Server-Side Components - A Pattern Language

(c) 2002 Markus Völter, MATHEMA AG, Germany

markus.voelter@mathema.de

Version 2, Feb 25, 2002

This pattern language defines an abstract architecture for server-side component infrastructures. It focuses on implementing a technical infrastructure for component based systems. Examples of such infrastructures are Enterprise Java Beans, the CORBA Component Model or COM+ (COM and MTS). It does not contain basic distribution patterns such as *Proxy* or *Broker*.

The paper consists of several sections. Each of them is introduced briefly here. **Intended Audience**, the next chapter, defines who should read the pattern language, and why. **More information about the technologies** hints at some literature and web resource that provide greater levels of detail for each of the example technologies EJB, COM+ and CCM. **Pattern Form** describes the form of each of the patterns in the language (Alexandrian). The next section, **A short Story** talks about a fictional dialog of a software developer and a consultant to put the patterns into context. This story appears several times throughout the language.

This edition of the language does not contain all the patterns that have been submitted in the EuroPLoP version. This is because the patterns have become much longer in the meantime. We left some of them away for space reasons. If you want to see all of them, either contact the author or have a look at the book "Server Component Patterns – component infrastructures illustrated with EJB" by Voelter, Schmid and Wolff. To be published by Wiley in Summer 2002.

Intended Audience

The pattern language can be used for several distinct purposes, and can therefore be used by different people:

People who need to work with EJB, CCM or COM+ can use the language to gain insight into the basic principles which are employed in the technology.

People who need to decide which of the three component technologies they should use, and therefore need to find commonalties and differences can use the language to understand the basic principles and common aspects of the different technologies.

Although many applications are built based on EJB, CCM or COM+, there are cases when these technologies cannot be used and a new, usually specialized component infrastructure must be built for a system. Here, the pattern language can help to outline the basic building blocks which make up a component infrastructure.

More Information about the technologies

The information found in this pattern language is not enough to really start working with the respective technologies. Therefore, I have included a couple of references to information on the technologies themselves.

- **Enterprise Java Beans (EJB):** A good introduction to EJB is Monson-Haefel's Enterprise Java Beans [MH00]. Of course, Suns EJB Specification [SUNEJB] is also worth reading. Java Server Pages [JSP00] gives a broader look at J2EE, including EJB.
- **CORBA Component Model (CCM):** The CCM is still quite new, therefore there is not too much available. The Specification by the OMG [OMGCCM] is of course very informative, Jon Siegel's CORBA 3 book [JS00] also contains a chapter on CCM.
- **Component Object Model + (COM+):** The book market is full of titles about this topic, I found Alan Gordon's COM+ Primer [AG00] to be a good introduction.

For information on these and other OO and component technologies you can also refer to Cetus-Links [CETUS]. For more, detailed patterns about distributed objects and server architecture I recommend [POSA2].

Pattern Form

The patterns in this language are written using Alexandrian form. This section briefly describes this pattern form.

Each pattern begins with its name, usually a short, one or two word phrase. The first section of the pattern is the pattern's context, which describes other patterns which must be implemented before the current pattern can be implemented. Then the pattern's problem description follows in bold font. After that, the pattern's body describes the problem in more detail, provides rationales, background information etc. The next section, again in bold font, provides the solution to the problem. The following section describes the resulting context, i.e. which consequences the application of the pattern has, and which other patterns can be applied when the current pattern has been implemented. The last section, in italics font, describes examples and known uses. Usually, the examples are taken from well-known component technologies, such as Enterprise Java Beans, CORBA Component Model and COM+. References to other patterns within this language are given using CAPITALIZED FONT.

A short story

For a pattern language is it critical that you, the reader, understand how the different patterns go together. You need to get the big picture. Thus, we want to start introducing the patterns with a hypothetical conversation between two people. One is a a component software consultant and the other one is a Java programmer from a large company's development staff (his statements are given in *italics*). If you don't understand all of it – don't worry. Just read the patterns and come back to the conversation afterwards. Part three of this book provides a much more detailes conversation based on EJB.

- - -

Ok, we have to do this new enterprise system for our customer iMagix, Inc. They want us to create an eBusiness solution for their rapidly growing internet site. They expect significant growth in the future – so the system

must be scalable. And because they have strong competitors, they must be able to add new features quickly to always be one step ahead of them... You know, the usual things, and it must be out until next November. This is just six months to go...

Is it just this system they want to build, or is the system a starting point for further systems. What I want to say is: Is reuse necessary and welcomed?

Mmh, yes, I think so.

Are they focussed on a specific technology?

They say, we're free to do anything, as long as it works fine… But looking at their current development, they seem to like Java.

Ever thought of using a component-based approach?

Of course! I mean, everything is component based today. Another buzzword. Can't get around it, can you?

Yes, but actually, beyond the hype, CBD[1] has some pretty neat advantages. Should we talk about these a little bit?

Well, why not. After all, that's why we hired you ...

Ok. So the basic assumptions are two things: You want to build an N-TIER SYSTEM in order to separate user interface from data storage and from business logic, and to distribute your system over several machines for load-balancing and failover. The second thing is: you want to SEPARATE CONCERNS.

What's that?

In this context, that means that you explicitly distinguish between functional aspects of your application - usually called business logic - and technical requirements, such as transactions, scalability, persistence, security…

Yeah, but why? I mean you always somehow separate the two, but…

But: If you do this systematically, you can buy something that takes care of your technical requirements. A so-called CONTAINER. This saves a lot of work.

Ok, sounds interesting. And where do I put my business logic?

You put it into COMPONENTS, which live in the CONTAINER. Each COMPONENT represents a process, a business entity or a service. Together, COMPONENTS and the CONTAINER make up the business tier of the application. Of course, you might need databases and webservers/browsers in addition...

And what does that mean regarding maintenance and evolution of the software? Sounds like I have my application built from separate "blocks of functionality".

First thing is, you can reuse technical and functional concerns separately, because they are kept separate throughout the whole lifecycle of your application. You can buy new version of your CONTAINER without changing the COMPONENTS. Second, your COMPONENTS need to have some specific characteristics to make all this work. Each COMPONENT must only be accessed via a well-

defined COMPONENT INTERFACE. And each COMPONENT must exhibit a certain degree of FUNCTIONAL VARIABILITY, or flexibility, in order to be reused in several contexts.

But life isn't that simple! You cannot completely separate technical and functional concerns. For example, you have to demarcate transactions or check security restrictions…

Yes that's why you use so-called ANNOTATIONS. These let you specify - not program - the way the CONTAINER should handle your COMPONENTS, from a technical point of view.

Mmmmmh, ok. Sounds as if that could work. So, we have a component for each business entity and process - can get quite many over time… The, how did you call it?

Container ?

Yes, CONTAINER, the CONTAINER probably needs to do something to keep resource usage acceptable, especially memory. I mean, each might need significant amounts of memory.

Yes, absolutely. Basically, the CONTAINER uses only VIRTUAL INSTANCES. There is a logical identity, for example a customer James Bons, and the physical identity of a specific component instance - both can be mapped as required using PASSIVATION and INSTANCE POOLING. This helps to manage resources effectively.

Ok, I understand that the CONTAINER assigns logical identities to physical identities as needed - but how can he know which one is needed?

Basically it uses a COMPONENT PROXY as a placeholder for the real instance, and it uses it to "catch" an incoming request and make sure that the required logical identity has a physical instance assigned. And by the way: LIFECYCLE CALLBACKS are used to let a COMPONENT instance impersonate different logical identities over time.

You're beginning to convince me. But hmm, how do I get in touch with these COMPONENTS, I mean, basically they are some kind of remote object, aren't they? At least, we want to have them remotely accessible.

Yes you're right. There is a two step process to obtain a reference to a COMPONENT instance. You use COMPONENT HOMES to create the concrete instances and a NAMING service to get a reference to the home.

Ok, I understand NAMING, I know it from CORBA. But what is a COMPONENT HOME?

You know the GoF [GHJV94] book?

Of course!

Good. So, a COMPONENT HOME is basically an application of the factory pattern, it manages all instances of a specific COMPONENT type.

Mmh, you talked of ANNOTATIONS which determine the behavior of the CONTAINER. Isn't it too much overhead for the CONTAINER to look at these ANNOTATIONS all the time at runtime?

Yes it would be if these ANNOTATIONS were interpreted at runtime. But there is an additional step called COMPONENT INSTALLATION, where, among other things, the ANNOTATIONS are read by the CONTAINER and a GLUE-CODE LAYER is generated. This code is directly executed at runtime and serves as an adapter between the generic CONTAINER and specific COMPONENTS with their specific technical requirements.

But listen, a COMPONENT cannot live on its own. You have to access certain parts of the environment at times, for example for database connections, or – I guess – to look up other COMPONENTS.

Right. The CONTAINER provides each COMPONENT instance with a COMPONENT CONTEXT, to access the world outside the COMPONENT. Or at least parts of it. Only those parts, the CONTAINER wants the COMPONENT to access.

