

The SOA Reference Model¹

By John Cheesman, Georgios Ntinolazos

Abstract

This is the first in a series of articles in which we provide precise guidance on implementing SOA. This builds upon and further details many of concepts introduced in our five part series last year, and will progressively create a common reference model and process for SOA.

Introduction

Regular readers of CBDI will be familiar with the notion that SOA is about more than Web Services - it's an approach to architecture in which the capabilities are provided and consumed as services. This approach is applicable to all types of capability and behaviors.

Not surprisingly SOA is not a completely new approach; in fact it has firm foundations in Design by Contract (DbC) and Component Based Development (CBD). However whilst there are good lessons to be learnt from these disciplines, they are incomplete in their support for a SOA, requiring new and or revised concepts to address the significant differences between the "interface" and "service" concepts, as well as issues such as loose coupling of components, runtime discovery, use and replacement, and technology independence.

As with any complex subject, it's difficult to make significant headway into comprehending its concepts and principles, or understanding how it compares to other approaches and product offerings, without some common reference points. This is certainly true of service-orientation, component-based development and distributed computing platforms.

This article sets out a reference model for SOA as a whole, summarized in Figure 1, and takes a brief tour through some of the areas, particularly looking at the underlying concepts and the SOA Application Reference Model. Subsequent articles will further develop each of these areas.

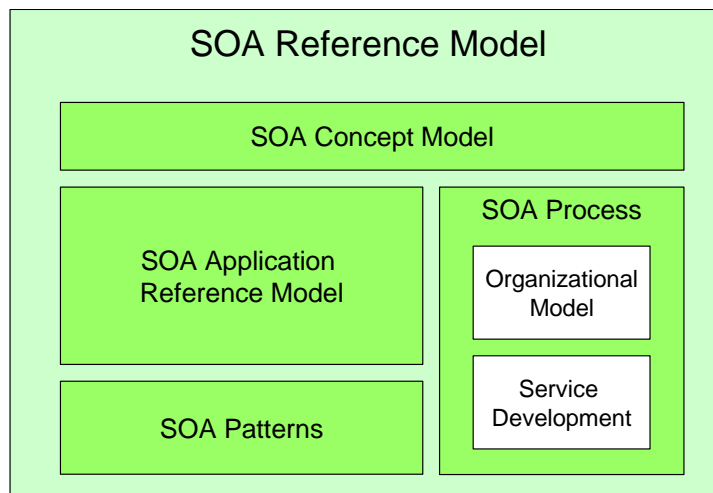


Figure 1 – Elements of SOA

The main areas of the reference model are as follows:

- SOA Concept Model – the underlying concepts used by the other areas. This model includes service, component, interface, assembly and so on.

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

- SOA Application Reference Model – a standard for the organization of the application layer which bridges the business and technology layers and creates the backdrop to enable service traceability throughout the development and deployment lifecycle.
- SOA Process – this is really a family of processes or approaches which describe how to actually implement SOA as a business. It covers a whole host of areas but in particular defines the organizational models that need to be adopted and the service development workflows – a full lifecycle process for creating software services
- SOA Patterns – a library of best practice in the application of both SOA concepts and the application reference model. This includes for example, different types of component, architecture styles, layering models and principles and so on.

The Application Layer

Let's start by looking at the traditional "application" layer in the business-application-technology stack (Figure 2). Applications are the bridge between business and technology. Applications exist to automate business processes (or some pieces of a business process) thereby making those processes more efficient and effective. They are therefore constantly leveraging technology evolutions to provide improved automation of the business processes, and in turn the business processes evolve to take advantage of new technologies and new application opportunities.

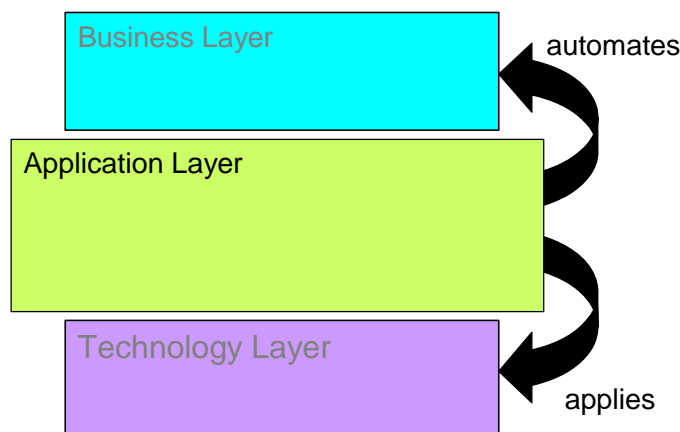


Figure 2 – The Application Layer – bridging business and technology

The application layer has a rather difficult job. Rather like a tectonic plate boundary, it has to bridge two worlds that are constantly changing or under pressure to change, and which have a degree inertia and a lot of momentum. To avoid looking like an earthquake zone it needs to exhibit a number of important characteristics:

- Flexibility – existing automated business processes can be updated or re-configured to support new business processes as required. This means managing the dependency relationship between the business process element and the process automation element (what we can call the "service")
- Technology-"independence" – this means managing the dependency relationship between the automation element and the implementation technology. This can be characterized as technology-independence but is perhaps more accurately the ability to manage multiple implementation technologies. This then includes the ability to integrate legacy systems and COTS products as well as to accommodate multiple technology platforms.

