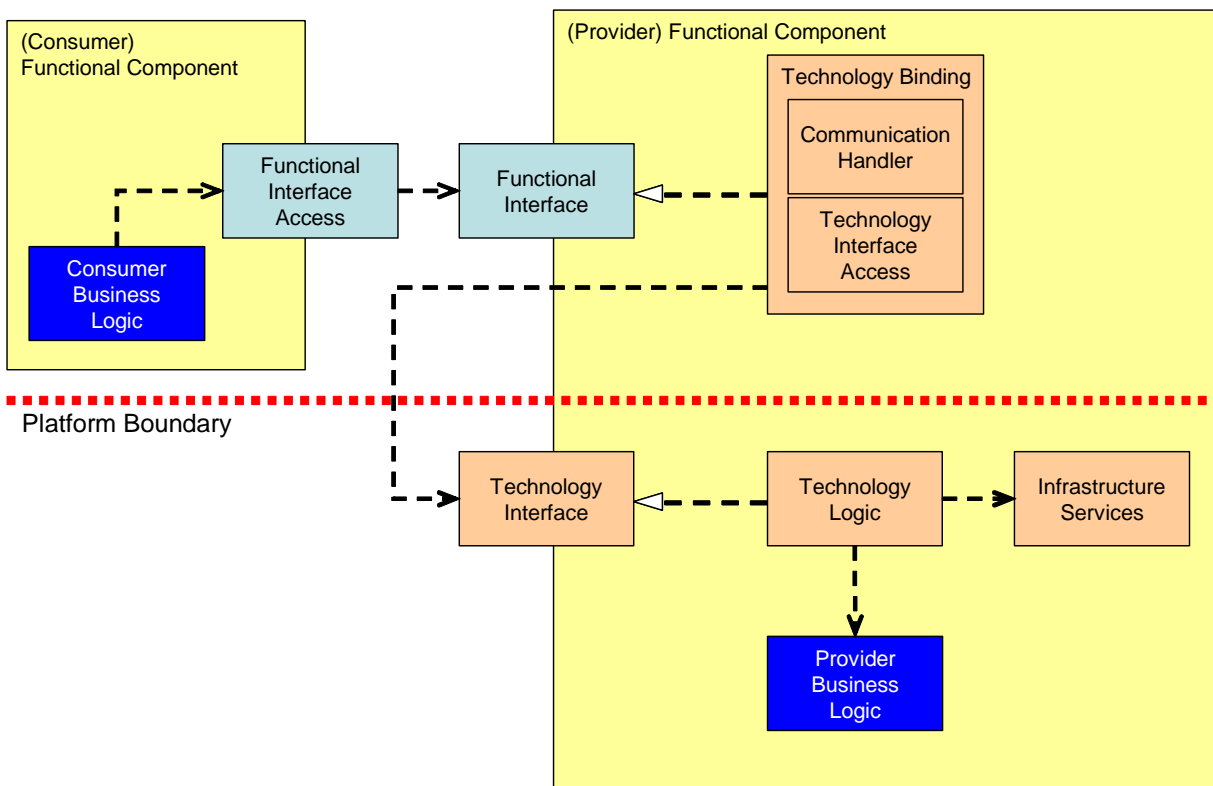


Figure 2 – Flexible Coupling Pattern Elements

Functional Interface

The Functional Interface, as described previously, represents the functional responsibilities of the provider in the overall service contract. The `ICustomerMgt` interface in Figure 1 is an example of such a functional interface. This interface may be defined in any appropriate specification language (e.g. UML, XML, or the various IDLs). The important point about the use of the term “interface” in this context is that it represents the functional interface, not a specific technology interface.

Functional Interface Access

This is the logical separation within the consumer FC of the code responsible for gaining access to the functional interface instance at runtime. In [2] we introduced the Service Discovery Service (SDS) and the concept of a Functional Assembly as mechanisms for registering the runtime FCOs and accessing their services. The SDS is similar in role to technology registry services such as UDDI or JNDI, but provides access to functional interfaces, not technology interfaces.