Yes, but…

And! In addition, the CONTAINER also MANAGES RESOURCES. It automatically creates and control database connection pools, for example. Your COMPONENTS can use these pools without caring about how and when the connections are created.

Ok ok, I give up! One last question: If the CONTAINER manages transactions and security for me, then he must have more information than what is available in a regular method call. I know that CORBA OTS uses a mechanism called context propagation. Is this also true for CBD?

Yes, we generally call it the INVOCATION CONTEXT. The transport protocols allow for that.

- - -

After this discussion, the software company started to look closer at component technologies. Several weeks later, the programmer came up with a couple of additional questions, or observations. Once again, he talked to the consultant.

- - -

Hi, how do you do?

I'm fine. What about you and your COMPONENTS?

We're doing quite well, thank you. However, while looking at EJB, COM+ and CCM, a found some interesting things I'd like to discuss with you. For example, they have several types of COMPONENTS.

Yes, there are ENTITY, SERVICE and SESSION COMPONENTS. Each type has its own characteristics regarding persistence, concurrency, etc. The persistent one needs a PRIMARY KEY for identification. References to the others can be stored persistently using a HANDLE.

Right. And in addition, all COMPONENT models impose some IMPLEMENTATION RESTRICTIONS on the COMPONENTS, to make sure they can run in the CONTAINER without compromising their own stability. More, they distinguish two kinds of errors: application errors and SYSTEM ERRORS. The Container provides some default error handling for the latter, such as aborting the current transaction.

Right. It seems like you have learned quite a bit. There are other interesting things. For example, COMPONENT Introspection can be used to help building a graphical ANNOTATION builder tool.

But tell me, what kind of support is there to really build complete applications from these components? Is there any help for assembling these?

Yes. Some component architectures provide a way to specify which other REQUIRED INTERFACES a component needs in order to function correctly. COMPONENT PACKAGES and ASSEMBLY PACKAGES are also important aspects. They help to distribute complete COMPONENTS.

Yes, and I learned that in some architectures provide PLUGGABLE RESOURCES, kind of custom-defined MANAGED RESOURCES.

Indeed. PLUGGABLE RESOURCES help to integrate your CONTAINER with other, perhaps legacy, backend applications by defining an interface to plug them into the CONTAINER.

So, tell me, is all this reality today? Are COMPONENTS the next silver bullet that solves all of our problems?

No, certainly not. Today's COMPONENT architectures, no matter whether it is EJB, CCM or COM+ are far from perfect. They have some flaws in their design, and some commercially available CONTAINERS suffer from compatibility problems and the usual bugs – after all, it's still a quite new technology. But nevertheless, COMPONENTS can solve some problems quite neatly.

- - -

An they were happy and used COMPONENTS till the end of their career... We hope that this brief conversation provided some motivation to read further. After reading the patterns, it is a good idea to come back to this section and read it again. You will see it with other eyes, and the overall picture will become even clearer.

Some basic principles

The design and creation of every software architecture is guided by a set of principles. They can be seen as high level goals or guidelines. Understanding and accepting these principles is crucial in understanding the architecture, because they guide architectural decisions. *Information Hiding* is such a principle, it is important for understanding object oriented programming.

Component architectures are no exception, they also build on a set of principles. We consider the following principles to be important: *Separation of Concerns*, and *Multitier Systems.*

Separation of Concerns

In today's rapidly changing world, businesses and their processes are changing at a very fast pace, and it is crucial for a business to adapt its software systems to support these changing processes. Business systems are so complex and expensive, that it is crucial to preserve the investments, once they have been made. This is even true when the business and the requirements are changing significantly. Thus, the primary goal when building mission-critical applications is to make sure they can evolve, to adapt to changing requirements.

There is another kind of change to be taken care of: Technology. Every couple of months, new technological possibilities arise, and usually businesses want to use them. The reasons are many. To make sure that changes can be introduced into a system with a reasonable amount of work, changes should be localized, and a change in requirements should ideally result in a change at only one places in the system. The changes that can happen to a system can be grouped: you will easily see that changes to the business logic and changes to technology have different reasons and constraints. You want to make sure, that changes to one aspect do not require changes to other aspects. This principle is called *separation of concerns*. We can identify two fundamentally different concerns in typical business software: *Functional concerns* and *technical concerns*.

Key to efficient software development is the separation of concerns. This provides the following benefits: independent evolution, management of complexity, separation of roles, and reuse of technical infrastructure.

As a result of separating the concerns you have two “blobs”, one for the functional concerns, one for the technical concerns. To make up a complete application, they have to be integrated again. Component architectures provide this integration in a well-defined manner.

Multitier Systems

Not too long ago, a piece of software was more or less self-contained: a program running on a certain operating system. Later on, external resources like databases or message queues became an increasingly important part of IT application systems. Today, the requirements for mission-critical enterprise systems are different, some of these requirements are outlined below:

- An application has several different groups of users.
- For many applications today, scalability is an explicit requirement.
- Availability is another important concern.

A popular solution for many of these problems is the following: The system is structured into *layers* (see [POSA]). Each layer has a well-defined responsibility and accesses only the layers below itself to implement its functionality. In the case of distributed systems, each of these layers can be placed on a separate machine, and can be accessed remotely. A layer which is accessed remotely is called a *tier*.

The Patterns

This section contains the actual patterns.

Component

You applied SEPARATION OF CONCERNS. As a consequence, your system is divided into two distinct parts: the functional part and the technical concerns.

A single big chunk of functionality that is separated from the technical concerns is better than mixing the two concerns. However, the part is still just a big ball of mud: changes to a particular part of the system's functionality might still affect the whole system. Moreover, independent deployment and update of each part is rather impossible and you want to be able to reuse distinct “functional parts” of the system separately.

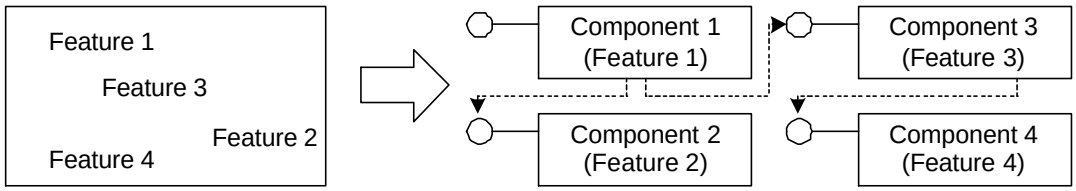
The primary reason for applying the principle of SEPARATION OF CONCERNS is to facilitate the evolution of a system and to promote reuse. Yet it is not sufficient to apply this principle at the most coarse-grained system level only. To be able to develop, evolve, reuse, handle, and combine the functionality of a system most effectively and efficiently, it is also important to separate different functional parts from each other. Ideally, system development happens by just assembling existing reusable functional parts.

Separating different functional parts is not sufficient enough though. In today's world, environments, markets, customers, requirements, and company structures change fairly quickly. Most such "external" changes also result in changes within the functional parts of a software system. However, immediate response to such change requests is only possible if we can localize changes and minimize the affects of changing a particular functional part onto other functional parts. For example, when

changing the implementation of a single functional part, we do not want to recompile the entire system. We also need to encapsulate our functional parts appropriately.

Therefore:

Decompose the functionality of your application into several distinct COMPONENTS, making each COMPONENT responsible for providing a specific self-reliant part of the overall functionality. Let every COMPONENT implement its responsibilities in full entirety, without introducing strong dependencies to other COMPONENTS. Compose the complete application by assembling the required functionality from these loosely coupled COMPONENTS.



There are two important concepts here: coupling and cohesion. A COMPONENT should exhibit a high level of cohesion, while being only loosely coupled to other COMPONENTS. This means that each COMPONENT should provide a set of features that naturally belong together, but it should not directly depend on the internals of other COMPONENTS. To achieve this loose coupling, COMPONENTS access each other only by COMPONENT INTERFACES. The COMPONENT INTERFACE is separate from the COMPONENT IMPLEMENTATION.

As a consequence of decomposing the functional concerns into COMPONENTS as described above, each COMPONENT becomes a small "application" in its own right. The application's functionalities are well encapsulated and localized. Changing a specific functionality only requires changes to one COMPONENT. Only this one COMPONENT has to be redeployed after a change.

As the COMPONENT INTERFACE is physically separated from the COMPONENT IMPLEMENTATION, clients need not be changed when the implementation of a COMPONENT is changed. In a way, this approach is a very sophisticated and consequent form of modularization.

Applications in turn, are created by "wiring" a set of COMPONENTS. The application itself is reduced to orchestrating the collaboration of the COMPONENTS. Different applications can be created by combining COMPONENTS in different ways. However, while an application might reuse a specific COMPONENT, it might require a slight modification of behavior or structure. Thus, the principle of VARIABILITY must be employed.

A COMPONENT implements only functional concerns. However, all functional concerns must be completed with technical concerns in order to fulfill the requirements of a particular system. Provide these technical requirements to COMPONENTS via a CONTAINER into which COMPONENT can be INSTALLED without affecting other COMPONENT as long as the COMPONENT INTERFACES remain stable.