Further, business priorities dictate that changes need to happen quickly (we are now told that applications must be not only "flexible" but "agile"). This in turn means that these dependencies

must accommodate rapid change – perhaps the business wants to standardize on its software support for customer-facing processes, for example. This would mean updating all applications that automated elements of such processes and creating a standardized set of automated services. It would also mean ensuring that all the current implementations of such services could be surfaced in that form. How would that look on your company's IT-impact Richter scale?

Service - the Business boundary

Before we take a look at the application layer itself it's helpful to think about exactly what the boundary is with the business layer. A good way of assess the business side of the line is to think in terms of so-called "business elements" [5]. A business element is a business process, a resource or an organizational unit. Sims took this idea forward in last year's SOA series [2] to define some specific kinds of business element, summarized again here:

- Resource Business Element (RBE) – an independent set of related business resources, typically clustered around a single "focus" resource. For example, a Customer RBE might have a Customer resource at its core, with auxiliary resources such as Address, History, Profile and so on.
- Service Business Element (SBE) – a grouping of those aspects or steps of a business process which can be considered atomic and are potentially fully automatable. If the business analysis were being done with automation in mind then an SBE might correspond to the system responsibilities of a use case specification [1]. For example, in the OpenAccount business process there may be steps such as IdentifyCustomer and CreateNewAccount. The SBE would group such automatable steps together into a logical unit, organized by business process.
- Delivery Business Element (DBE) – a combination of RBEs and SBEs which delivers value to the business. For example, combining a Customer RBE together with SBEs that relate principally to the Customer resource would make a sensible combination of business responsibilities and would likely belong to a particular organizational unit (such as the Customer Services department).

In some situations it may seem appropriate to organize steps into SBEs by resource rather than business process in which case the distinction between an SBE and a DBE may be unnecessary. However, having DBE as a distinct concept allows SBEs to be grouped in a variety of ways and different patterns of application to be adopted.

The nice thing about the business element concept set is that it gives us some clear, business-oriented service groupings to refer to, which are independent of the process used to define them. For example, an SBE may represent a particular set of use cases or business stories as required by the project scope. Or it may be a particular set of business process steps that have a common requirement such as auditing. However, RBEs, SBEs, DBEs are how they are scoped is a SOA process matter and we'll cover that later in the series.

The main point here is that the fundamental bridge concept between the business layer and the application layer is the **service**. It is the atomic unit of automation from a *business* perspective.

To keep terminology clear we will qualify the service term (see Figure 3).

- business service is the name for the service identified in the business layer
- software service is the name for that aspect of the software implementation which corresponds to the business service

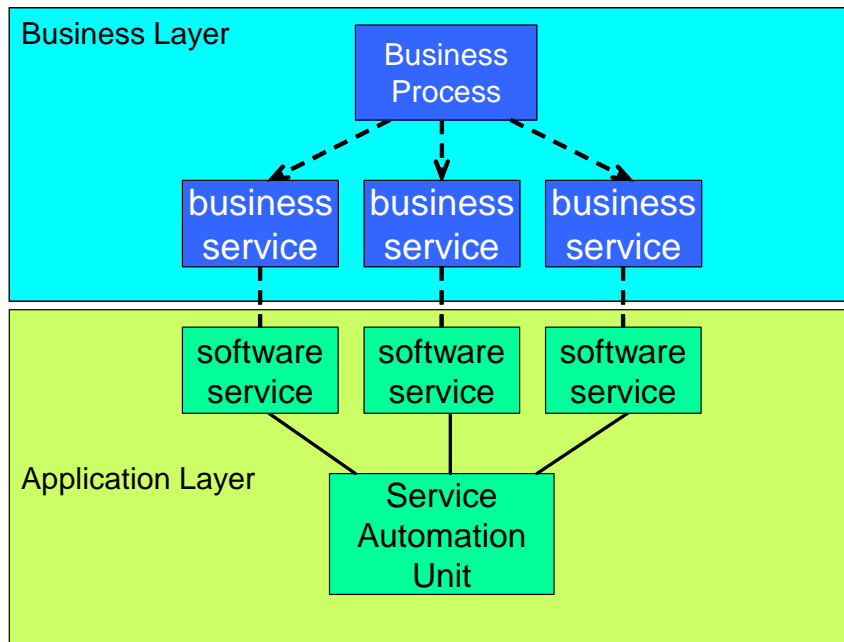


Figure 3 – Services define the business-application layer boundary

Now, from a service development process point of view we have a number of implementation options regarding the scope of the services we automate and deliver to the business. A simple approach would be to deliver a single automation unit that offers the services required by a particular DBE and for a small DBE this might be a good choice – this automation unit is traditionally called an application (hence the name of this layer) although its scope may have been defined using other criteria than the scope of the DBE. For more complex DBEs it may traditionally correspond to (a subset of) a whole range of different of applications.