Technology Interface

A technology interface is a technology-specific mechanism for accessing a service of an FCO and depends on the FC implementation. While the functional interface is part of the functional specification of the FC, the technology interface is an implementation-specific refinement of the functional interface. At runtime an FCO may have many technology interface instances. For example, if the `CustomerManager` FC were implemented using EJB technology then it may have Home and Remote EJB technology interfaces corresponding to `ICustomerMgt`. It might also have a web service technology interface offering the same functional interface. In web service terminology the runtime technology interface instances are called service endpoints.

Technology Binding

There are three main aspects to the technology binding:

- Communication Handler is responsible for managing the communication protocols and data formats to be used when communicating between the implementation technology of the consumer and the implementation technology of the provider.
- Technology Interface Access is responsible for establishing access to a technology interface instance at runtime. This is similar in function to the Functional Interface Access role, but operates at the technology level. Whereas Functional Interface Access would use the SDS and the notion of a Functional Assembly, Technology Interface Access would use technology-specific equivalents of these, such as JNDI and the EJB environment context. Technology interface(s) will depend on the implementation technology of the provider. This is why this access role is logically part of the provider FC – it understands the implementation approach used. If an FC provides many different technology interfaces then it will also provide the corresponding technology bindings. If the Technology Interface Access is part of the Consumer (as detailed in many of the technology-level design patterns such as Façade) then there is a tight dependency between the Consumer and Provider’s implementation technology. This is what we are trying to avoid.
- The Technology Interface Access element is also responsible for any mapping needed between the functional interface organization of services and the technology interface organization of services and any mapping in their semantics. Technology design patterns can mean that the operation correspondence is trivial for newly developed FCs, but for legacy wrapping situations this may be more complicated – the existing

organization of services into technical interfaces may not reflect the most appropriate functional interface groupings. Additionally, the exception types need to be described in a technology-independent way and it is the job of the technology interface access element to perform this translation (e.g. translating a `java.rmi.RemoteException` into a functional `InfrastructureException`).

Technology Logic

Technology Logic represents the technology-specific code which realizes the Technology Interface and manages any required communication protocol translation. In container-based technology platforms much of this logic would typically be generated or provided by the implementation framework. This is also the logical location where calls to infrastructure services are made and kept separate from business logic. In EJB technology the technology logic is the EJB itself, generated by the J2EE platform, combined with the Bean class written by the developer. The EJB provides the hooks for the required infrastructure services and delegates to the Bean Class, which in turn provides access to the provider business logic. The same is true in .NET – the .NET Framework provides built-in or generated classes and interfaces which then delegate to the provider business logic.

Infrastructure Services

Infrastructure services would normally be provided by the technology platform such as .NET or the J2EE application server being used. Individual services, such as transaction management, might alternatively be provided by a specific transaction processing (TP) system such as CICS. In the web services arena standards are being developed for the specification of such infrastructure services too (see [1], [5] for example).

Responsibilities and Ownership

Perhaps one of the most important things to note about Figure 2 is that although the technology binding aspects of the solution are deployed (at runtime) with the Consumer, they are logically part of the *Provider* – the logical Functional Component boundary extends across the technology platform boundary. The onus is on the service provider to publish a variety of technology bindings. The appropriate one can then be selected for a given logical runtime architecture applied to a specific technical architecture. This selection can either be done manually at assembly time (for static assemblies) or potentially dynamically at runtime if sufficient information about the technical architecture and the operational requirements are codified and available.

Operational Requirements

In this article we are discussing a pattern for flexible coupling with respect to the functional aspects of the service contract – what about the operational aspects of the contract, the SLA? Some elements of operational requirements are met by the provision of infrastructure services such as security and transaction support. The relevant context information can be passed through the functional service calls and dealt with by the Technology Logic which delegates to the appropriate infrastructure service. This manifests itself either as a specific context parameter on operations at the programming language level, or as additional message header tags if using an XML-based definition language such as WSDL (Web Service Definition Language) (see [4], [1]).

Some operational requirements such as performance are essentially handled by the FC implementation and its mapping to the Technical Architecture. Other operational requirements,

such as security levels, need to be addressed in both places, with both specific infrastructure services and with the appropriate technical architecture support (e.g. the use of https).