Note that a COMPONENT is a passive entity. It waits for invocations from clients (which might be real remote clients or other COMPONENTS), executes the invoked operation, and then gets back to passive mode again. This means that a COMPONENT instance cannot start doing something on its own. This has important consequences for the way the CONTAINER can be implemented: the CONTAINER can safely assume, that if no method invocations arrive for a specific instance, the instance will never execute any code – thus it could for example be removed from memory until the next invocation arrives. See VIRTUAL INSTANCES and COMPONENT PROXY for details.

We already mentioned that a COMPONENT consists of several parts. A COMPONENT INTERFACE is used to reduce dependencies and to make the implementation exchangeable. COMPONENT HOMES are used to managed the instances of a specific COMPONENT, they provide a way to create, find or destroy those instances. An COMPONENT IMPLEMENTATION is of course also necessary, it implements the functional concerns of the operations specified in the interface. And, last but not least, you need to define ANNOTATIONS for the COMPONENT. They specify how the COMPONENT should be integrated with the technical services the CONTAINER provides.

It is not always simple to actually find suitable parts of functionality that lend themselves to being encapsulated as one COMPONENT. Finding such parts is a task usually done during the analysis phase and requires some experience. Component architectures don't provide help here, however, they usually provide different types of COMPONENTS for different types of requirements. ENTITY COMPONENTS are used to represent business entities and usually contain persistent state. SERVICE COMPONENT are stateless and provide services to the application. SESSION COMPONENTS are stateful and are usually used to represent small processes or use cases. Each of these types has different characteristics and requires specific treatment by the CONTAINER, actually, different CONTAINERS are used for each kind of COMPONENT.

EJB, CCM and COM+ are all based on the concept of a COMPONENT. In all three cases, an application is built by assembling, or "wiring" COMPONENTS. All three technologies consider COMPONENTS the smallest technical building block of a component based system.

There is also another notion of "component" which is also called a Business COMPONENT [HS00]. It spans several layers, such as user/session/application or logic/persistence and can therefore consist of several COMPONENTS as defined here. Business COMPONENTS are not what we understand when when we talk about a COMPONENT.

EJB is a relatively new component model that has no "legacy history". It uses the Java technology and has found widespread use for enterprise applications. EJB is part of the Java 2 Enterprise Edition (J2EE) [SUNJ2EE], a complete suite of technologies to build server-side business applications.

The CORBA Component Model (CCM) has two ancestors: CORBA and EJB. It takes the EJB component model to the CORBA world, retaining almost all of the aspects of CORBA while still being compatible to EJB. Thus, CORBA clients and EJB clients can access CCM COMPONENTS. Of course, some features such as multiple interfaces are not accessible for them. At the time of this writing, there are only partial implementations of the CCM available.

COM+ is the successor to MTS and DCOM, which in turn is an extension of COM to allow remote access to COMPONENTS. COM itself – the Microsoft Component Object Model – has been around for a long time in several variations and with several names such as COM, OLE, or ActiveX. COM+ supports the concept of component aggregation. *The interfaces of aggregated COMPONENTS can be 'published' by aggregating COMPONENT, enhancing reuse while hiding the aggregation structure from clients.*

Component Interface

You decomposed the functional concerns of an application into COMPONENTS. Your system should be made up of loosely coupled, collaborating COMPONENTS.

A COMPONENT needs to collaborate with other COMPONENTS in order to be really useful. However, to facilitate reuse, you want to make sure that the COMPONENTS and its clients do not depend on the implementation of other COMPONENTS. Furthermore, you want to be able to evolve the implementation of a COMPONENT without touching other COMPONENTS or clients that use it. Last bit not least, you may want to provide multiple alternative implementations for a specific COMPONENT.

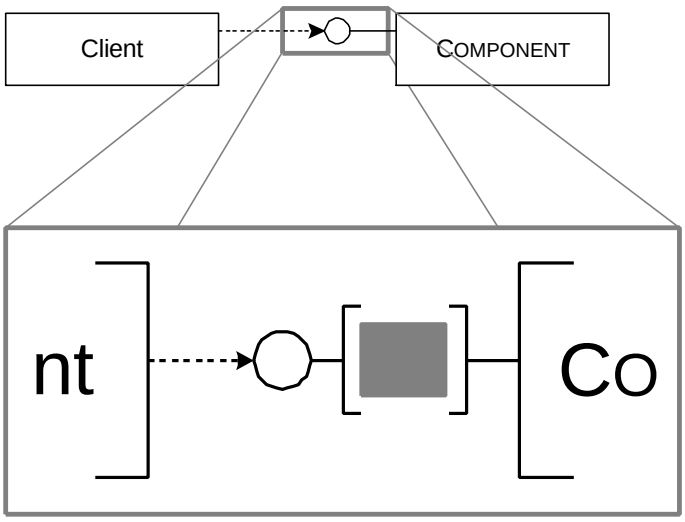
COMPONENTS are introduced to serve as functional building blocks from which you can assemble an application. One of the primary goals of decomposing application functionality into COMPONENTS is to provide a means to evolve each chunk of self-reliant functionality independent of of other parts of the application. If COMPONENTS would directly depend on each other, either from a compilation or a deployment point of view, the use of COMPONENTS as independent building blocks would be questionable, however.

Instead, what you need is a means to decouple the clients of a COMPONENT from this COMPONENT'S implementation: clients, whether actual clients or client COMPONENTS, should rely only on a specification of *what* the COMPONENT does, not *how* the COMPONENT does it. This independence of the *how* can even go as far as that the programming language or the underlying operating system should be transparent to the client.

If this can be achieved, you can reuse COMPONENTS as black box entities. When you build applications from COMPONENTS, you assemble them based on the functionality they provide. You reuse COMPONENTS as a building blocks, 'wiring' them together to form the complete system.

Therefore:

Provide a COMPONENT INTERFACE which declares all the operations provided by a COMPONENT, together with their signatures and, ideally, their semantics. Use this interfaces as a contract between the client and the COMPONENT. Let clients access a COMPONENT only through its interface.



The COMPONENT INTERFACE must be strongly separated from the COMPONENT IMPLEMENTATION. It must also be possible to use a different COMPONENT IMPLEMENTATION with the same COMPONENT INTERFACE, in order to be able to evolve the implementation. Of course, if the implementation changes in a way that requires a change to the interface, because operations are added or signatures have to be changed, the COMPONENT INTERFACE must be adapted accordingly.

In current practice, a COMPONENT INTERFACE only specifies the signature of the defined operations. The semantic meaning of an operation is usually defined only as a plain text specification. Note, that semantics is not the same as the implementation. You can easily specify the functional semantics of an operation without saying anything about its concrete technical implementation, for example using pre- and postconditions as proposed in Meyer's design-by-contract. As an interface does not formally define its semantics, a client cannot be sure that the COMPONENT actually does what the client expects, because

- the client's expectations may be wrong,
- the implementer got the requirements wrong and implemented the Component Interface in a wrong way.

To enhance the chances for interoperability, accessing this interface should be standardized. Ideally a binary standard should be provided. This allows clients in different programming languages or operating systems to access the COMPONENT INTERFACE. Details of remote access to the COMPONENT are handled by the COMPONENT BUS which hides the details of remote transport.

As the COMPONENT INTERFACE is separated from the COMPONENT IMPLEMENTATION, and because clients will only ever see and depend on the COMPONENT INTERFACE, the COMPONENT IMPLEMENTATION is exchangeable. To actually achieve this exchangeability in practice, make sure that the COMPONENT INTERFACE does not imply anything about the COMPONENT IMPLEMENTATION.

A COMPONENT can have multiple COMPONENT INTERFACES. If this is the case, you might want to provide a way for a client to query the COMPONENT about its COMPONENT INTERFACES and access them separately. This further reduces dependencies. Using version tags, this can help to evolve each interface separately from the others. Moreover, this approach allows a more role-oriented way of software development, where the interfaces define the different roles that can be 'played' by a COMPONENT. As these role-specific COMPONENT INTERFACES are a kind of *Adapter* [GHJV94]: they allow otherwise unrelated COMPONENTS to be used by the same client, because the role provides them the COMPONENT INTERFACE this client expects.

Providing COMPONENT INTERFACES separately from the COMPONENT IMPLEMENTATION has additional advantages, especially for the CONTAINER: it is the basis for many optimizations. The CONTAINER usually generates a COMPONENT PROXY which formally implements the COMPONENT INTERFACE and is used to attach the COMPONENT to the COMPONENT BUS. On the other side, it invokes LIFECYCLE CALLBACK operations, which are necessary to provide VIRTUAL INSTANCES, which is in turn the basis for PASSIVATION and POOLING. To actually realize the functional issues, the COMPONENT PROXY delegates invocations to the COMPONENT IMPLEMENTATION.