One of the key principles of SOA is a new perspective on the basic delivery unit of the application layer. Rather than it being the traditional notion of an application, it is a service automation unit whose scope will depend on your SOA process. Who knows, in the future we might even start calling these units “applications”. In established component-based architecture approaches [1, 7] they correspond very nicely with the concept of a system-component.

Scoping the delivery unit of the application layer to align with the requirements of the business provides an important flexibility characteristic to software development. Business process change can then be more readily accommodated since the application layer is fundamentally organized in business terms.

Organizing the Application Layer

Keeping terminology straight is one of the main goals of the SOA Application Reference Model. The application layer unfortunately adds a level of terminology complexity that we don’t often need to address at the business level and that is the distinction between development-time (or design-time) and run-time. However, ignoring this distinction when considering the application layer is a recipe for confusion or hand-waving or both so we’ll tackle it head-on. Two of the key concepts in the application layer are component and interface so let’s lay out some terminology distinctions there first, and relate things to services. We’ll then consider the relationship between the application layer and the technology layer and some of the design patterns and standards we can use to achieve technology independence in the functional architecture. Finally we’ll return to the need to standardize and abstract away all the technology “glue” code via an application platform.

Functional Components

A software service is a runtime concept and is provided by a Functional Component Object through one or more runtime interfaces.

One of the difficulties with formalizing component-based approaches over the years has been the ongoing difficulty in standardizing on concepts and terminology. The term “component” has natural appeal and has been widely used to cover a whole range of concepts. One way to make any progress is to establish a clear namespace for concepts by providing qualifiers, although even the qualifiers can have multiple uses (e.g. “business component” can also mean many things).

In a previous article reviewing the state of component-based development [6] we introduced a set of “component forms” which distinguished different aspects of a component during its lifecycle (its specification, its implementation, its deployment packaging, its runtime instance and so on) as well as the important viewpoint difference between a purely functional perspective and a technology-specific perspective. Some of the key points are repeated here and a summary concept diagram is shown in Figure 5.

A functional component (FC) is a development-time concept. It is an encapsulated piece of software that offers services (at runtime) through functional interfaces. It has an independent software development lifecycle and may be independently deployed (although as we’ll see later it typically would be deployed with others to create a co-ordinated runtime assembly).

At runtime we have functional component objects (FCOs), or component instances as they are also known [8]. A key difference between development time and runtime is that the runtime object has *state* (see Figure 4).

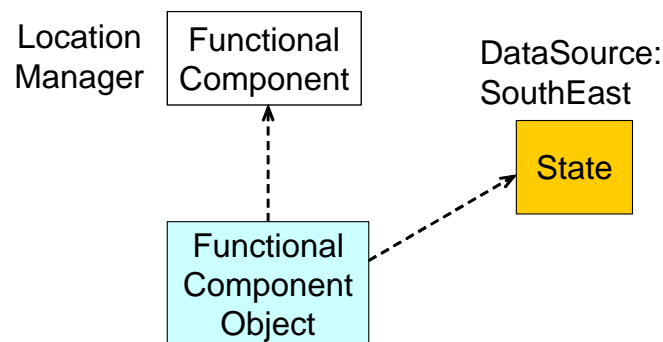


Figure 4 – Runtime objects combine software and state

For example, you may have a LocationManager FC which handles information about geographical locations such as addresses, grid references and so on. Whilst you may find its interface definitions very interesting, the services the FCO offers at runtime may be of little value without some defined FCO state – the findLocation() operation is pretty useless without some Locations to find.

Now this is not true of all FCOs. If the service is inherently “stateless” (i.e., it is a function) and the results depend totally on the provided input then it may still provide a useful service. For example, a Fahrenheit-Centigrade converter needs no state of its own. Alternatively, some FCOs have fairly static or constant state. For example, an Income Tax calculation service mainly operates using the information supplied to it, but it does rely on a number of internal constants such as the tax rates, tax band levels and so on. This state is relatively constant but may change from year to year in which case the service would need to be redeployed, or potentially reconfigured at runtime. Many of the current public examples of web services are of

this stateless nature. Having no state also alleviates many of the complexities involved with security and transactions.

Although FCs are the unit of independent development, this does not mean that the FCOs are independent at runtime. To operate they may require the services of other FCOs and to ensure that an FC is truly independent and encapsulated these dependencies must be captured in the FC specification. This is done by specifying an FC in terms of both the interfaces it provides and the interfaces it requires (Figure 5). It is then the assembler's job to ensure that these dependencies are resolved at runtime – more on assemblies later.