An EJB Implementation Example

Let's examine how the pattern operates by looking at an EJB implementation of the CustomerManager FC providing its services through EJB interfaces. We'll then look at surfacing the same interface as a web service.

The functional interface is ICustomerMgt. This can be defined in any appropriate specification language. Here we've used UML to make it graphical. ICustomerMgt is shown in Figure 3.

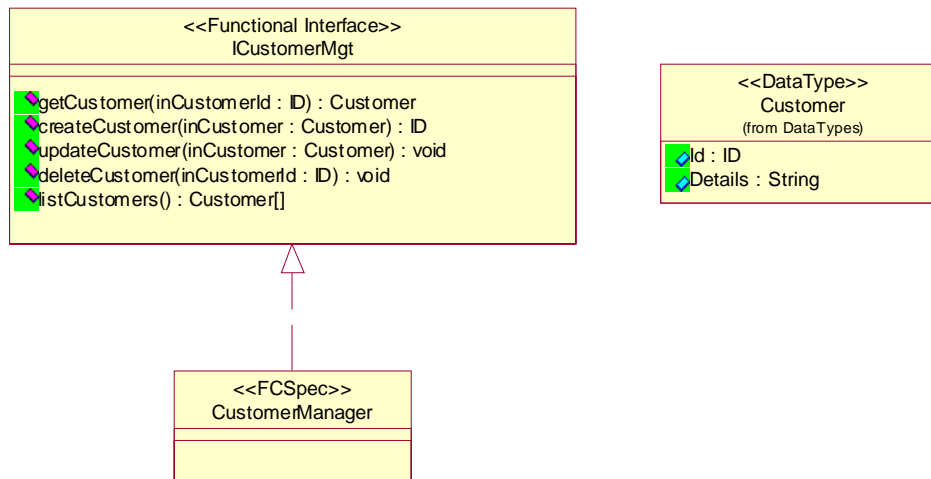


Figure 3 – ICustomerMgt Functional Interface

This is a very simple interface offering basic CRUDL (Create, Read, Update, Delete, List) functionality on customers. Customer data is typed by the Customer structured data type and ID is a simple scalar data type for typing identifiers.

UML is not a programming language though, so to represent this interface in code we need to map it to a specific programming language. This is the level at which the consumer and the provider will interact with the functional interface. For the EJB example we will choose Java to keep things simple. Note however, that whilst this is a programming language mapping, it is not a technology platform mapping – using Java as the functional interface language does not imply or require the use of EJB technology in the implementation.

We can map the UML stereotype <<Functional Interface>> to a Java interface, and the structured data type <<DataType>> to a Java interface representing such as type. We call this a TransferObject interface and its implementing class a transfer object (or TO) class, to use consistent terminology with existing Java architecture patterns.

To handle operational requirements a generalized context parameter is also needed and this has been included for completeness. As described above, this is used by the technology logic to hook in the required infrastructure services.

This gives us the following functional interface represented in Java (here the stereotype <<interface>> means Java interface):

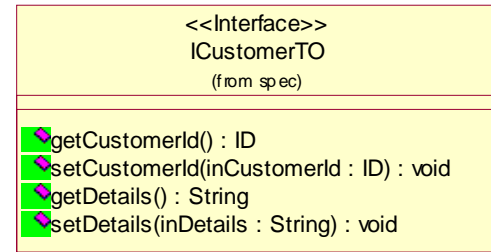
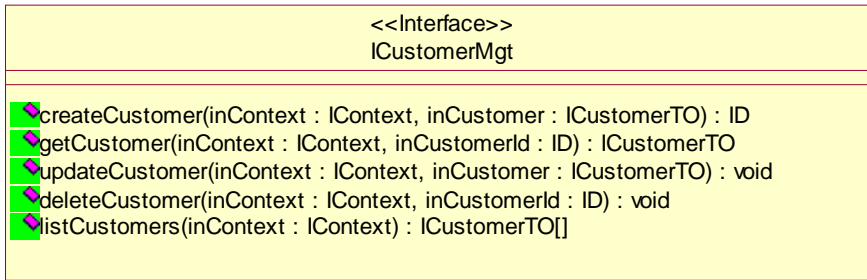