Some people advocate the use of completely generic COMPONENT INTERFACES. For example, it could consist of exactly one operation called *do(taskXML): resultXML* which only operates with XML data. It takes an XML string as parameter, which, conforming to a certain DTD, specifies the task to be executed. The result is again specified in XML. The advantage of this approach is, that a change in the COMPONENT IMPLEMENTATION never ever requires changes to the COMPONENT INTERFACE – you never need to recompile clients. However, the dependencies are still there, they are just not reflected in the COMPONENT INTERFACE, and thus not enforced by the compiler or the CONTAINER. For example, as you still have to change the COMPONENT IMPLEMENTATION to implement new requirements, you still need to reinstall the COMPONENT IMPLEMENTATION, and clients might need adaptation to provide newly required data in the XML. Another disadvantage of this approach is that is circumvents some of the mechanisms provided by the CONTAINER. As the operation to be executed is specified as part of the XML data, you cannot use ANNOTATIONS in the way they are intended. It is thus questionable whether this kind of COMPONENT INTERFACES is actually an advantage.

All example technologies use this pattern. Because EJB is limited to Java for COMPONENT *development, the* COMPONENT INTERFACE *is specified using Java's* interface *construct. There are just some additional rules and restrictions on the operation parameters used in* COMPONENT INTERFACES*. From version 2.0 onwards, EJB provides two kinds of interfaces: those used for regular, potentially remote access (this was already there in pre-2.0 EJB) and so-called local interfaces, that serve as an optimization for bean-to-bean communication in the same* APPLICATION SERVER*. The two interfaces are completely independent of each other, they can even have different operations. A Bean can have only remote, only local or also both kinds of interfaces. EJB Remote Interfaces must inherit from* javax.ejb.EJBObject*. Operations must throw the* java.rmi.RemoteException*. This is necessary to allow the framework to report* SYSTEM ERRORS *such as failed network communication or resource problems. Local interfaces must extends* javax.ejb.EJBLocalObject*, and the operations must not throw* RemoteExceptions *– the subset of* SYSTEM ERRORS *that can still occur in non-distributed settings are signaled using an* EJBException*, a subclass of* RuntimeException*.*
An EJB COMPONENT*, a bean, can formally only implement one local and one remote interface. These interfaces can of course inherit from several base interfaces. Client references to* COMPONENT *instances are declared using this interface or any of its base interfaces as their static type. Client code only depends on the interface, just as it is described in this pattern.*

In CCM, the programmer starts by defining one or more interfaces in IDL, CORBA's Interface Definition Language. These interfaces are not yet related to any COMPONENT*. Each could be implemented by an ordinary CORBA object. In a second step, a* COMPONENT *is defined that* supports *or* provides *one or more of the interfaces. Supported interfaces are implemented by the* COMPONENT *directly. Provided interfaces are implemented as a separate facet: a client has to request this interface from the* COMPONENT *at runtime explicitly. The definition of* COMPONENTS *as a collection of interfaces is done using Component IDL (CIDL), an extension of IDL which allows a more succinct notation of* COMPONENTS*. Because IDL and CIDL is programming language independent and many programming language mappings exist, it is possible to implement CCM* COMPONENTS *with several different programming languages. Among others these are C++, Java, Ada, Cobol, Smalltalk, and Perl. Of course this is only possible if a* CONTAINER *for the language is available, which is not the case at the time of this writing. Each interface is given a unique repository ID to identify interfaces in CORBA interface repository. For the ID, several different formats exist. It is up to the developer to ensure its uniqueness over time and space.*

Container

According to the principle of SEPARATION OF CONCERNS, you decomposed your functional requirements into COMPONENTS.

Your COMPONENTS address functional concerns only, that is, they contain pure business logic that does not care about technical concerns. However, to actually execute these COMPONENTS as part of a concrete system, you also need something that provides the technical concerns and integrates the components with their environment. Moreover, you want to be able to reuse the technical concerns effectively.

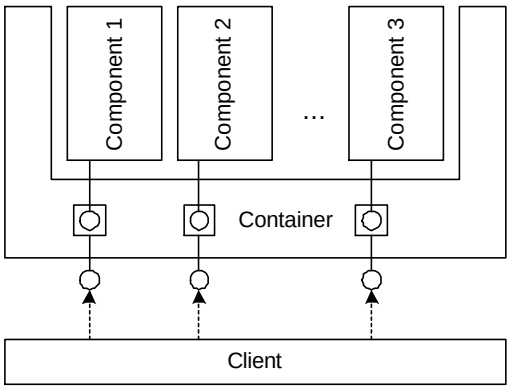
There are two kinds of technical concerns: those that clients require, such as security, transactions, and those that the COMPONENT itself requires, for example concurrency, persistence, and access to infrastructure resources. COMPONENTS not only need to benefit from these technical concerns without being tied to a particular infrastructure or platform, such a transaction monitor or a security infrastructure. Moreover, these services must be provided non-intrusively: they have to be added 'from the outside.' From the client perspective, the component should appear to be a single entity

providing their functionality integrated with the required technical concerns. From the component's perspective, the technical concerns should be provided 'magically' – the component code must not be polluted by code covering technical concerns.

As many COMPONENTS and COMPONENT-based applications depend on the same technical concerns, their implementations should be reusable. In other words, you do not want to reinvent and reimplement the technical concerns for each application over and over again. The goal is to standardize and implement these services generically, for example using frameworks or code generation techniques.

Therefore:

Provide an execution environment which is responsible for adding the technical concerns to the COMPONENTS. This environment is generally called CONTAINER. Conceptually, it wraps the COMPONENTS, thus giving clients the illusion as if functional and technical concerns were tightly integrated. To implement these technical concerns in a reusable manner, a CONTAINER uses frameworks and other generic techniques such as code generation.[2]



To integrate the COMPONENTS with the CONTAINER while not polluting their code with technical concerns, use ANNOTATIONS. ANNOTATIONS specify the technical requirements for a particular COMPONENT separately from their COMPONENT IMPLEMENTATION. To allow the CONTAINER to apply these ANNOTATIONS on the COMPONENTS requires an explicit step, the COMPONENT INSTALLATION. This prepares the CONTAINER to host and execute the new COMPONENT as specified. Usually, this involves the creation of a GLUE-CODE LAYER, which adapts the generic parts of the CONTAINER to a specific COMPONENT.

There are different ways how the combination of the generic parts of a CONTAINER and a GLUE-CODE LAYER share their work. Either the CONTAINER can already provide most functionality generically, and the GLUE-CODE LAYER merely plays the role of an adapter. Or, the CONTAINER is basically just a collection of hooks calling back into the GLUE-CODE LAYER:, which then does most of the work. The extreme case is that the CONTAINER is generated altogether.

To optimize resource consumption and performance, a CONTAINER usually provides VIRTUAL INSTANCES. This decouples the lifecycle of a logical entity visible to client applications or other COMPONENTS from its physical representation in the CONTAINER.

---

2 Note that it is not the goal of this book to provide details of CONTAINER implementation. We focus on the interfaces needed by COMPONENTS and the CONTAINER to facilitate their cooperation.

As the different COMPONENT types – SERVICE, SESSION and ENTITY – have different characteristics, you will provide different CONTAINERS. They are integrated in an APPLICATION SERVER which provides all the services common to the different CONTAINER types.

The management of resources used by a COMPONENT such as a database connection is also a technical concern. Therefore, the CONTAINER must MANAGE RESOURCES on the behalf of COMPONENTS it hosts. The CONTAINER cooperates with the APPLICATION SERVER in these respects.

To realize high-availability requirement or to provide load-balancing, CONTAINERS can be federated, or clustered. A set of physically separated CONTAINER, which usually run on different machines, is logically joined to act as one CONTAINER. This is transparent for the client, and for the COMPONENT developer.

EJB requires a special piece of software that plays the role of the CONTAINER. Usually, the CONTAINER is embedded in a larger application, a J2EE APPLICATION SERVER. A J2EE conforming APPLICATION SERVER has to provide additional services, such as NAMING accessible through the JNDI API, transactions or Servlets. From EJB 2.0 onwards, messaging is integrated with EJBs in the form of Message Driven Beans: beans that act as "message receivers".
Usually, J2EE APPLICATION SERVERS are implemented in Java and provide separate CONTAINERS for each component type. EJB CONTAINERS have to conform to the EJB specification to allow seamless exchange of COMPONENTS between different CONTAINER vendors. Nevertheless, they are free to offer different levels of quality of service. For example, a CONTAINER might offer load balancing, high availability, or advanced caching strategies while other CONTAINERS don't do that.

CCM also uses the concept of a CONTAINER, although no complete implementations are on the market at the time of this writing. CCM explicitly defines different CONTAINER types for the different types of COMPONENTS, thus there are entity, service, process and session CONTAINERS. To implement these different types of CONTAINERS, the facilities already provided by the Portable Object Adapter[3] (POA) are used. This means, that a particular CONTAINER is based on settings for threading, lifespan, activation, servant retention and transaction policies. Because all this is already part of "ordinary" CORBA, it is possible to access a COMPONENT from a client that does not know, that the CORBA object to which it has a reference is actually a COMPONENT. Some features such as multiple interfaces are of course not accessible for ordinary CORBA clients. In addition to the technical concerns mentioned above, CCM CONTAINERS also handle persistence in a well-defined way. The persistent state of a COMPONENT can be specified abstractly, and a mapping to logical storage devices can be provided. These logical storage devices are then mapped to real databases with CONTAINER-provided tools.