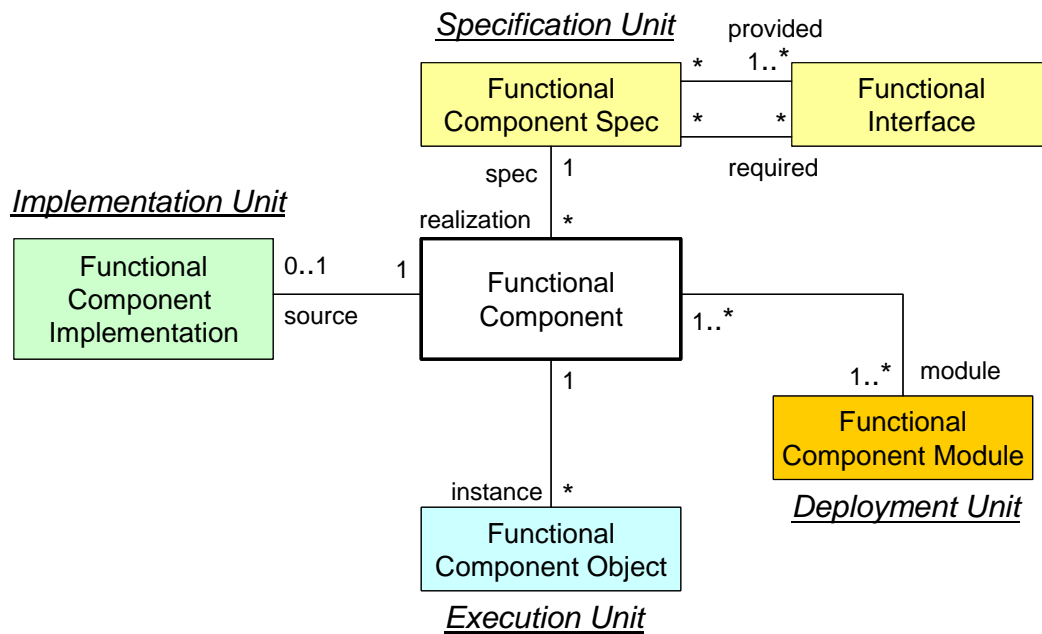


Figure 5 – Functional Component Forms

Coming back to services, we can now state that a software service is a runtime concept and is provided by an FCO through one or more runtime interfaces (Figure 6). The service represents a business perspective and maps on to the operations of runtime functional interfaces. In the diagram this is shown as a many-many relationship between service and interface and, as discussed above, the details of this mapping are more to do with the SOA process, which we will detail in later articles, than the reference model per se. Working top-down it would certainly be natural to organize functional components and their interfaces to align with SBEs, RBEs or DBEs for example, but there are also other policies that can be applied. There may be many other constraints on the form of the functional interface and the scope of a functional component if working bottom-up.

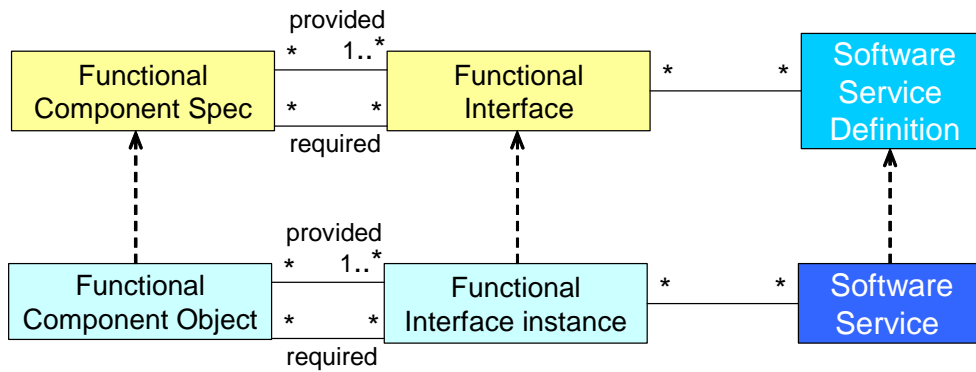


Figure 6 – Functional Components and Services

The Technology boundary

The Functional Component concept is a mechanism for clearly separating and managing the boundary between the application layer and the technology layer. FCs provide a homogeneous construct for functional application architecture, allowing the implementation and deployment details to be managed separately.

FC specifications are therefore technology-platform-independent and may be defined in any appropriate specification language. This distinguishes FCs from technology component (TC) concepts like Enterprise JavaBeans (EJBs) or .NET components – these are part of the FC implementation. Further, FCs may not use distributed component technology platforms at all, but be implemented using legacy and COTS systems. The FC concept therefore provides a technology-neutral application architecture concept and the link to the technology platform occurs inside the FC (see Figure 7). This allows of course many different technology platforms to be involved in the technology layer (Application servers, Web Servers, Message Queues, Transaction Processing Monitors, and so on). One important consideration therefore when defining functional component specifications is ensuring that any required functional execution context can be passed between FCOs (e.g. transaction status). It is a requirement of the FC implementation to ensure that the functional execution context can be applied appropriately using the technology platform it has chosen.