Figure 4 – ICustomerMgt Functional Interface in Java

The source code form is useful for showing the Exceptions – an important part of the contract:

```

package com.SOA.demo;
import ...;
public interface ICustomerMgt {
    public ID createCustomer(IContext inContext, ICustomerTO inCustomer)
        throws InvalidArgumentsException, NoPreconditionMetException, AlreadyExistsException, InfrastructureException;
    public ICustomerTO getCustomer(IContext inContext, ID inCustomerId)
        throws InvalidArgumentsException, NoPreconditionMetException, NotFoundException, InfrastructureException;
    public void updateCustomer(IContext inContext, ICustomerTO inCustomer)
        throws InvalidArgumentsException, NoPreconditionMetException, NotFoundException, InfrastructureException;
    public void deleteCustomer(IContext inContext, ID inCustomerId)
        throws InvalidArgumentsException, NoPreconditionMetException, NotFoundException, InfrastructureException;
    public ICustomerTO[] listCustomers(IContext inContext)
        throws InvalidArgumentsException, NoPreconditionMetException, InfrastructureException;
}

```

The Exceptions defined here are purely functional and do not imply any particular technology platform or communication protocol. If we now compare this with the technology interfaces which are used within the FC implementation we can see the differences (see Figure 5).

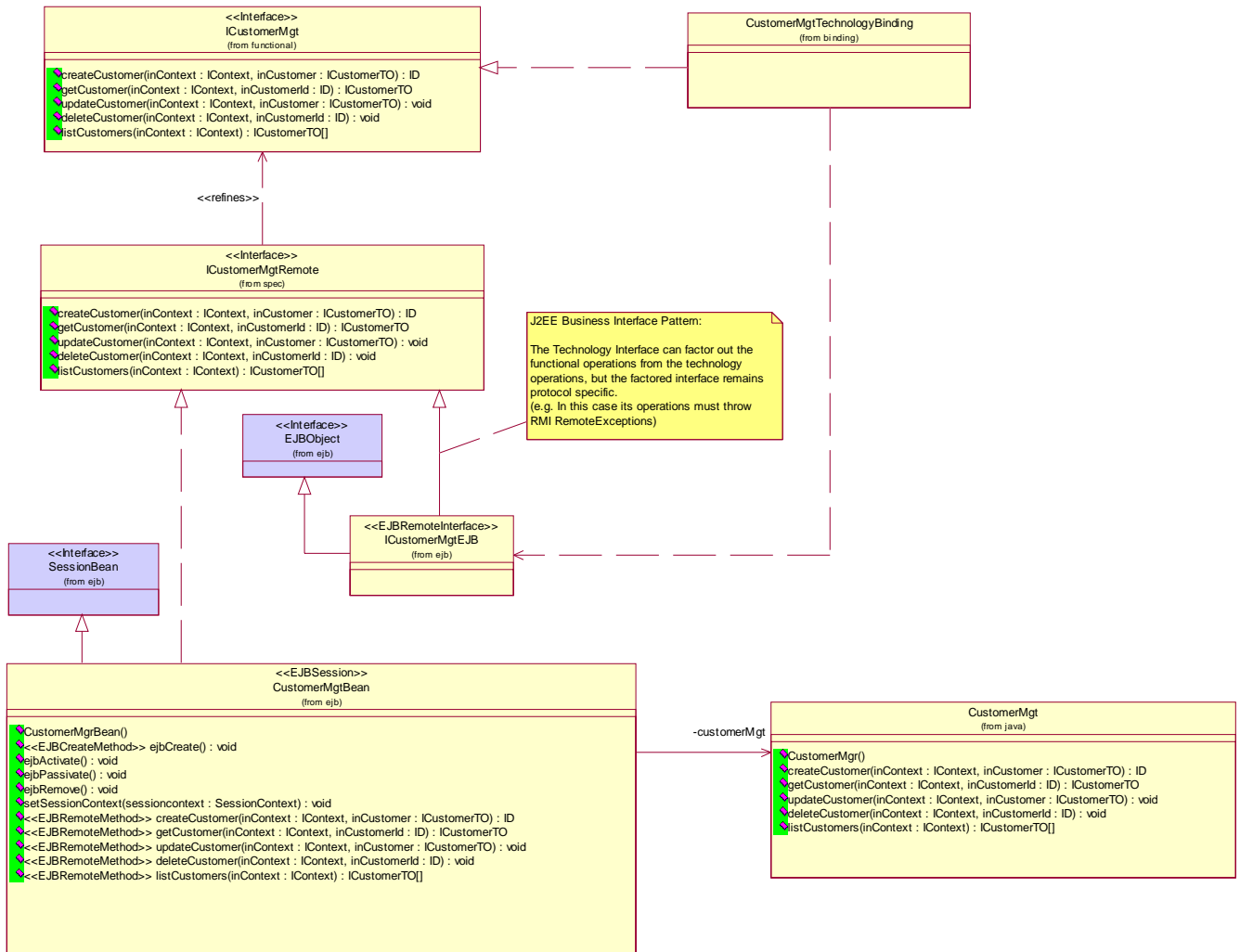


Figure 5 – Implementing Functional Interfaces in EJB

The pattern for the implementation of the CustomerManager FC in EJB technology in Figure 5 follows the flexible coupling pattern element summary in Figure 2 for the Provider FC.

The Functional Interface defines the functional behaviour of the FC, independent of technology. This is what the Consumer FC sees – hence the Consumer is insulated from anything to do with the EJB technology choice, RMI calls and so on. Within the CustomerManager FC implementation the functional operations are surfaced by an EJBRemoteInterface (called ICustomerMgtEJB). The use of this technology choice implies communication via Java RMI (Remote Method Invocation) and all the operations of this interface must throw an RMI RemoteException.