In COM+, the role of the CONTAINER is played by parts of the Windows 2000 operating system. In the pre-COM+ days, the Microsoft Transaction Server (MTS) played the role the CONTAINER and was not part of the operating system. Every COM+ application, which consists of several COMPONENTS in their own DLLs, runs in a surrogate process controlled by COM+. It handles threading, remoting, synchronization, and pooling. COM+ provides only one type of COMPONENTS (stateless), and persistence is not explicitly addressed by COM+.

3 The Portable Object Adapter is a framework for managing CORBA server objects inside an application. In contrast to the Basic Object Adapter , the BOA, it is more flexible (for example regarding activation and eviction policies) and it is better standardized and therefore more portable.

Annotations

You use COMPONENTS to implement the functional concerns and a CONTAINER to take care of the technical concerns.

Due to the principle of SEPARATION OF CONCERNS, the CONTAINER is responsible for providing the technical concerns. In order to be applicable for all COMPONENTS, the CONTAINER implements the technical concerns in a flexible, generic way. They have to be configured to a certain extent to make sure they fit for a particular Component. However, we don't want to code this information explicitly for each COMPONENT as part of its COMPONENT IMPLEMENTATION.

Generality and reusability always means that there is something to configure, or specify when a generic artifact should be used. This is also true for the CONTAINER. Consider transactions: it is not enough that the CONTAINER *can* handle transactions for you – you have to tell it *how* it should handle transactions for a specific COMPONENT. There are many ways how transactions can be handled:

- an operation can run in a transactional context
- an operation might be required to run within a transaction
- an operation might be required to run in its own, new transaction
- an operation might not be allowed to run in a transaction at all.

The CONTAINER can provide all these alternatives to you. But you have to help it decide which alternative to use.

Another example is security. The CONTAINER can enforce permission checks on COMPONENT instances. But you need to tell the container, what these permissions are. Who is allowed to create a COMPONENT instance? Who is allowed to invoke which operation on which instance?

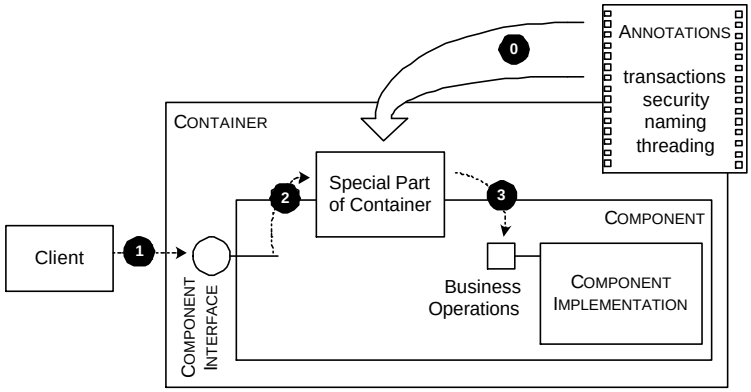
The technical concerns realized by the CONTAINER can usually be characterized by the following properties:

There is only a limited number of possible options for each concern (for example, a transaction can be required, allowed, or forbidden for an operation) or the options can be defined with regards to a set of data items and options (such as: user XY is allowed to invoke the operation abc()).

Once the above decision is made, the implementation of the concern follows simple, predefined rules or structures.

Therefore:

Provide a means for the COMPONENT developer to annotate the COMPONENTS with the technical concerns. Such ANNOTATIONS are a part of the COMPONENT, but separate from the COMPONENT IMPLEMENTATION and the COMPONENT INTERFACE. These ANNOTATIONS specify how the CONTAINER should apply the technical concerns for the COMPONENT. ANNOTATIONS are not programs, they are a high-level specification. It is the CONTAINER'S job to decide about the concrete implementation.



0: ANNOTATIONS are used to "configure" the CONTAINER
1: Client invokes operation on the COMPONENT INTERFACE
2: Invocation is intercepted by the CONTAINER
3: Invocation is forwarded to actual COMPONENT IMPLEMENTATION

ANNOTATIONS do not relieve the COMPONENT developer from thinking about technical concerns. He still has to make reasonable decisions about what the CONTAINER should do with his COMPONENT. But the programmer is not required to *code* these things. As described in SEPARATION OF CONCERNS this significant advantages regarding people, responsibilities, performance optimizations, reuse, and more.

It is important to understand that the CONTAINER can't do wizardry and make clever decisions. Wrong specifications in the ANNOTATIONS can have consequences that would be considered a serious bug in the overall application. For example, transaction attributes can be very important regarding the correctness of an application.

Due to these characteristics of the technical concerns mentioned above, it is usually possible to use a GUI-based tool to help programmers to specify the ANNOTATIONS, providing the predefined options for selection. To facilitate the creation of such a tool, and to allow the CONTAINER to verify the specification the ANNOTATIONS, COMPONENT INTROSPECTION can help. COMPONENT INTROSPECTION allows external entities such as tools or the CONTAINER to access information about a COMPONENT, for example a list of operations and their parameters.

The CONTAINER can implement ANNOTATIONS in different ways. To ensure reasonable performance, and to adapt a specific COMPONENTS to the generic parts of the CONTAINER, the CONTAINER will generate a GLUE CODE LAYER. This is usually during COMPONENT INSTALLATION. Because the programmer only specifies what the CONTAINER should do, but not how, this leaves a lot of freedom for optimizations on the part of the container.

Apart from the obvious examples transactions and security, there are other things that are often part of ANNOTATIONS:

- Name of the Component in the Naming system
- The names and types of Managed Resources which are accessed by the Component
- Dependencies on other Component Interfaces
- A particular threading model
- Information for optimizations which cannot be implemented by the Container without additional, perhaps domain-specific knowledge
- Quality of service parameters, that might for example configure the Component Bus.

in the COM-world, ANNOTATIONS are also known under the name *Declarative Programming* and *Attribute-Based Programming* (.

In EJB, the Deployment Descriptor is an instance of this pattern. It contains security policies, specification of transactional properties, and other general information, such as whether a session bean is stateful or stateless, the name of the bean in JNDI, et cetera. For Entity Beans it also contains (partially vendor-specific) information for container-managed persistence, if used, and about relationships among Entity Beans.

In EJB 1.0 the deployment descriptor used to be a serialized Java object. This lead to many problems regarding portability and handling. This is why in EJB 1.1 and subsequent versions the deployment descriptor is an XML file.

The deployment descriptor must be supplied together with the COMPONENT IMPLEMENTATION and the COMPONENT INTERFACE for deployment in a J2EE APPLICATION SERVER. The deployment descriptor can be created by hand using a normal text editor, although almost all EJB server vendors provide GUI tools to facilitate the process of setting up the deployment descriptors. These tools use COMPONENT INTROSPECTION to gain information about the COMPONENT. Whether it is more appropriate to use a tool or edit the deployment descriptors manually, depends very much on the size of the project and the development process/tools used by the development team.

A CORBA component descriptor serves the purpose of ANNOTATIONS in CCM. It is XML-based like the deployment descriptor in EJB. A CORBA component descriptor provides among other things information on a COMPONENT and all its supported and provided interfaces as well as threading, transaction and security policies. The provided, supported, and REQUIRED INTERFACES are referenced using their interface repository id, allowing additional details about the interfaces to be retrieved from the repository.

COM+ also uses ANNOTATIONS for several aspects of a COMPONENT'S behavior. In this context, the concept is known as attribute based programming. These attributes are stored in a special configuration database called the COM+ catalog. They can be specified when the COMPONENT is installed using a tool called the Component Services Explorer. The aspects of a COMPONENT that can be configured by attributes are among others transactions, synchronization, pooling, the construction string (initialization data), just-in-time activation, and security.

Component Bus

You use a CONTAINER to host your COMPONENTS. To implement a N-TIER ARCHITECTURE, remote access to these COMPONENTS is necessary.

In an N-TIER ARCHITECTURE your COMPONENTS can reside on different machines than the client applications or within the same process (CONTAINER or APPLICATION SERVER). Client programming should be the same no matter if the accessed COMPONENTS are located locally or remote. Moreover, it is critical that remote access performs well in the face of many COMPONENT instances and clients – quality-of-service properties must be realized.

Providing remote access to objects is a problem that has essentially been solved by object oriented middleware such as CORBA, DCOM or Java's RMI. Patterns like *Broker* [POSA] or remote *Proxy* [GHJV94] described extensively how such systems are implemented. However, this is not enough for component architectures.

First of all, clients or client COMPONENTS should not need to be changed when the underlying transport protocol changes, for example from RMI's JRMP to IIOP or DCE/RPC. A COMPONENT or

client developer should not have to deal with any protocol specific aspects. Everything should be handled from the CONTAINER as part of the technical concerns.