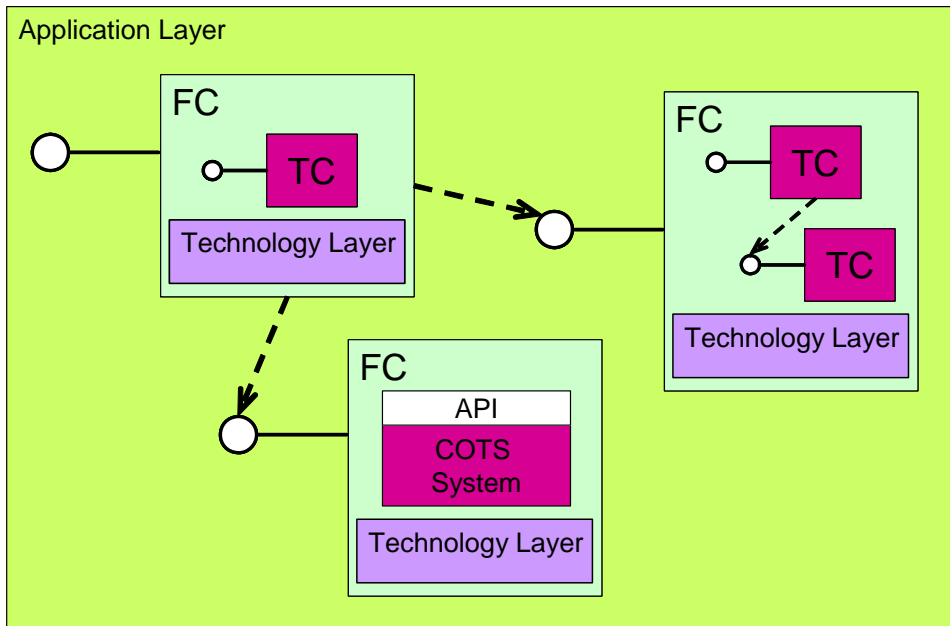


Figure 7 – Functional Components manage the Technology Layer boundary

One of the difficulties in applying component concepts fully at the application layer level has been the technology-related constraints in the TC platforms (e.g., communication protocols, TC lifecycle management, Container service interoperability, etc). These constraints significantly impede flexible solution composition. The FC approach employs proven component-based software engineering principles, and builds an abstraction layer on top of the current technology to alleviate these problems.

Functional Component Model

An important aspect of application architectural flexibility is *loose-coupling* between FCs whereby the consuming FC defines the functional interfaces it needs independently of the functional interfaces provided by others.

To ensure that the interfaces of an FC are purely functional an FC implementation needs to include a technology-binding element to the technology-specific interface(s) being employed internally. The internal interface might be an EJB Home or Remote Interface, a Message Queue message, or a COTS system API for example (see Figure 8).

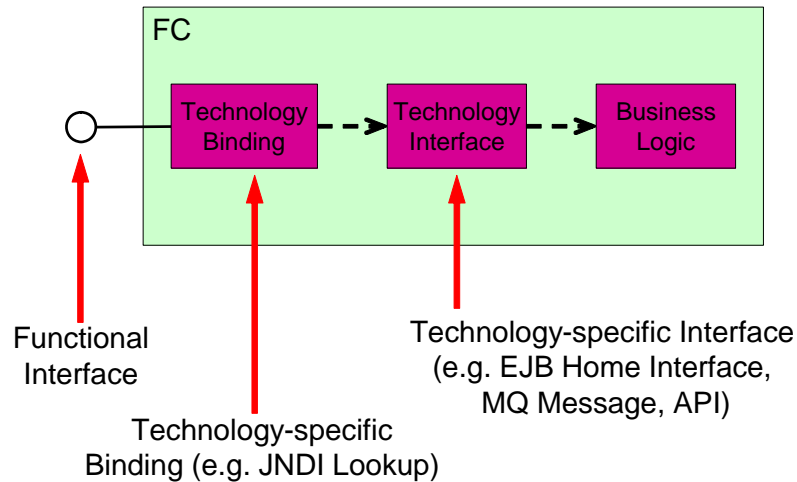


Figure 8 – Organization of a Functional Component

This separation of functional interface and technology-binding is part of the fundamental functional component model which forms part of the SOA application reference model. The technology binding piece can be packaged separately and become mobile code, allowing the FC implementation to be upgraded, adopting new technologies without impacting the client - a new technology-binding can be dynamically deployed to the client without changing the functional interface.

How do web services and WSDL fit in?