The J2EE platform automatically provides an implementation of the EJB itself (not shown) and that EJB then delegates to the Bean class written by the FC Developer (CustomerMgtBean). The EJB and the CustomerMgtBean collectively represent the TechnologyLogic shown in Figure 2. The EJB provides the container hook for the provision of Infrastructure services, and the Bean Class delegates to the business logic, represented here by the regular CustomerMgt class.

The technology binding is represented by the CustomerMgtTechnologyBinding class. This class actually offers (realizes) the functional interface and is physically located with the consumer. It is responsible for communicating with the appropriate technology interface of the FC and also for

translating any technology-level exceptions (such as RMI Remote Exception) into functional exceptions (such as InfrastructureException³).

As an aside, the example also shows the use of the standard J2EE Business Interface pattern whereby the “functional” operations of the technology interface are factored out into a separate interface (also called ICustomerMgtRemote). This is useful since it provides an interface for the bean class to implement which then gives compile-time checking. However, it is not technology independent because the operations of the interface still have to throw an RMI Remote Exception. The <<refines>> dependency shows that the functional ICustomerMgt interface and the technology-dependent ICustomerMgtRemote interface are strongly related, but one is purely functional and one is not. Clearly automation support for the process of implementing a functional interface can generate many of these related implementation elements.

Functional Components and Technology Components

It is worth noting also the clear distinction between the Functional Component (FC) and the Technology Component (TC) here. The TC is the EJB. It represents a specific runtime mechanism for accessing some of the services of the FC. In this example it corresponds to a functional interface. If we were to add another functional interface to the FC then we would potentially have another EJB to handle that.

The Web Services approach

If we look at the structure of a Web Service Definition Language (WSDL) file we see the same separation of functional interface and technology binding. A WSDL file also defines the runtime web service itself which means that it covers both design-time service definition information and service (instance) information (see Figure 6).