Second, clients should not need to care about the actual location of a specific COMPONENT instance because this is a technical concern. This is even more true in the face of relocation of COMPONENTS for reasons of failover or load-balancing.

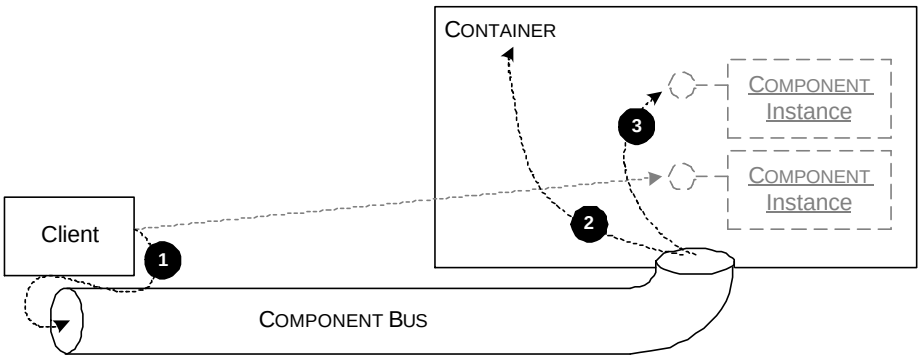
Third, the remote access technology must be integrated with the implementation of VIRTUAL INSTANCES. Depending on how they are implemented, there might be varying numbers of physical COMPONENT instances in the CONTAINER. Remote access must still perform well for very large numbers of physical (or even logical) instances and many clients. For example, it usually not feasable to use a separate network connection or socket for each physical instance.

In the case that the caller and the invoked instance are located in the same process, the invocation should be optimized. The overhead compared to a normal method invocation should be as low as possible. This scenario is quite typical. It happens whenver a COMPONENT invokes an operation on another COMPONENT which is located in the same CONTAINER or APPLICATION SERVER.

There should also be a way to change the semantics of the transport protocol, ideally without effect on client programs and COMPONENT IMPLEMENTATIONS. It should be possible to transport COMPONENT invocations synchronously or asynchronously. Quality-of-service parameters such as guaranteed delivery for asynchronous invocations or request timeouts should also be configurable, the COMPONENT developer and the client programmer should not need to care.

Therefore:

Access COMPONENTS only via a COMPONENT BUS, a generic communication infrastructure for remote and local COMPONENT access. Because the COMPONENT BUS is integrated with the CONTAINER, it can provide efficient access to a large number COMPONENT instances. Because it knows the location of the caller and the invocation target, it can efficiently optimize local invocations. The COMPONENT BUS hides the underlying low-level transport protocol, its semantics and its quality-of-service attributes. Its configuration is done with the help of ANNOTATIONS.



1: Client invokes an operation on the remote VIRTUAL INSTANCE
2: COMPONENT BUS accesses the CONTAINER to manage the technical concerns
3: Invocation is forwarded to the VIRTUAL INSTANCE

A generic COMPONENT BUS into which clients and COMPONENTS can be plugged-in allows you to relocate COMPONENTS and their clients in anyway you like. This builds the basis for load-balancing and failover.

Because of its generic nature, the communication protocol used by the COMPONENT BUS can be switched according to the technical requirements, such as quality of service, performance, standard conformance or the underlying physical transport layer. Usually, the standard distributed object communication technologies (such as CORBA, RMI), or messaging middleware is used.

To control the technical aspects of the underlying transport protocol, a specific form of ANNOTATIONS can be used. Here, the QoS attributes, the selected protocol, or some performance optimizations can be specified.

While the primary task of the COMPONENT BUS is of course to transport method invocation requests and their results, this is not enough. To allow the CONTAINER to take care of the technical requirements, it needs to know things like the current transaction or the security context. An INVOCATION CONTEXT is used for that purpose. The CLIENT LIBRARY is used to fill in this information on the client side.

As mentioned, the COMPONENT BUS is also responsible for optimizing resource usage, especially for sockets or other network I/O. In most cases, one remote connection is used to communicate with many, sometimes even all instances of a specific COMPONENT. To still be able to deliver the invocations to the correct instance, the COMPONENT BUS needs to cooperate with the COMPONENT PROXY. At the minimum, the COMPONENT BUS has to transport the id of the logical invocation target. On the server side, the COMPONENT BUS usually collaborates with the COMPONENT PROXY in order to access VIRTUAL INSTANCES.

It is not always possible to hide of the underlying protocol completely. This is especially true in the case of asynchronous communication based on message-oriented middleware. For example, to exploit the benefits of asynchronous communication, the client programming model has to be aware of the asynchronicity. The same is true inside the COMPONENT IMPLEMENTATIONS: Because a MoM does not invoke business methods on the target (instead it delivers messages), a method to handle incoming messages must be provided. This might even have consequences for the COMPONENT INTERFACE.

Note that in the case of local invocations, it is not possible to use ordinary method invocations: the INVOCATION CONTEXT has to be propagated, the technical concerns have to be enforced by the CONTAINER and the INVOCATION CONTEXT has to be propagated.

In EJB, the COMPONENT BUS is implemented using Java's Remote Method Invocation (RMI) feature. RMI itself can work with different lower-level communication protocols, for example JRMP, which is RMI's native transport protocol, or IIOP, CORBA's TCP/IP-based transport protocol. Up to EJB 1.1, the COMPONENT BUS only supports synchronous invocations. In version 2.0, asynchronous invocations based on the Java Messaging Service (JMS) are available. However, this is not transparent to the bean developer, because a specific bean type has to be used, namely message driven beans. Message driven beans (MDBs) basically stateless session beans with a specific, predefined interface that can handle incoming messages. An MDB is attached to a specific queue or topic. To access an MDB, a client has to post a message to the respective queue or topic. Thus, asynchronous delivery is also not transparent to the client.

EJB's COMPONENT BUS does not support additional quality of service properties. Depending on the underlying protocol, the INVOCATION CONTEXT can be transported directly (as in IIOP) or it has to be transported manually (as in JRMP).

EJB 2.0 introduced a feature called local interfaces. Here, the developer specifies a separate COMPONENT INTERFACE and COMPONENT HOME for use in local invocations. Because local interfaces can only be invoked locally, the CONTAINER can significantly optimize these invocations. Insofar this is a "manual" optimization of the COMPONENT BUS.

CCM, being CORBA-based, uses CORBA as its COMPONENT BUS. *CORBA itself supports pluggable protocols – they can be native to the ORB or they can be one of the standard communication protocols (GIOP, IIOP). Asynchronous delivery can be achieved in two ways: Either by declaring an operation* oneway *(which basically uses a separate thread for method invocations), or by using more sophisticated methods based on callback interfaces or polling objects. Note that the latter two possibilities are not transparent neither for the* COMPONENT *developer, nor for the clients.* COMPONENT IMPLEMENTATION *is simplified, because a lot of the necessary code for the* COMPONENT BUS *is generated during* COMPONENT INSTALLATION.

COM+ uses the same communication protocol as DCOM, namely Microsoft's version of DCE RPC. It is possible to use asynchronous communication based on the Microsoft Message Queue (MSMQ). In this case, the CLIENT-SIDE PROXY *records the method invocation and posts a corresponding message to a predefined message queue. On the server side, the* COMPONENT PROXY *listens to the queue, decodes the message and invokes the operation on the* COMPONENT. *Except for some limitations to the method signature (no return values, no parameters passed by reference), asynchronicity is just a matter of configuration during* COMPONENT INSTALLATION. *Semantically, this mechanism is similar to CORBA's oneway operations.*

Client-Side Proxy

Using a COMPONENT BUS, you remotely access your COMPONENTS in the CONTAINER.

The client programming model should be as simple as possible. In particular, the remoteness of the COMPONENTS and access to the COMPONENT BUS should happen automatically. Moreover, the necessary data for the INVOCATION CONTEXT need to be filled in somehow. This might even include the remote instance's identity, depending on the way how the CONTAINER implements VIRTUAL INSTANCES.

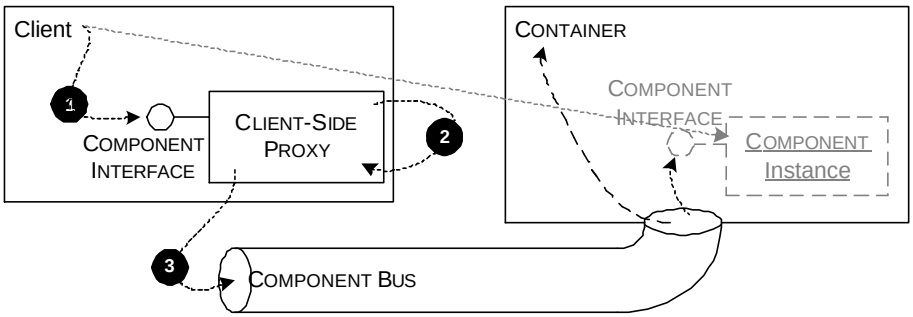
Client programming should be as intuitive as possible. So, as in every distributed, object-oriented environment, client method invocations should happen in the same way as local operations. The conversion of the invocations into remote method invocations and the attachment to the COMPONENT BUS should happen automatically.