Although the functional interface definition defines the operations that can be invoked it does not define the communication protocol to be used when invoking it (e.g. SOAP, RMI). In order for one FC to use services provided by another FC there must also be agreement on this protocol.

Web services fit very neatly with this model and the distinction between functional interface and communication protocol. The Web Service Definition Language (WSDL) provides two main aspects of a software service definition:

1. technology-independent definition of the operations and parameters of a “service”. This corresponds directly with the concept of Functional Interface.
2. specification of a particular communication protocol (e.g. SOAP)

An advantage of using the web services standard is that the functional interface to be used by the client FC and the supplier FC can be generated into the programming language of choice in each FC and generated proxy code handles the translation and communication in a similar way to CORBA IDL. This means that FC developers see the functional interface represented in their chosen implementation language. Note however, this is a programming language binding (e.g. to Java) and not a technology-platform binding (e.g. to EJB). This second step is provided by the FC developer as part of the technology binding.

However, this approach is predicated on the agreement of functional interfaces (and protocols) by the connected parties. Another important aspect of application architectural flexibility is *loose-coupling* between FCs whereby the consuming FC defines the functional interfaces it needs independently of the functional interfaces provided by others. This makes the specification and implementation of an FC totally independent of any other functional specification and shifts the responsibility for aligning functional interfaces to the assembling architect, which is logical place for it. This architectural benefit this approach provides can be achieved by introducing explicit connectors into the runtime assembly. Loosely-coupled

assemblies also introduces the integration problem of FC contract alignment and whose responsibility it is – the FC supplier or the assembler or both. We will discuss this topic further in subsequent articles.

Assemblies

Assembly is the process of linking together runtime software elements. In our case this means identifying a set of FCOs (and other assembly elements such as connectors and user interface elements where appropriate) and linking them together so that their external dependencies are resolved and consistent with the planned architecture (see Figure 9). We call the result of this process a Functional Assembly (the term Assembly alone already has multiple connotations). The appropriate FCOs working in the context of an assembly will then constitute one or more Service Automation Units as shown in Figure 3.

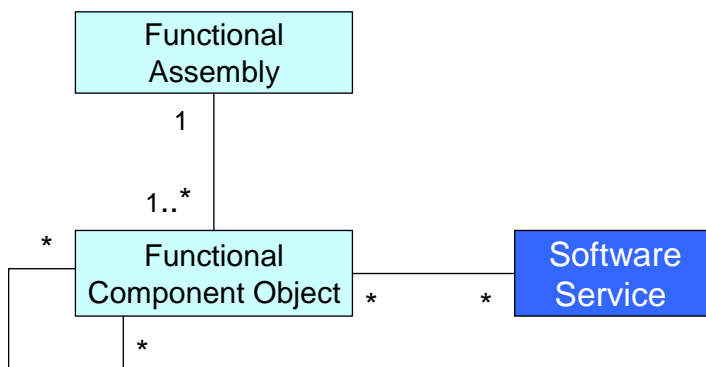


Figure 9 – Functional Assembly

As with FC design there are important parallels between the creation of the functional assembly and the technology-specific concept of an assembly of TCs, but they are distinct constructs. In particular it is important to understand the Functional Assembly structure in order to populate the deployment information of the TCs correctly. In fact, this has always been the case, but the fact of making the FCO and Functional Assembly notions explicit serves as an important tool for validating TC deployment descriptors.

Service Discovery and Dynamic Assembly

Using the Service Discovery Service, an FCO allows for runtime assembly re-configuration. The Functional Assembly provides the appropriate runtime environment context for an FCO. It allows each FCO to assume that the services it needs are available within the assembly (after all, part of the process of assembling FCs was to ensure that this was the case). This means that each FCO can operate without worrying about the lifecycle of the other FCOs in the assembly – they can be instantiated independently. This is a different form of loose-coupling – runtime lifecycle-independence. This can be achieved by providing one well-known service within the assembly – a service-discovery service (SDS), and registering the assembly information for the SDS to access via a registration service (see Figure 10).

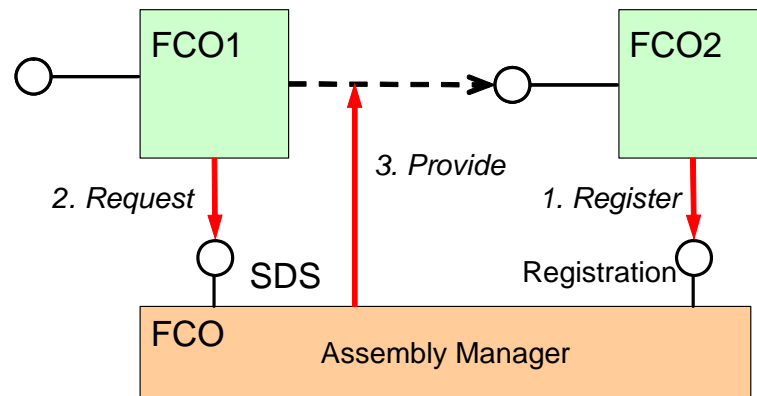


Figure 10 – Service Discovery

Using the SDS, an FCO can gain access to the services it requires through a simple call. It is insulated from both the way that service is provided (e.g. how the FCO is implemented) and from the assembly configuration itself. It also allows for dynamic (runtime) assembly re-configuration by changing the FCOs in the assembly. FCOs can be notified of the need to refresh their service connections via the runtime environment context. Building in context sensitivity to FCOs allows them to adapt to changes in their runtime environment. For example, if a particular service such as CreditCardCheck needs replacing by another one due to increasingly poor quality of service characteristics this can be achieved dynamically.

Application Platform

Building software directly on top of a technology platform means that a certain proportion of the application or FC code will be related to accessing technology platform services. There are two main problems with this:

- Functional logic is mixed up with technology-platform logic. This means the developer needs to understand *both* how to implement the functional requirement in code (usually called the “business logic”) and how to write the code to use the technology-platform services (this code is sometimes referred to unceremoniously as “glue” [4]). The skills needed to write these two types of code are often quite different. Further, it increases the overall size of the code which in turn increases the maintenance overhead.
- The “same” technology-platform logic or glue code will appear throughout the business logic code but each glue code occurrence may vary since there are no constraints to ensure a standardized form. This also increases the maintenance overhead since code is replicated and gives rise to bugs since code variations may not conform to best practice.

A solution to these problems is to establish an application platform, above the level of the technology. This can provide a level of indirection between the business logic itself and the technology platform by providing framework-based standardized implementations of FCs and their design elements such as technology bindings. It can also incorporate standardized functional services which are employed by many FCs, for example, logging, error handling, audit trails, and so on), functional component standards and types such as assembly and environmental context definitions, and a standard implementation of a service discovery service. Going further, with sufficient richness in the functional component model it could provide declarative access to some of these services rather than requiring them to be explicitly requested. For example, simply tagging a functional operation as requiring audit control would result in any call to that operation being audited without placing any requirement on either the provider or the consumer of that service.

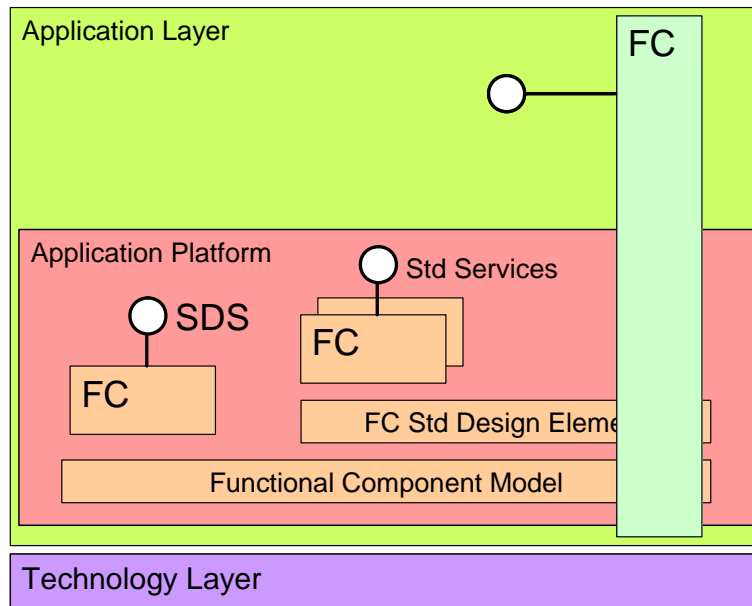


Figure 11 – Application Platform

The application platform abstracts above the various technology platforms found in the technology layer. Regardless of the technology-dependency of a given FC it can access the application platform services in a standard way. The functional component model and any FC Standard Design Elements would potentially need instantiating for each technology platform supported. Similarly, standard services could be duplicated on each technology platform if the quality of service due to cross-technology-platform access was found to be insufficient.

SOA Application Reference Model

If we now pull all these elements together we can define an SOA Application Reference Model as depicted in Figure 12.

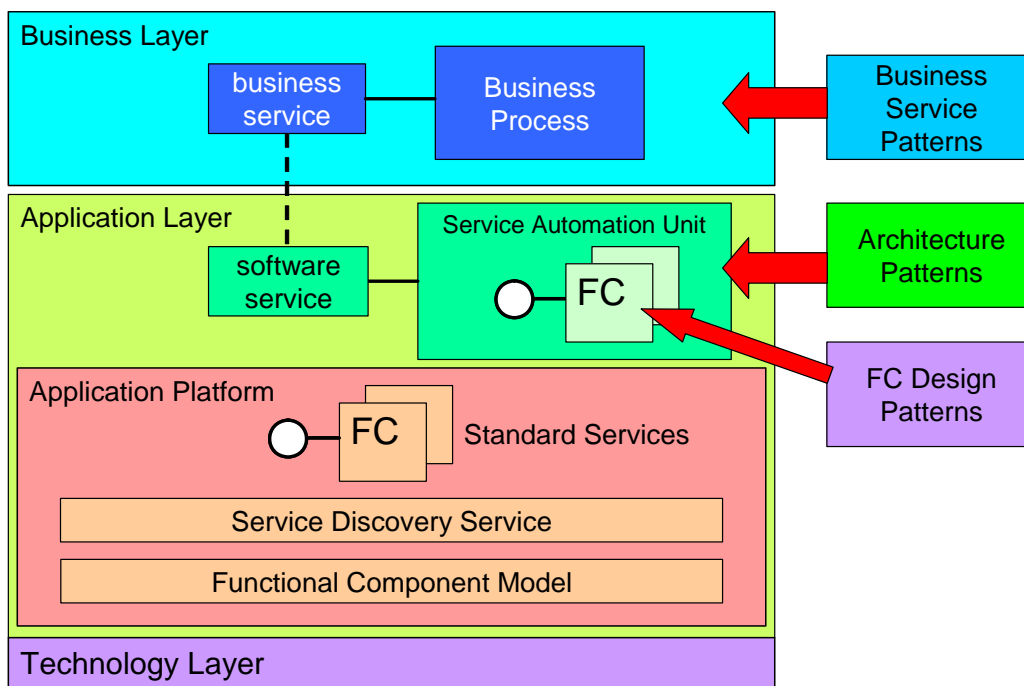


Figure 12 – SOA Application Reference Model

The unit of delivery to the business is the service automation unit and this comprises one or more functional components offering software services through their interfaces. Patterns apply at many levels and we have shown some of the most important ones.

Business service patterns define business service grouping constructs (such as SBE, DBE and so on) giving structure to the business services and providing a base for the SOA Process.

Architecture patterns define different types of FC (e.g. Process FC, Domain FC, System FC, Infrastructure FC, ...) and architectural styles and principles for the way to link these together. Application layering models as defined in [1], [3] and [6] provide good examples of such patterns as is the topical approach of creating an orchestration layer for underlying software services.

FC Design patterns provide support in both the use of the application platform and the technology. Many design patterns available today are effectively technology design patterns helping you to use, for example, web services, EJBs or .NET. These fit naturally within the FC Design pattern area.

Conclusion

We have taken a light tour round some of the main areas of the SOA Reference Model and perhaps one of the major themes to come across is its sheer scope. Hopefully it is not too daunting - orienting software provision to align with business services is a major change. But many of the key technology pieces are already in place. The trick is to apply these in the correct way at the application level.

SOA debates in the industry have suffered from a lack of precision thus far. The need for a reference model is becoming paramount to ensure all the parallel activities that are happening have some hope of being joined-up for the customer. We have not shied away from addressing the need for some of this precision, particularly the distinction between development time and runtime concepts, and the need to maintain a functional view *throughout* the lifecycle, not just during analysis and design.

An ongoing industry trend has been the rise and rise of the platform, from operating system, to transaction processing monitors and databases, to distributed object systems, to web servers and application servers. This trend is inexorable. The logical next step is to include application layer services as we have described in this article and establish an application platform. However, it is important that the fundamental concept distinctions are in place (for example the difference between Functional Component and Technology Component). Some of the product directions we see happening at the moment would indicate that they are missing, with, for example, business workflow products linking directly into technology components and missing out the critical functional layer.

Architecture, as always, continues to be important, but with increasing complexity, flexibility and change in an SOA world, understanding architectures becomes more difficult. The introduction and standardization of SOA concepts is critical and in turn will create standards for new architectural viewpoints such as FC specification architectures and FCO assemblies.

Perhaps one of the most important differences that SOA adds is the move away from the fixed application to the dynamic assembly. This possibility tends to give system testers nightmares, but is critical for businesses to be able to compete and remain agile in their marketplaces. With the business drivers clearly there solutions to some of the arising technical challenges such as "what does it mean to test a dynamic assembly?" will need to be found and new ways of thinking established.

So where do we go from here?

To turn theory into reality we will be focusing the next article on a real world example. This will help to understand some of the new SOA concepts and structures and act as a reference point for subsequent articles on the SOA Process which is where we take the series after that.

References

1. John Cheesman and John Daniels, "UML Components", Addison Wesley, 2000
2. CBDi Best Practice Report – SOA Part 2 – the Bridge, April 2003
3. CBDi Best Practice Report – SOA Part 3 – Federation, May 2003
4. CBDi Best Practice Report – SOA Part 4 – the Platform, June 2003
5. David Taylor, "Business Engineering with Object Technology", Wiley 1995
6. CBDi Roadmap Report – Component-based Development – Where next?, July/August 2002
7. Peter Herzum and Oliver Sims, "Business Component Factory", Wiley 2000
8. Clemens Szyperski, "Component Software", Addison Wesley, 1999