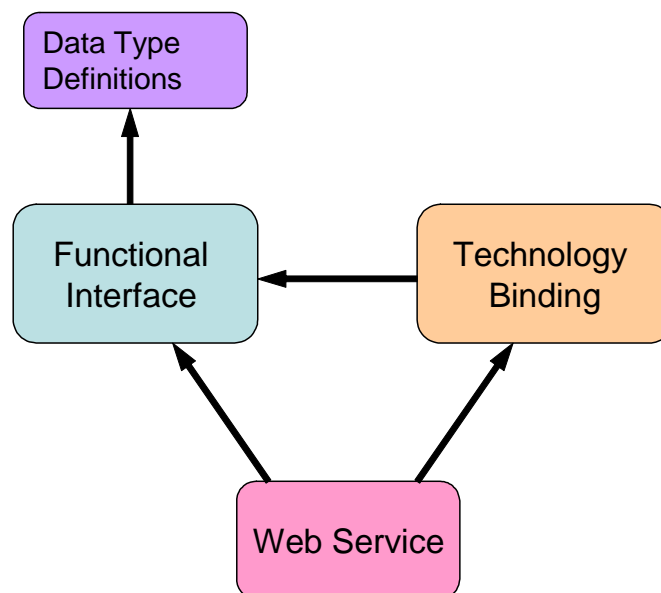


Figure 6 – WSDL Structure

Web services concepts map well to Figure 2. However, if we look at how web services are actually implemented and applied, we see that the label of “technology-independence” is open

³ The standardization of this set of exceptions is part of the Functional Component Model, introduced as part of the SOA Reference Model in the March edition [2].

to interpretation. In particular, although the abstracted functional interface is present, it is used to generate technology-specific interfaces and stubs, and the consumer interacts with those directly (see Figure 7).

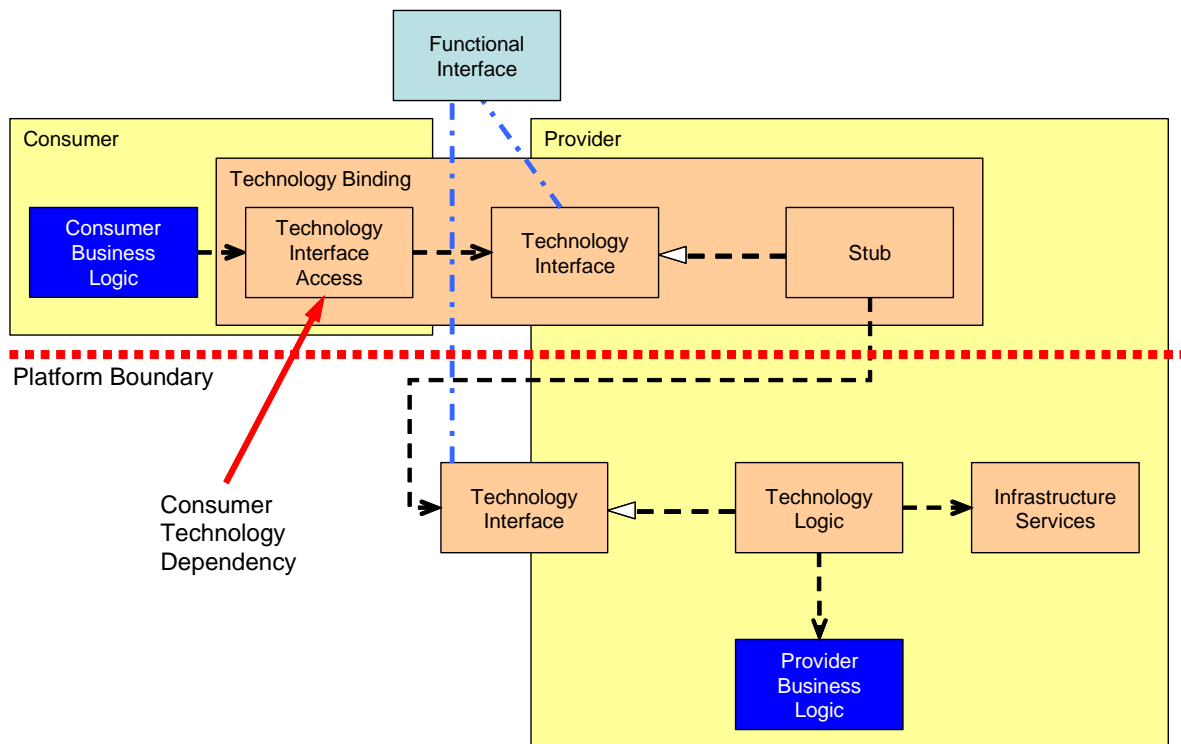


Figure 7 – the Web Services approach

So, although the web services approach gives flexibility in technology binding *choices*, the technology-dependency is still there once the choice has been made.