However, in a COMPONENT environment things are a little bit more complicated than in a purely distributed object situation. This is because things like the transaction context or the client's security data has to be transported with every method invocation. This is done using an INVOCATION CONTEXT, but of course, the necessary data has to be provided to the COMPONENT BUS with every invocation to allow the COMPONENT BUS to fill the INVOCATION CONTEXT.

There is another problem related to the implementation of VIRTUAL INSTANCES and the COMPONENT BUS. Depending on the way how VIRTUAL INSTANCES are implemented in the CONTAINER, and how the COMPONENT BUS manages remote connections, the INVOCATION CONTEXT might also need to include an identifier of the VIRTUAL INSTANCE for which the invocation is intended.

Therefore:

Provide a CLIENT-SIDE PROXY for the remote COMPONENTS. It implements the respective COMPONENT INTERFACE and accepts local method invocations. It forwards the invocation data to the COMPONENT BUS and also supplies the necessary data for the INVOCATION CONTEXT.



1: Client invokes an operation on a VIRTUAL INSTANCE, in reality it reaches the CLIENT-SIDE PROXY
2: Proxy adds the data for the INVOCATION CONTEXT
3: Proxy hands over the invocation to the COMPONENT BUS

The CLIENT-SIDE PROXY is typically a generated artifact. It has to implement the COMPONENT INTERFACE of the remote COMPONENT which it represents, but its implementation is highly stereotypical, thus it can be generated easily. The generation is usually done during COMPONENT INSTALLATION.

The generated proxy is then packaged into the CLIENT LIBRARY. This complete library contains additional artifacts necessary for the client and it is created during COMPONENT INSTALLATION. In some cases, the proxies can also be automatically downloaded from the server.

The way how the CLIENT-SIDE PROXY actually works internally depends very much on the way how the COMPONENT BUS and the CONTAINER work. They all have one thing in common: for each remote reference to a COMPONENT instance, there is one local proxy object. The proxy objects are usually very small and they basically just forward the invocation to the COMPONENT BUS. Doing this, they supply the COMPONENT type, the instance id and the information about the invoked operation as well as the data for the INVOCATION CONTEXT. In most cases, all the CLIENT-SIDE PROXIES use one remote connection provided by the COMPONENT BUS, otherwise there would potentially be many remote connections – a potential performance bottleneck.

Instances of the proxy are created when the COMPONENT BUS returns references to COMPONENT instances as a consequence of calling a method on a COMPONENT HOME.

It depends on the component architecture and, in particular, the COMPONENT BUS whether COMPONENT-specific code is required for the CLIENT-SIDE PROXY, or whether it can be implemented generically with just some definitions and parameters. If the programming language supports reflection, a generic implementation is possible, however not always desirable from a performance point of view. Typically, the bare minimum usually consists of header files or other programming language artifacts which define the COMPONENT INTERFACE of the COMPONENT that should be accessed.

Note that this pattern is also used in many distributed object systems and it has been described several times in [GHJV94] (the *Remote Proxy*) and in [POSA]. We include it here for completeness and because in because here it has the additional responsibility of providing the INVOCATION CONTEXT data to the COMPONENT BUS.

In EJB, the CLIENT-SIDE PROXIES are called stubs. Usually they are part of the CLIENT LIBRARY and have a name like _MyComponentStub. *Instead of providing them as part of the CLIENT LIBRARY, they can also be downloaded from the server if the underlying RMI subsystem is configured appropriately. The internal*

implementation really varies greatly. In many cases, the mechanisms of the Java reflection API are used in some way or another.

In CCM, references to remote COMPONENT instances are basically remote object references. Remote objects are accessed in CORBA using generated stubs – an implementation of this pattern.

In COM+, the same mechanism is used. The proxy is part of the marshalling-DLL.

Client Library

Using a COMPONENT BUS and a CLIENT-SIDE PROXY you remotely access your COMPONENTS in the CONTAINER.

Clients use a CLIENT-SIDE PROXY to access remote COMPONENTS as if they were local objects. But the proxy class has to be made available to clients. Clients also need access to several other helper classes as well as to the types of the parameters of operations and their return values. Configuration data must be made available to clients in order to allow them to bootstrap and contact the APPLICATION SERVER. Last but not least, the code to attach clients to the COMPONENT BUS itself must be available to client programs.

The CLIENT-SIDE PROXY is usually a programming language class. This class must be available to the client programs in order to allow them to use it in their process and access the instances locally.

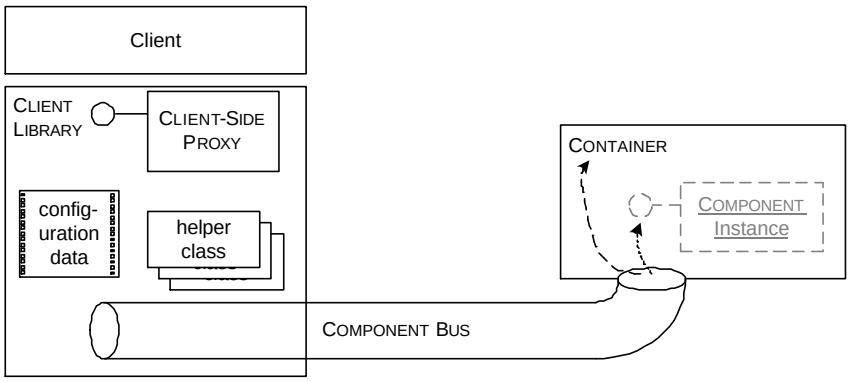
If a COMPONENT uses certain types of parameters that are not part of the programming language specific standard libraries, then these types must also be supplied to the client, otherwise it cannot be compiled and linked successfully.

There is always a certain amount of configuration data necessary to successfully run a client application. For example, the network address of the APPLICATION SERVER has to be defined at the client side.

Last but not least, there must be code available on the client site which attaches the CLIENT-SIDE PROXIES to the COMPONENT BUS and which implements the COMPONENT BUS itself. This implementation must be compatible with the implementation of the CONTAINER itself because it cooperates with the CONTAINER to ensure the correct semantics of VIRTUAL INSTANCES.

Therefore:

Create a CLIENT LIBRARY upon COMPONENT INSTALLATION on the server. Most importantly, it contains the CLIENT-SIDE PROXY which is the placeholder of the remote COMPONENTS in the client processes. In addition, it contains any other parametrization or initialization parameters that are necessary for the client to access CONTAINER and APPLICATION SERVER remotely, as well as a suitable implementation of the COMPONENT BUS.



Using this pattern allows the client application developer to "transparently" access the COMPONENTS, integrated with the technical concerns handled by the CONTAINER. Note that the CLIENT LIBRARY is under control of the CONTAINER. It is up to the CONTAINER to provide all the information the client requires, and to generate the CLIENT-SIDE PROXY in a way that suits its own needs.

Not everything is hard-wired in the CLIENT LIBRARY. Usually, there are some initialization parameters that can be set when the application is started to make sure the CLIENT LIBRARY does not need to be recreated all the time. For example, the network address of the APPLICATION SERVER is usually specified when the application starts.

If the client programming environment supports access to the COMPONENT BUS directly, then the CLIENT LIBRARY does not need to contain specific code to access it. However, usually the COMPONENT BUS has some CONTAINER-specific characteristics to support its implementation of VIRTUAL INSTANCES, so some COMPONENT BUS code is almost always required.

In EJB, most deployment tools automatically create a client JAR file during COMPONENT INSTALLATION. It needs to be installed in the clients classpath and is used by the client to access the COMPONENT on the server. The infrastructure aspects, such as security classes, and the COMPONENT BUS access itself is provided in the form of a JAR file that's generic for the CONTAINER and must be made available separately to the clients. Optionally some of these classes can also be downloaded from the APPLICATION SERVER on-the-fly.

In CCM, because of the lack of implementations, there are no concrete examples, but it is expected to work basically the same way as in the case of EJB. CORBA, the underlying transport layer, also uses generated classes that need to be made available to the client, so in CCM it will be the same.

In the case of COM+, a DLL is generated, called the marshaling DLL, that contains proxies and stubs to allow the client to contact the server object. It is dynamically linked to the client application.

Component Home

You decomposed the functionality into COMPONENTS. Your application is assembled from collaborating, loosely coupled COMPONENTS.

The procedure of how to create or find COMPONENT instances in the CONTAINER is not trivial. It depends largely on the technical concerns, and their implementation in the CONTAINER. But clients still need to create or find instances, independent of the concrete way how this is done.

A client who wants to invoke operations on a specific COMPONENT instance must first obtain a reference to this instance. A new instance can be created, or an older one needs to be found. It depends on the type of the COMPONENT what it actually means to create a new instance, or find an old one:

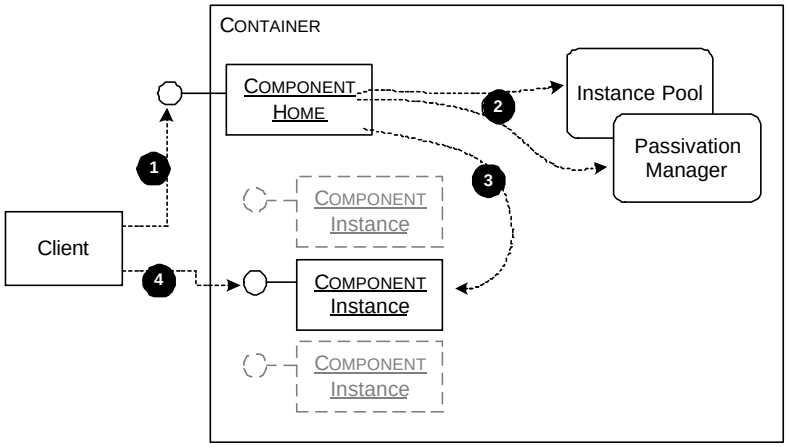
- For SERVICE COMPONENTS, you just need to get in contact with *any* instance This is because all SERVICE COMPONENTS are stateless and thus equal. Therefore the clients just gets one SERVICE COMPONENT from the pool. Refinding an instance is not necessary.
- For SESSION COMPONENTS, creating a new instance really means creating a new physical instance. Refinding an instance is not necessary, because SESSION COMPONENTS have no identity and are not shared between clients.
- For ENTITY COMPONENTS, the situation is again different. Creating an instance really means that new persistent data is created. As more than one client in more than one session need to access the persistent data, refinding entities is necessary. Refinding an instance is possibly based on the PRIMARY KEY of an instance.

As can be seen from the list above, creating or finding an instance possibly involves quite some logic that depends on the technical concerns and should thus be handled by the CONTAINER. The use of VIRTUAL INSTANCES make the situation even more complicated.

The situation is even more interesting in case of clustered APPLICATION SERVERS: you don't even know the concrete CONTAINER where the instance should be created.

Therefore:

For each COMPONENT, provide a COMPONENT HOME, which serves as an interface for clients to manage instances of the respective COMPONENT. Depending on the type of COMPONENT, the COMPONENT HOME contains different operations to create and find instances.



Before: Client looks up the COMPONENT HOME in NAMING
1: Client accesses COMPONENT HOME requesting a (new) COMPONENT instance
2: Depending on the COMPONENT kind, the COMPONENT HOME accesses the instance pool or the passivation manager
3: The COMPONENT HOME provides an instance to the client
4: The client accesses the instance

Note that the COMPONENT HOME is just an interface. This means, that the CONTAINER is free to implement it in a way that suits its particular internal structure. The steps required to implement the interface can be arbitrarily complex. This is usually done during COMPONENT INSTALLATION.

The COMPONENT HOME is an implementation of the *Factory* and *Finder* patterns [GHJV94]. As such, it is the clients' one and only access point to control the lifecycle of logical COMPONENT instances. Note

that, because of VIRTUAL INSTANCES, this has nothing to do with the lifecycle management provided by the CONTAINER for the physical instances coordinated using the LIFECYCLE CALLBACK interface.

To make the programming model simpler, the COMPONENT HOME should have the same "look and feel" for the programmer as other COMPONENTS.

The operations provided in the COMPONENT HOME vary, depending on the type of COMPONENT. For all types of COMPONENTS, an operation to create new logical instances has to be provided. What such a creation actually means depends on the way the VIRTUAL INSTANCES are implemented. In addition, ENTITY COMPONENTS need to be re-found (based on the PRIMARY KEY and probably other attributes), because they have a logical identity and persistent state. The COMPONENT HOME therefore needs to provide finder operations.

These finder operations often cannot be implemented completely by the CONTAINER. Consider a finder that allows you to find all *Person* entities with a first name "Joe". The business logic of how these entities are found usually has to be implemented by the COMPONENT programmer (for example by an SQL select statement). The implementation can be provided as part of the COMPONENT IMPLEMENTATION or as a separate artifact. Actually creating instances from these entities is the job of the CONTAINER and is part of the GLUE-CODE LAYER.

The COMPONENT HOME provides a well-defined way to get in touch with COMPONENT instances. Of course, now you need a way get in touch with the COMPONENT HOME. To solve this problem, NAMING contains a COMPONENT HOME instance for each COMPONENT.

When creating a COMPONENT (bean) in EJB, the programmer has to define a home interface (COMPONENT HOME) in addition to the bean's remote interface (COMPONENT INTERFACE). Depending on the COMPONENT type, the developer has to specify several create(), remove(), *and* find...() *operations. For EJB 2.0 and later, the home interface of entity beans can also be used to define business operations. These operations will be called on beans in the pooled state and serve as a way to implement operations that are not bound to a bean instance. They might be seen as the equivalent to static methods in Java.*

The create and find operations of the home interface can be overloaded with different signatures and serve as factory operations for the bean: they return one (create operations, find operations) or more (find operations) instances. Note that the programmer never really implements the home interface. If bean managed persistence is used, then the developer has to provide implementations for the home's operations in the COMPONENT IMPLEMENTATION class. If container managed persistence is used, he does not need to implement the operations at all – they are automatically implemented by the CONTAINER or a special IMPLEMENTATION PLUG-IN. Some additional specifications (such as an SQL string for the finders) usually has to be specified in the deployment descriptor. In both cases, the actual implementation of the home interface itself is generated ba the CONTAINER as part of the GLUE-CODE LAYER, to allow for INSTANCE POOLING and PASSIVATION. EJB 2.0 introduced an optimization for local invocations (from a collocated bean). These beans can use the local home and the local interface. Conceptually, they are the same as their remote counterparts (home interface and remote interface) but they can be used locally only.

In CCM, you also have to declare home interfaces for COMPONENTS. As in EJB, a home manages exactly one COMPONENT type. Home interfaces are by definition normal IDL interfaces, however, a CIDL shorthand exists. A home can work with PRIMARY KEYS or not, depending on the type of COMPONENT it manages.

Homes provide at least one create operation. Homes for COMPONENTS with primary keys also have a find and a destroy operation. The find operations are based on the PRIMARY KEY. In addition to the default create and find operations, the programmer can declare additional factory and finder operations. Only if such operations are

declared the home interface has to be explicitly implemented in a home executor. Otherwise, the home implementation is automatically generated.

In COM+ a COMPONENT (a CoClass *in COM+ terminology) is the implementation of one or more interfaces. For each COMPONENT (not interface!) the programmer has to create a factory. Programmatically, the factory itself is a COMPONENT, and its interface must inherit from* IClassFactory. *It provides an operation* CreateInstance *to create a new, non-initialized instance of the COMPONENT. The client programmer uses this operation to create a new COMPONENT instance. Upon creation, the programmer can specify which of the COMPONENT INTERFACES implemented by the COMPONENT implements he would like to receive.*

A short story, again

In the beginning I told the short story of Eric and John. Why not read it again, now you know what they're talking about?

Acknowledgements

First of all, I have to thank Sabena Belgian Airlines for showing so bad films on the flight back home from OOPSLA that I had to start a pattern language instead.

I have received a lot of valuable and insightful comments and reviews from many people, all of them have contributed significantly to this pattern language as it is now. The following is a list in alphabetical order: Jim Coplien, Costin Cozianu, Kristijan Cvetkovic, Matthias Hessler, Michael Kircher, Francis Pouatcha, Stephan Preis, Alexander Schmid, Michael Schneider, Oliver Vogel and Eberhard Wolff. Thanks to all of you.

Literature and Online Resources

[AG00] Alan Gordon, *The COM and COM+ Programming Primer*; Prentice-Hall, 2000

[CETUS] Schneider, et. al., *Cetus-Links*, http://www.cetus-links.org

[GHJV94] Gamma, Helm, Johnson, Vlissides; *Design Patterns*; Addison-Wesley 1994

[JS00] Jon Siegel, *CORBA 3*, Wiley, 2000

[JSP00] Subrahmanyam, et. al, *Professional Java Server Programming* ; Wrox Press, 2000

[KJ00] Michael Kircher, and Prashant Jain, *Lookup Pattern*, EuroPLoP 2000 conference, Irsee, Germany

[MH00] Richard Monson-Haefel, *Enterprise Java Beans*, Second Edition; Wiley, 2000

[OMGCCM] OMG, *The CCM specification*, available from http://www.omg.org

[POSA] Buschmann, Meunier, Rohnert, Sommerlad, Stal; *Pattern-Oriented Software Architecture*; Wiley, 1996

[POSA2] Schmidt, Stal, Rohnert, Buschmann; *Pattern-Oriented Software Architecture, Volume 2*; Wiley, 2000

[SUNEJB] Sun Microsystems, *EJB Specification*, available from http://java.sun.com/products/ejb