If we look at a code snippet from the java Technology Interface access code for accessing CustomerManager services as a web service we can see the technology dependency very clearly:

```

...
import javax.xml.rpc.Stub;
...
Stub stub = (Stub)(new CustomerService_Impl().getICustomerMgtPort());
ICustomerMgt customerService = (ICustomerMgt) stub;
customerService.createCustomer(...);

```

There are two technology-dependencies here: 1) the stub is specific to JAX-RPC (java.xml.rpc) and 2) the ICustomerMgt interface used is not purely functional – it has some technology dependencies (e.g. it has to inherit from java.rmi.Remote and its operations must raise RemoteExceptions).

The pattern we are presenting (in Figure 2) moves all responsibility for the technology interface and its access away from the consumer – the generated web service client code is part of the Provider FC, and so in the code for accessing that web service client.

Comparison with other approaches

The pattern we have presented here builds on some of the principles and approaches identified elsewhere. Here we briefly compare the approach with the distributed systems approaches of J2EE, .NET, Web Services and CORBA. The .NET comparison examines the published Enterprise Solution Pattern (ESP) for service contracts [3].

| | Functional Interface Access | Functional Interface | Technology Binding | Technology Interface | Technology Logic and Infrastructure Service |
|-------------|-----------------------------|----------------------|------------------------------------|----------------------|---|
| J2EE | | | RMI, JNDI, EJB Home/Remote (proxy) | EJB Home EJB Remote | EJB EJB Container + EJB Server |
| .NET ESP | | | Service Gateway, Service Interface | Service Interface | .NET Framework .NET Platform |
| WebServices | | WSDL | UDDI, WebService stub | Webservice | Pluggable (see [1]) |
| CORBA | | IDL | Stub, IIOP | Tie/Skeleton | Procedural access (no Container) |

There are a number of points to note:

- Both .NET and J2EE technologies provide technology-specific interfaces but no functional interface equivalents. This means that Consumers include Technology Interface Access elements rather than Functional Interface Access elements. The .NET Service Gateway and Service Interface patterns simply generalize technology-interface-level access and share responsibilities between Consumer and Provider.
- Both Web Service Definition Language(WSDL) and CORBA Interface Definition Language (IDL) provide a form of functional interface definition, but it is used as a mechanism for generating technology interfaces rather than the runtime functional interface itself. As with the Microsoft Gateway pattern, the technology binding is seen as a shared responsibility between the Consumer and Provider (rather than the Provider's responsibility with binding selection by the Assembler).
- There is no equivalent of the functional Service Discovery Service (SDS), rather there are only technology-level SDS's such as UDDI and JNDI. (UDDI is deemed technology-level since it serves up technology interfaces).
- Flexible-coupling across technology platforms will require standardization of the different context information needed for passing between infrastructure service implementations. These standards are now emerging. Although CORBA also specified separate

infrastructure services it did not provide a container-based or declarative model for accessing them. CORBA simply addressed the interoperability problem.

Back to the requirements

Let's return to the hypothetical insurance company. With the introduction of flexible coupling we can plot a path through some of the standard legacy wrapping and business integration issues. For example, the current insurance company is actually a merger between a car insurance company, with a car insurance system and its own customer management system, and a house insurance company with its corresponding systems. With the merger the company wants to integrate its customer information and be able to identify cross-selling opportunities and new combined insurance products.

An important architectural first step in this integration would be to introduce a standardized functional interface for customer management and to modify the existing consumers and providers of those services to use that interface. There would still be two distinct deployed customer management services, but only one *functional specification* of those services. Technology Bindings would need to be created for each legacy system to link to the original technology interfaces. This step would insulate the two consuming systems (Car Insurance Services and House Insurance Services) from the technology platforms being used in the Customer Management service. They would simply depend on the appropriate FCO. A Marketing programs system could then be introduced which used both FCOs and identified immediate business opportunities (see Figure 8).

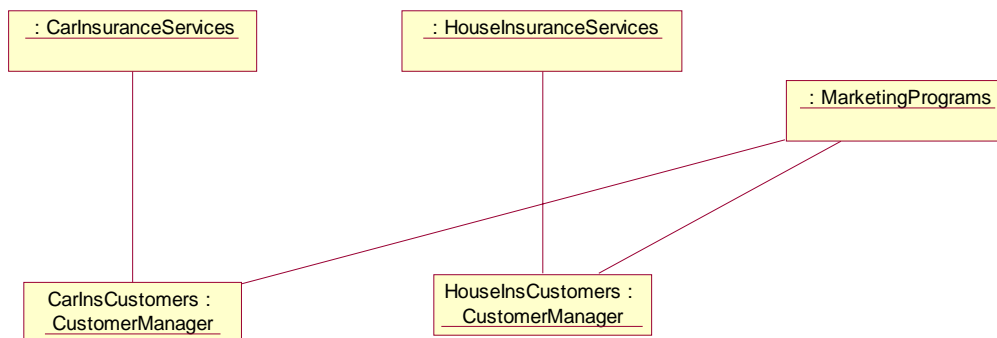


Figure 8 – Insurance Company FCO Architecture

As a second step, the opportunity would now exist to begin to unify the customer management services over time, assuming their operational requirements were compatible. There are a number of potential strategies for this: introducing a third implementation of Customer manager which delegated to the first two; switching all customer creation and update functionality over to one service only, but allowing read access to both instances – in parallel, data from one system could be moved to the other; switching such services to the new, third instance and moving off both existing systems, and so on. The important point is that once the functional interface has been introduced both the consumers and providers of the service can be changed with clearly understood and managed impact. Technology platforms can be changed or upgraded, new business processes can be piloted, legacy systems can be retired and so on, all under a standardized, managed functional service-oriented architecture.

Conclusion

As usual, things are going in the right direction, but we aren't there yet. Web Services in particular offer a number of elements needed for a flexible coupling solution, but current

patterns of application still represent a technology-level perspective on the problem. If we can leverage some of the features that Web Services offer, combined with potential standardization of cross-technology infrastructure services, held together by a standardized Functional Component perspective, and good SOA principles and processes, we will loosen the coupling between the application layer and the technology layer and provide much more flexibility within IT departments to meet business requirements.

But all this flexibility will need careful management, processes, standards and patterns. Part 1 of this series introduced a reference model for SOA, highlighting the need to standardize on some of the fundamental runtime concepts. This article has provided the detail of a pattern for flexible coupling in that runtime, based on clear definitions of Functional Component and Service. An important shift is that responsibility for the provision of technology bindings lies with the provider, not the consumer. When publishing services, providers need to be moving towards the publication of functional interfaces, not technology interfaces. In turn this will make the published services much simpler to access which will increase their take up.

One of the key missing pieces is the standardization of infrastructure service interoperability. However, businesses must also be pragmatic and take a view on what degree of technology independence is worth having and in what timeframe. For example, it may be that the functional decoupling gives a lot of benefits from an application software evolution point of view and allows certain types of technical architecture changes (e.g. changes to communication protocols), but infrastructure service separation is not a strong requirement for the priority use cases to be supported, since consumers and providers run on exactly the same technology platforms (and versions of those platforms). In this case the operational context can be passed as-is, in its technology-specific form, without the need to translate it back and forth into neutral formats.

The first step for an IT department looking for a way forward is to understand fully their existing application architectures and technology dependencies. This understanding can then be used to define a roadmap towards SOA at the functional level in tune with their technology platform and legacy system migration plans. For example, the impact of upgrading the application server version can be scoped and plans made. Moving towards functional interfaces is a good step forward because it provides opportunities for change without mandating particular timescales – this allows *incremental* evolution of the application architecture. For example, the introduction of a technology-independent functional interface does not preclude existing consumers continuing to directly access technology interfaces, but does allow new and upgraded functionality to take the technology-independent route.

References

- [1] <http://www-106.ibm.com/developerworks/webservices/library/ws-secrtrans/>
- [2] CBDi Best Practice Report – The SOA Reference Model, March 2004
- [3] Enterprise Solution Patterns using Microsoft .NET, version 2.0, chapter 6; Microsoft Press
- [4] WSDL 2.0 - <http://webservices.xml.com/lpt/a/ws/2004/05/19/wsd2.html>
- [5] OASIS Web Service Security Standard, <http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf>