Server-Side Components - A Pattern Language

(c) 2000 Markus Völter, MATHEMA AG, Germany

markus.voelter@mathema.de

Version 2.2, 2001-04-15

Abstract

Enterprise-level component-based development is an important topic in today's software industry. Several technologies for components on the server exist, among them EJB, CCM and COM+. Although these technologies have significant differences, they are all built on the same core concepts. This paper captures these concepts in the form of a pattern language in Alexandrian style. The patterns are useful for people who have to understand the concepts behind these technologies in order to develop better applications, to develop their own component architectures, or to evaluate the three competing architectures on the market today.

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 Program [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

This pattern language uses the Alexandrian pattern form. For brevitiy, a description of this form is ommitted in the submission.

Basic Principles

Every software architecture builds on certain principles. The understanding and acceptance of 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. There are a number of principles underlying component architectures, and this part explains them. In particular, this principles are Separation of Concerns, Multitier Systems and Functional Variability.

You might ask what is the difference between patterns and principles. The most important distinction is that patterns define a specific structure and interactions. Principles, on the other hand, do not have a specific structure, they are a kind of guideline, or rationale, for the structure of specific patterns. Principles can also be regarded as high-level goals, which we want to reach with by applying the patterns. Many patterns will reference the principles in order to explain why a pattern has some specific structure.

Principle: Separation of Concerns

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

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: Sometimes there are real advantages of using new technologies, so adapting them results in a more efficient system, less cost, or other "measurable" advantages. In many cases, however, new technologies are adapted because of the hype-factor. Take WAP as an example. Currently, there are few useful WAP applications, but already many systems support it in order to be "modern".

Looking at software systems, two fundamentally different concerns can be identified: Functional concerns and technical concerns.

Functional Concerns

Functional concerns are the realization of the business requirements in a system. Another name for functional concerns is "business logic". They are usually described by domain experts in a requirements document. For example, look at the following requirement for a e-commerce application:

A customer can add products to a shopping cart by selecting the appropriate link in the catalog view. When a new product is added, the new total price of the shopping cart has to be recalculated. VAT and shipping is only added when the customer selects to "buy" the elements in the shopping cart.

This requirement could have been written by a person who does not know anything about how to implement an e-commerce application, technically. The definition of these kinds of concerns is the work for a domain expert.

Technical Concerns

When creating a software system, the functional requirements have to be implemented using specific software artifacts. A system architecture has to be defined, and things like transactional integrity, security, load balancing, failover, etc. have to be considered. This is not of interest for the domain expert, and completely different qualifications are required.

Benefits of separating the concerns

Key to efficient software development is the separation of concerns, if this is possible[1]. This has the following advantages:

- **Overall Reduction of Complexity:** If the two concerns can be separated, this results in reduced complexity of the system as a whole, because the two aspects can be considered independently. Basically, this is a specific kind of modularization, and the goal of modularization is to make complexity manageable.
- **Separation of Roles:** The developers working on the business logic, the functional requirements, don't need to care about technical concerns. They need not be experts in transactional programs, security, etc. They can focus completely on their business requirements. On the other side, technically skilled people who know the ins and outs of transactions, load-balancing, etc. don't need to care about the functional requirements (in which they are usually not interested anyway). As a consequence, developers can work on what they know best, and are thus more productive (and usually happier!).
- **Independent Evolution:** Because the two different concerns of the application are separated, both can evolve independently of the other (at least, to some extent). For example, a business process can easily be modified without changing the underlying transaction processing infrastructure, and on the other side, a new load-balancing algorithm can be introduced without chances to the business logic.
- **Reuse of technical infrastructure:** If the technical infrastructure is isolated, and therefore independent of the functional requirements, then the infrastructure can be reused in several different projects, independent of their functional requirements. Once the technical infrastructure is standardized, it can even be purchased from a third-party vendor. For example, EJB is such a standardized component architecture.

Of course, separation of the two concerns also has some drawbacks. Sometimes it is not really possible to cleanly separate out different concerns, at least not with the facilities provided by mainstream languages and tools. Multi-Dimensional Separation of Concerns [MDSOC] and aspect-oriented programming [AOP] are experimental systems which aim to solve this problem. However, as we will see, in component architectures, this principle is one of the most important ones.

Later on, the complete system is assembled from the artifacts which cover functional and technical requiremens. To make this possible, both sides have to stick to some conventions, or contracts, which are defined by the component infrastructure.

---

1 Separation of concerns does not contradict approaches like for example XP, which advocates that developers should integrate regularly and as many people as possible know as many parts of the system as possible. Within each concern, techniques like XP, etc. can and should be applied.

Principle: Multitier Architecture

Once again, have a look at "the old world". A piece of software was more or less self-contained. A program running on a certain operating system. Later on, external resources like databases, etc. were becoming a substantial part of IT application systems. Today, the requirements for mission-critical enterprise systems are different, some requirements are outlined below:

- An application has several different groups of users. Each of these groups uses the software for a different purpose. Imagine an e-commerce application. The customers browse the catalog, and buy products. Naturally, they use a web-based interface (browser) for this purpose. On the other side, the employees of the company offering the e-commerce application might want to manage orders, package them, and manage invoices and cash flow. This is a completely different requirement, and other, more flexible kinds of user-interfaces are required, such as Java or C++ application.
  To make this kind of flexibility possible, the user interface should be decoupled from the business logic itself, remote access to the business logic is necessary.

- Applications today must scale heavily. Imagine a small e-commerce site, having 1000 users per day. This can easily be hosted on one single machine, which runs the database, the webserver, etc. Now imagine this site is voted "Best innovative web shop of the month" by a popular magazine: Within a couple of days, the website may have to cope with 10 or 100 times more users. This cannot be easily handled by one machine. So the database might run on a separate host, the web server(s) could run on a cluster, and possibly, the application has to be partitioned, according to some partitioning scheme.
  To allow for such kinds of flexibility, it must be possible to distribute the different parts (or layers, see [POSA]) of an application over different hosts without requiring fundamental changes to the application itself, in order to allow for load-balancing.

- Again, consider a business which relies completely on its software applications, such as in banking, insurance, e-commerce, etc. If their system goes down for ten seconds, the business will be bankrupt. So, reliability is a very important concern here, critical system components will be installed redundantly, and fail-over strategies have to be put in place.
  Because hardware failure is even more critical than software failure in these kinds of applications, it is necessary to run specific services on more than one machine, once again requiring the possibility to distribute functionality over several machines.

A proven solution to many of these problems is a system structure known as multitier architecture. Here, the system is logically separated into several different layers. There are many possible layering structures, the most well known in the one shown in the following illustration:

The "Application client" structure is a typical application that can keep track of the state of the program. Only if real business-logic has to be used, the business logic tier is accessed. The business logic tier itself uses the data tier for storing its persistent state.

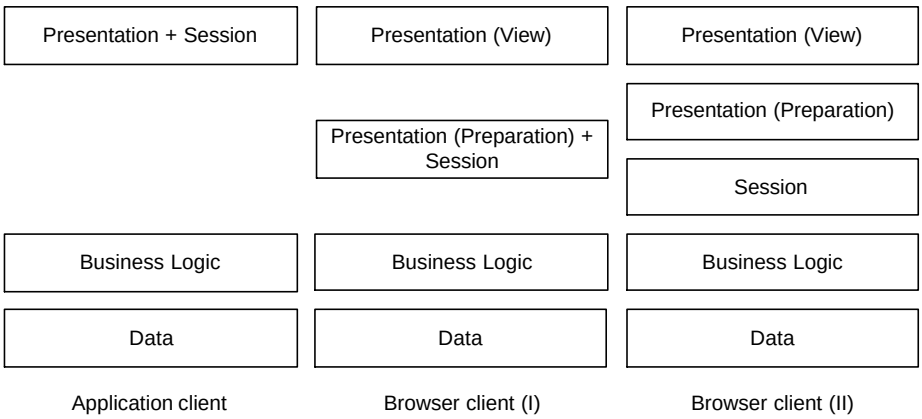


Illustration 1Different tier configurations in n-tier systems

The "Browser client" scenarios are different. A browser can only show content and collect user input (much like a 3270 terminal in the old days). The content that should be displayed has to be prepared in order for the browser to display it. This is the responsibility of the webserver. In addition, the browser itself cannot keep track of the application state. Therefore, this state has to be managed somewhere else. Usually, as in "Browser client (I)" this is done by the web server's session management. If this is not possible (e.g. because a system features several web servers, and therefore the session state has to be kept in a central place), the session management can also be achieved by a separate layer (as in "Browser client (II)"), usually this is technically a part of the business logic.

To be able to realize all the requirements outlined above, each of these layers must be accessible remotely. This allows each of the layers to run on a separate machine. It is also possible to split each of the layers and distribute the parts over different machines. How a real-life configuration actually looks like, depends on the required performance, load factor, reliability, etc. A typical installation can be seen in the following illustration:

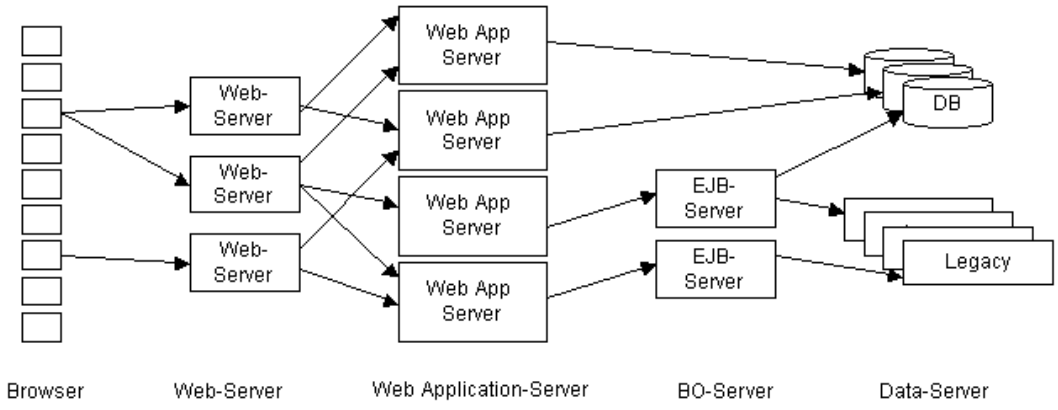


Illustration 2Partitioning of n-tier applications

In the illustration, the system is partitioned in several ways:

- The static HTML content is provided by “normal” web servers.
- Dynamic content is forwarded to web application servers which directly access a database for catalog data, user preferences, etc, usually for read-only access. They also manage sessions.
- Whenever interactive business logic is needed, the web application servers use business object servers in the next layer, which in turn access the databases or legacy systems.

This configuration ensures that the load is handled as early in the system as possible. The BO (Business Object-) servers are only accessed when they are really needed, this is also true for the web application servers. As a consequence, each of the layers can be scaled individually according to the usage profile encountered in real life.

Please note, that all of the described requirements are technical in nature. The business logic programmer should not need to care about these problems. Therefore, the application architecture should take care of this. This concern is at the heart of component architectures, as will be seen later.

Principle: Functional Variability

To be able to cope with today's requirements regarding time-to-market and flexibility, it is important to be able to reuse available software assets. The principles introduced above already help in this respect, especially separation of concerns: By separating out the technical concerns, these can be reused independently from the concrete business requirementsm and they can be purchased from third-party suppiers, which is especially important for the technical concerns.

But reuse is also necessary for the functional concerns. Have a look at the following example: A company wants to define an entity called *Employee*. To define it, the attributes and the behavior of an Employee has to be defined. This requires a sound analysis of the entity Employee. The requirements of all departments of the company have to be taken into account, which is very hard, if not even impossible. What is even harder, is to find out in which way the entity will change over time. To be able to reuse effectively, this kind of change needs to be taken into account.

Several approaches are have been proposed in order to overcome this problem. On conceptual level, domain analysis and product line architectures try to explicitly model a whole domain, creating an architecture for a family of systems as the output. These are usually implemented using frameworks or code generation. Frameworks are especially interesting, because they explicitly point out the places in the system, where it is possible to “adjust” something to the concrete requirements of an application.

Generalizing the solutions leads to a concept we call functional variability. This means, that the business logic embedded in a software system needs to be configurable to some extend. It is important to realize that this has nothing to do with technical concerns, e.g. on which

machine runs which part of the system or how security or transactions a re handled. Instead, we aim for flexibility in the concept of the entity itself. For example, an *Employee* entity could be designed to contain only the core attributes and behaviour, and can, in addition, store a list of name-value pairs to store additional attributes, which can be defined specifically for each use case (or department, in our example). The component need not be changed, and reuse is encouraged and simplified[2].

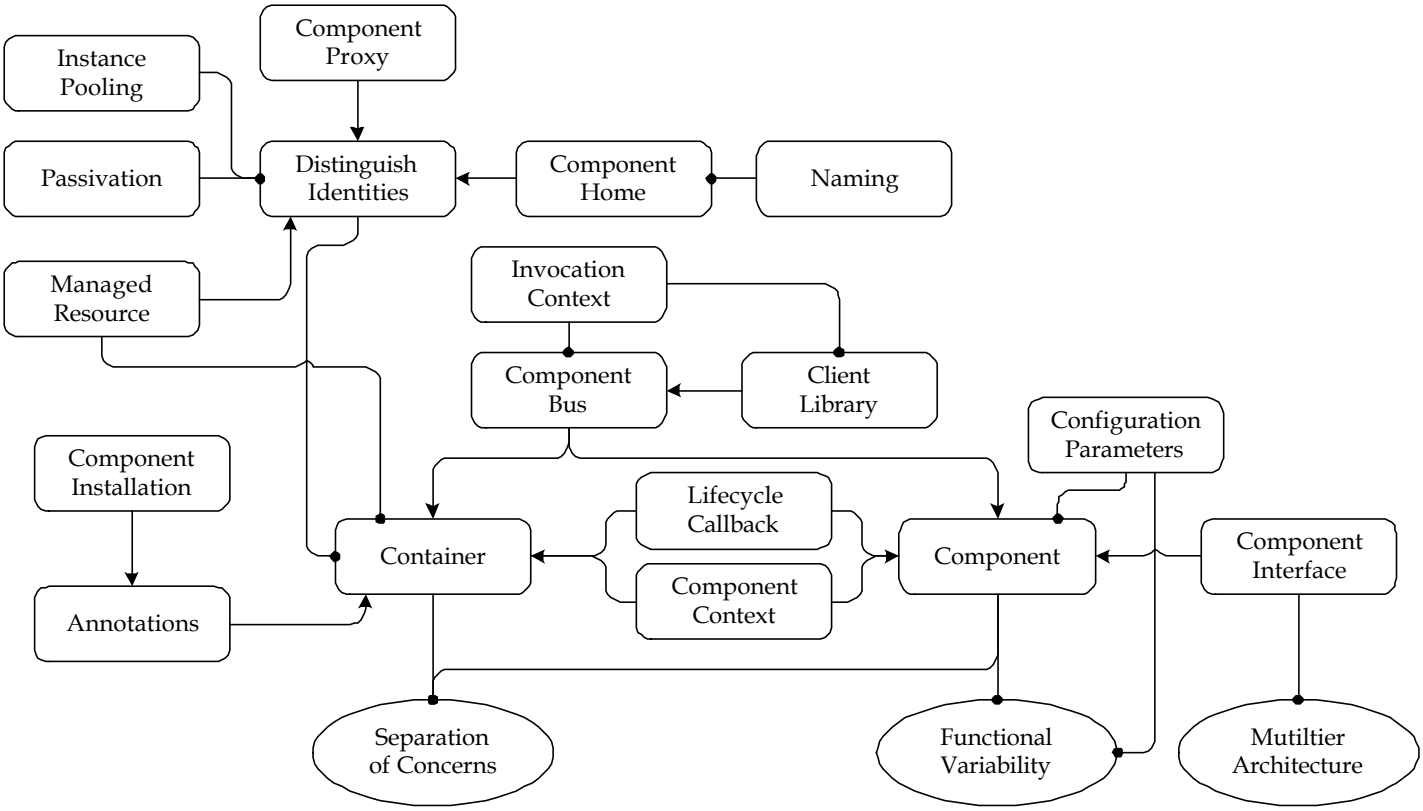
This kind of flexibility is especially important for software that is intended explicitly for reuse, which is the case in an open market of off-the-shelf components that can be used to assemble applications. Here, in order to gain a significant market share, the manufacturer needs to take into account the requirements of many customers – otherwise they will prefer to custom-develop their software and not buy off-the-shelf components.

While the component architectures usually only provide a very limited support for this kind of flexibility, some application design patterns can help significantly.

Pattern Overview

A pattern language takes its power mainly from the way, the patterns are related among each other. We have identified three distinct relationship types among the patterns. The following table shows their semantics and the graphical notation used in the above language map

2 Of course, you must make sure that the *Employee* component is still conceptually an *Employee* component. So, these additional attributes should only be used for additional data which does not change the nature of the concept. Otherwise, you just introduce a kind of weak typing, and reuse is made harder because everybody uses the component completely differently.



A ● —— B

A provides context for B

As in Alexander's patterns, the Pattern A provides the context for pattern B. That means, that once you implemented pattern A, you can also implement B. However, B is not necessary for A to work.

A ◄—— B

B is required to make A work

If you want to apply pattern A, you also need to implement pattern B, because A does not work otherwise. However, B can also be used in other contexts, that is the reason why it is a separate pattern from A. Usually, A is a higher level pattern than B, B helps to solve a particular aspect of A.

A ◁—— B

B is a specialization of A

B is a specific form of A, i.e. usually it has a more specific context than A. It provides a specific adaptation of A for certain circumstances.

A hypothetical Conversation

The following is conversation heard in a software development company. The two people talking Eric, a component software consultant and John (*italics*), a Java programmer from the company's development staff. If you don't understand all of it – don't worry. Just read the patterns and come back to the short story afterwards.

? ? ?

Ok, Eric, we have to do this new Enterprise System for 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, think so.

Are they focussed on a specific technology?

They like Java. But they say, we're free to do anything, as long as it works…

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 has some pretty neat advantages. Should we talk about that a little bit?

Well, why not.

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?

This 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 or an entity. Together, COMPONENTS and the CONTAINER the two 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?

First thing is, you can reuse technical and functional concerns separately, because they are kept separate throughout the lifecycle. Second, your components need to have some characteristics. Each component should only be accessed via a well-defined COMPONENT INTERFACE. Second, 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, Eric. 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 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 memory usage low.

Yes, absolutely. Basically, the CONTAINER DISTINGUISHES IDENTITIES. There is a logical identity, for example a customer McDill, 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 can be 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 COMPONENETS, I mean, basically they are some kind of remote object, aren't they?

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, know it from CORBA. But what is a COMPONENT HOME?

You know the GoF book?

Of course!

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

Mmh, you talked of ANNOTATIONS which determine the behaviour of the CONTAINER. Isn't this too much overhead 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 adapter-code is generated. This code is directly executed at runtime.

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.

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 the INVOCATION CONTEXT. The transport protocols allow for that.

Ok. I think I will have a more detailed look at these patterns...

The Patterns

This part contains the pattern language that "generates" a server-side component architecture. Enterprise Java Beans, CORBA Components, and COM+ are used to illustrate the concepts briefly, and show that the patterns have practical relevance.

The part is structured as follows. First, a hypothetical conversation between a developer and a component technology consultant is shown, that shows how the patterns can be applied in order to generate a component architecture. Then, each of the patterns is explained.

The patterns are divided into two groups – mandatory and optional patterns. Mandatory patterns have to be applied in order to generate a useful component architecture. The optional pattern may help, but the a system can also work without them. This paper only contains the mandatory patterns.

Component

According to the principle of SEPARATE CONCERNS, you have separated the functional part from the technical infrastructure. The functional part is now "one big chunk" of features .

†††

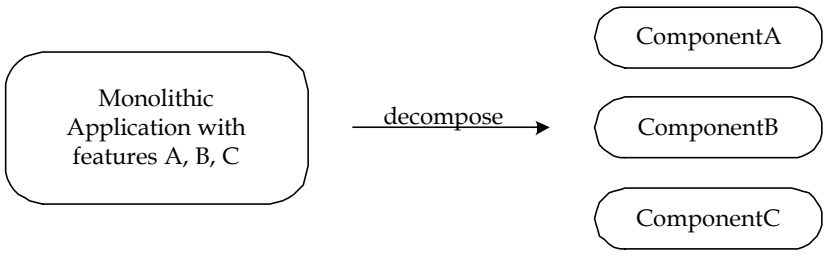
Your business is changing quickly, so do the functional requirements for your system. You want to benefit from functional reuse on an enterprise wide level on a high granularity level (in contrast to objects). You want to evolve parts of your system(s) independently from others.

An enterprise system usually contains requirements from many stakeholders. It is a collection of entities and processes which, together, define the structure and behavior of a business application.

Because of the quickly changing environment, markets, company structures, etc. all these requirements are changing quickly and you have to adapt your software implementation accordingly. These changes should be possible with as few work as possible and with modifications in as few places as possible. The main reason is that changing deployed software is expensive because you need to retest and redeploy.

Therefore:

Decompose your application into several distinct parts called components. Each of these COMPONENTS is responsible for a specific part of the whole application's functionality, and implements this part completely. A component does not directly depend on other component's concrete implementation, only on their COMPONENT INTERFACE. An application is made up of a set of loosely coupled components.



†††

There are two important concepts here: cohesion and coupling. Using these two characteristics, it is important for a component to exhibit a high level of cohesion, while the components are coupled loosely with each other. This means that each component provides features that naturally belong together; but it does not directly depend on other components' internals. To achieve this loose coupling, COMPONENT INTERFACES are used.

As a consequence, each component becomes a small application in its own right during every phase of the project. The functionalities are well encapsulated. Changes to one functionality are therefore localized to one component. Only this one component has to be redeployed Using a clearly defined COMPONENT INTERFACE which is also physically separated from the component implementation, clients need not be changed when the implementation is changed. In a way, this approach is a very sophisticated and consequent form of modularization.

Testing is also simplified. If the COMPONENT's specification is clear enough and the changed component's operations are semantically unchanged, only the changed COMPONENT needs to be tested, clients will not notice the difference. Of course, if you receive a new version of a

COMPONENT from your vendor, or if it exhibits significant FUNCTIONAL VARIABILITY, you might want to test the clients which depend on specific semantics of the changed COMPONENT.

To get the most of the component based approach, stick to the defined components all through their software development lifecycle, i.e. design, development, implementation, test, deployment, maintenance, etc. This reduces your overall project complexity and simplifies project management and organization. This approach is possible, because an application is assembled from the single Components using their clearly defined COMPONENT INTERFACE only.

For components to be really reusable, the principle of VARIABILITY must be employed. This allows them to be used in several different, but similar, contexts. Part four of this book provides a couple of patterns that can be used to achieve variability.

For different aspects of the application, different kinds of components will be necessary, such as ENTITY COMPONENTS, SERVICE COMPONENTS or SESSION COMPONENTS.

†††

EJB, CCM and COM+ are based on the concept of a component. In all three cases, an application can be built by assembling ("wiring") components. All three technologies consider the components the smallest technical building block of a component based system. There is another notion of "component" which is also called a business component [HS00]. It spans several layers (user-session-application logic-persistence) and can therefore consist of several components (component as defined here).

EJB is a relatively new component model that has no "legacy history". 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 and still being compatible to EJB. Because of this, normal CORBA clients can use CCM components, of course some features (such as multiple interfaces) are not accessible for them.

COM+ is the successor to MTS and DCOM, which in turn is an extension of COM to allow remote access to components.

Component Interface

You have decomposed functional requirements into COMPONENTS. You now have to "reassemble" your application by letting your COMPONENTS collaborate.

†††

A COMPONENT needs to collaborate with other COMPONENTS in order to be really useful. COMPONENTS should exhibit loose coupling with other components, i.e. they should not depend on other components' implementations. In practice, you do not even want to know how another component is implemented, as long as it does what it is supposed to do.

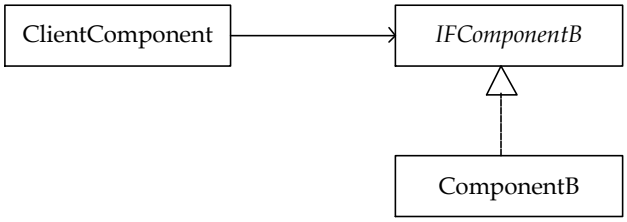
COMPONENTS are reused as black box entities. When you build applications from COMPONENTS, you assemble them based on the functionality they provide, you reuse a component as a building block, adding "glue code" to "wire" them.

To realize the potential advantages of localized functionality as described in the COMPONENT pattern, you want to minimize dependencies among components. For example, you want to be able to change the implementation of a COMPONENT without consequences for the clients of this component. You might even want to implement it in another programming language, or buy an implementation from another vendor.

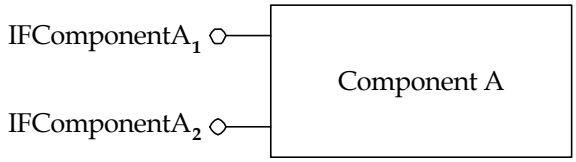
Therefore:

Define a public interface to the component that declares all provided operations and defines their semantics. A client accesses a COMPONENT using only the interface. Accessing this interface should be standardized, i.e. a binary standard should provided through the COMPONENT BUS. Make sure that the interface definition contains no assumptions about the implementation.

The following illustration shows the UML notation for a COMPONENT and its INTERFACE.



Note, that it is also possible for a component to have several interfaces. It depends on the technology if and how this is supported. Another, often used notation for components and their interfaces is the following:



Clients of a COMPONENT will only see (and depend) on the public interface, making the implementation exchangeable. It is possible to allow the CONTAINER to provide implementations for component interfaces, which is usually provided during COMPONENT INSTALLATION.

The approach has other advantages. For example, attachment to the COMPONENT BUS and DISTINGUISHING IDENTITIES is simplified, because a component can be represented by a COMPONENT

PROXY. In addition, the CONTAINER can implement the COMPONENT INTERFACE in any way it wishes, providing for different additional benefits, e.g. using INTERCEPTION.

The definition of an interface should be done in a way that does not imply anything about the implementation of the interface. An interface definition language can be used for this purpose.

A component implementation can have multiple interfaces. You might then also provide a way for a client to query the component about the available interfaces and access them separately. Using version tags, this can help to evolve each interface separately from the others. In addition, this allows a more role-oriented way of software development, where the interfaces define roles which can be "played" (i.e. implemented) by several components. As roles are a kind of *Adapter* [GHJV94], they allow COMPONENTS with completely unrelated interfaces to be used by the same client, because the role gives them the interface the client expects.

In current practice, a component interface only specifies the signature of the defined operations. The semantic meaning of an operation is usually defined only in plain textual specifications. As a consequence, a client cannot be sure, that the component does what he expects because

- ✍ The client might expect the wrong thing
- ✍ The implementer got the requirements wrong and implemented the interface in the wrong way.

To overcome these problems, you can use design-by-contract (i.e. you define pre- and postconditions for each operation which would have to be enforced by the container, see [ISE]) or state machines.

To define the technical semantics of the interface, (such as transactional integrity, security, etc.) you can use ANNOTATIONS. It is the CONTAINERs job to implement them. The business logic programmer does not need to care – this is an example of the principle of Separation of Concerns.

Some people advocate the use of completely generic interfaces. For example, the interface could consist of exactly one operation called *do(taskXML)* which takes an XML string as parameter, which, conforming to a certain DTD, specifies the task to be executed. The advantage of this approach is, that a change in functionality never requires changes to the interface – you never need to recompile clients. However, usually you still have to change the component implementation, because it has to be able to accomplish the new tasks, therefore reinstallation of the component is still necessary. Although not externally visible, you still have dependencies among the components, they are just not visible (and enforcable) through the interface. In addition, you cannot use ANNOTATIONS in the way they are intended. In can be questioned whether this is actually an advantage. For some kinds of applications it is useful, though, see WEAKLY TYPED INTERFACE in part 4.

All component technologies use this pattern. Because EJB is limited to Java (for component implementations, not for clients) the interface is specified using Java's interface feature although with some rules and restrictions on the operation parameters (see below). All EJB interfaces must inherit from javax.ejb.EJBObject. and must throw the java.rmi.RemoteException to allow for infrastructure-related error information (such as failed network connection). An EJB component (a bean) can formally only implement one interface, which can of course be created by inheriting from several base interfaces. Client references to component instances are declared using this interface(or any of its base interfaces) as their static type, therefore the code only depends on the interface.

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

In COM+, a component can also have multiple interfaces. All interfaces must inherit from IUnknown, an interface that allows a client to query the component for the other supported interfaces. Each interface is identified by a globally unique interface id which is automatically generated and therefore guaranteed to be unique. Again, clients are programmed against the interface, and know nothing about how the interface is implemented. COM+ uses type libraries, a form of COMPONENT REFLECTION, to make type information for operations available to clients.

Have a look at *Extension Interface* pattern described in [POSA2]. It describes how "[..] multiple interfaces [can be] exported by a component, to prevent bloating of interfaces and breaking of the client code when developers extend or modify the functionality of the component."

Container

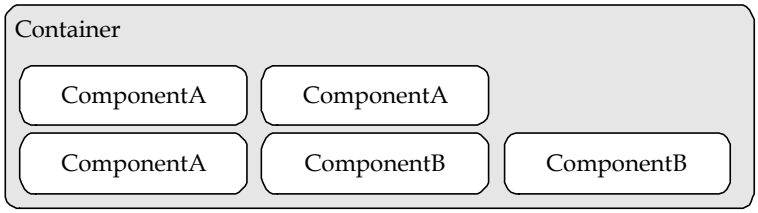
You accept the principle of SEPARATION OF CONCERNS. You have decomposed your functional requirements into COMPONENTS.

Your COMPONENTS only contain functional logic (business logic). They don't care about technical concerns. To allow the application to run, you need something that provides the technical concerns. You have to recombine the functional and technical concerns into a complete application.

Your COMPONENTS rely on "something from the outside" to supply the technical concerns of the application. Examples for such technical concerns are Persistence, Transactions, Security, Load-balancing, etc. This "external entity", the technical application architecture, should be reusable on its own. You do not want to reinvent the technical concerns for each application you create. Many of the services provided by it can be standardized implemented generically, e.g. in the form of a framework, code generation, etc.

Therefore:

Create a container for the COMPONENTS in which the COMPONENTS are executed. The container is responsible to enforce all technical requirements on the components, using standardized frameworks and other techniques such as code generation. The container and the INTERFACES of the COMPONENTS it hosts are accessed using the COMPONENT BUS.



†††

Once you have the container in place, you will need ANNOTATIONS to specify to the container how exactly it should behave for a specific component. The COMPONENTS have to be explicitly installed in the container to give the container the chance to implement the ANNOTATIONS. This process is called COMPONENT INSTALLATION. The Container will provide MANAGED RESOURCES and COMPONENT HOMES to the COMPONENTS using a COMPONENT CONTEXT. The Container uses LIFECYCLE CALLBACK to manage the lifecycle of physical component instances. To optimize performance, it will DISTINGUISH IDENTITIES.

†††

EJB requires a special piece of software to play the role of the Container. Usually, it is embedded in a larger application, a so-called application server. An J2EE conforming application server has to provide additional services, such as Naming and Transactions. Other services, such as messaging, are optional. Usually, J2EE application servers are implemented in Java, too, and provide a separate container for each component type.

CCM also uses the concept of a Container, although no implementations are on the market (at the time of writing). CCM defines different container types for the different types of components, thus there are entity, service, process and session container types. To implement

these different types of containers, the facilities already provided by the POA are used. This means, that a particular container is created based on settings for Threading, Lifespan, Activation, Servant Retention and Transaction settings. Doing this, 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.

In COM+, the role of the container is played by parts of the Windows 2000 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, pooling, etc.

A Container and its different technical aspects can be implemented using several other patterns. The remote communication aspect is usually implemented using DOC middleware based on the *Broker* pattern [POSA].

Component Bus (aka Broker)

You have a CONTAINER to host your COMPONENTS. You want to implement a multitier system (see principles)

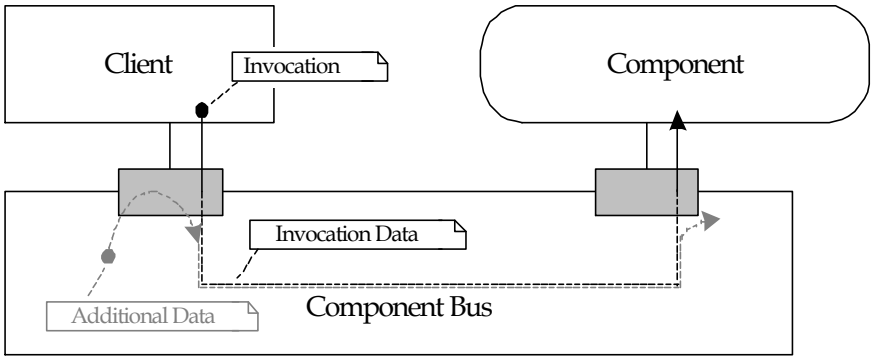
†††

CONTAINERS (and therefore, your COMPONENTS) reside on different machines than your client application, or possibly your client COMPONENTS. You must make sure that the location of the COMPONENTS remains transparent to the clients. COMPONENTS must be remotely accessible without taking care of the mechanics of remote access and the underlying tranport protocol.

Client applications and other COMPONENTS need to communicate with a specific COMPONENT. Because location is a technical aspect of an application, it should be possible to relocate a COMPONENT without affecting clients and the COMPONENT itself. This is necessary to be able to realize the principle of a multitier system, to cope with increasing load or fail-over requirements.

Therefore:

Provide a component bus, a communication infrastructure to communicate remotely. The component bus should hide the underlying low-level transport protocol. The component bus must be able to transport more information than just method names and paramters, in particular, it must be possible to transport an INVOCATION CONTEXT . Attach the component to the bus during COMPONENT INSTALLATION. The CONTAINER takes care of how to attach to the bus.



†††

A generic component bus into which clients and COMPONENTS can plug allows you to relocate COMPONENTS and their clients in any way you like. Also, the communication protocols can be switched according to the technical requirements, such as quality of service, performance, standard conformance or the underlying physical transport. Usually, the standard distributed object communication technologies (such as CORBA, RMI) are used.

In addition to the INVOCATION CONTEXT, the bus must also transport the logical id of the COMPONENT at which the request is targetted, because of the DISTINCTION OF IDENTITIES, they usually all talk to the same remote object (proxy).

To make sure that client can access the COMPONENT BUS, it is usually used in conjunction with a CLIENT LIBRARY.

†††

In EJB, the component bus is implemented using Java's Remote Method Invocation feature (RMI). RMI itself can work with different lower-level communication protocols, e.g. JRMP, RMI's native transport protocol, or IIOP, CORBA's TCP/IP transport protocol.

COM+ uses the same communication protocol as DCOM, namely Microsoft's version of DCE RPC.

CCM, as it is CORBA-based uses CORBA as its component bus. CORBA itself has pluggable protocols – they can be native to the ORB or can be one of the standard communication protocols (GIOP, IIOP).

Distinguish Identities

You have separated the COMPONENT INTERFACE from its implementation. Your design results in potentially many logical component identities, especially when you use ENTITY COMPONENTS.

†††

Your application will probably consist of many logical component instances which are referenced by clients at the same time. Each Component instance takes requires memory and other resources. For a large number of instances your container will run into resource (especially memory) problems.

Imagine employees in a big company. Say, the company has 10.000 employees, which might result in 10.000 employee component instances. Because logically, they exist all the time, your component system must be able to work on each of them, basically all the time. However, it is not possible to keep all instances active (i.e. in memory) all the time, because this would require too many resources.

Therefore:

Distinguish logical and physical component identities. A phyical COMPONENT instance can represent several logical identities at the same time. Make sure that only those logical instances are actually in memory that are really invoked upon at any given time.

†††

The total number of instances which are in memory at any given time will be much lower than the number of logically active instances. The workload and memory usage of your CONTAINER will drop. In practice, it is not possible to build reasonably sized component based systems without using this pattern. Because you still want the clients to be able to keep references to logical instances which currently have no physical identity, you will want to use COMPONENT PROXY.

There are several ways how this pattern can be implemented. For SESSION and ENTITY COMPONENTS the CONTAINER can use INSTANCE POOLING. It keeps a pool of prepared physical instances and assigns them to logical entities when needed, using LIFECYCLE CALLBACK operations on the component instances. In the case of SESSION COMPONENTS, PASSIVATION can be used, which means that components which have not been used for a specific period of time, can be removed from memory, storing the state on disk or in a database. Note that the programmer might have to do something in order to make this possible, specifically, he has to implement the LIFECYCLE CALLBACK operations correctly. The CONTAINER invokes them inform the component of the upcoming state changes.

†††

In EJB, the CONTAINER usually uses INSTANCE POOLING and PASSIVATION. To make this possible, the distinction of identities must be implemented.

In CORBA, the Portable Object Adapter (POA) allows to do exactly the same for CORBA object instances. A CORBA object is the logical instance. A Servant is a physical object that "implements" the logical CORBA object. Each CORBA object is associated with a POA instance, and each POA provides activation and deactivation policies to handle the swapping of identities. This technique is also used in CCM.

In COM+, the distinction of identities is also realized. This is easily proven by the fact that COM+ allows just-in-time activation, a feature where the physical component instance is created only when a request arrives. Pooling is also possible.

Component Proxy

You are using DISTINGUISH IDENTITIES to keep resource usage in acceptable limits.

†††

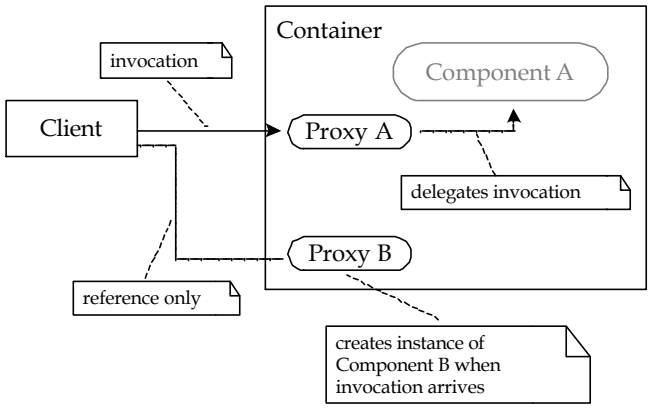
When DISTINGUISHING IDENTITIES, your clients will need to hold references to logical COMPONENT instances which are not really in memory. If the client invokes an operation on such a COMPONENT, the CONTAINER must have a way to obtain a real instance. In addition, the Container will need to do additional stuff before an invocation actually reaches a physical instance.

Consider the situation when a client calls an operation on a COMPONENT for which the Annotations specify that a transaction is required. The CONTAINER must have a chance to start the transaction before the operation reaches the COMPONENT instance – and commit it or roll it back afterwards.

If security does not allow the client to invoke the particular operation, a security exception must be thrown – again, just before the invocation reaches its target.

Therefore:

Provide a proxy for the phyiscal COMPONENT instances. Clients never have a reference to the physical component instance, they only talk to a *proxy* (usually provided by the CONTAINER). This proxy can take care of the enforcement of the technical requirements as specified in the ANNOTATIONS. In particular, it can create or assign a phyiscal COMPONENT instance to temporarily represent the logical instance, necessary to DISTINGUISH IDENTITIES. One proxy is usually responsible for several logical COMPONENT instances.



†††

As a consequence, the client always has valid references to something he considers the actual COMPONENT, although in reality he only references a proxy. Of course, this proxy implements the COMPONENT INTERFACE of the target COMPONENT. The proxy forwards the invocation to the real instance, possibly after obtaining one.

Usually, the proxy is responsible for several logical identities (of the same COMPONENT), then an invocation from the client must include the logical COMPONENT id, in order for the proxy to do the forwarding correctly. The id must be transported by the COMPONENT BUS, it is usually part of the INVOCATION CONTEXT and supplied by the CLIENT LIBRARY. It is the responsibility of the CONTAINER to find out and use a suitable number of proxies.

The proxy has a quite simple behaviour: check security, start transaction (if necessary), obtain a physical instance, forward request, finish transaction. Thus it can easily be generated, which is usually done during COMPONENT INSTALLATION.

If you need extensible functionality the proxy is a good place to plug in INTERCEPTORS.

†††

In EJB, an object called the *EJBObject* serves as the proxy. It implements the COMPONENT INTERFACE (implements in a true Java sense) and forwards the invocation to the real implementation object provided by the bean programmer. The proxy is generated during COMPONENT INSTALLATION.

The concept is also implemented by CCM, although concrete implementations are not yet available.

COM+ uses the same technique. The server-side stub (part of the marshalling-DLL) plays this role.

COMPONENT PROXY is basically an implementation of the *Proxy* pattern [GHJV94]. It plays the role of a virtual proxy, a remote proxy, and a security proxy.

Lifecycle Callback

Your COMPONENTS live in a CONTAINER and you DISTINGUISH IDENTITIES. You use INSTANCE POOLING or PASSIVATION to keep resource consumption acceptable.

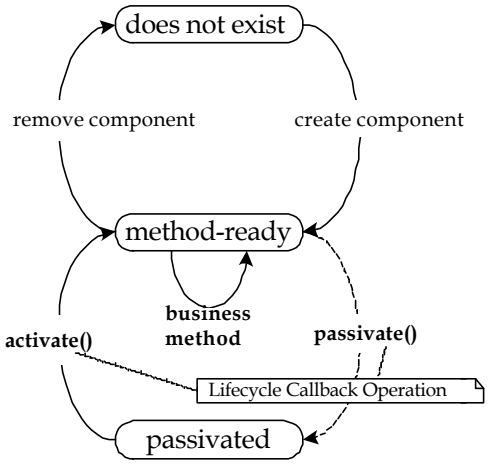
†††

A physical COMPONENT instance will represent different logical identities during its lifetime, i.e. it has to change its logical identity (INSTANCE POOLING). Every COMPONENT has to be initialized after birth. If you use PASSIVATION, the COMPONENT needs to return its resources before it is passivated, and has to reacquire them upon reactivation.

Imagine an ENTITY COMPONENT in a CONTAINER. When the physical COMPONENT is created, it must be initialized (at least) with the COMPONENT CONTEXT to allow it to get the MANAGED RESOURCES it requires. Then it is perhaps placed into a pool, where it resides until the CONTAINER needs a COMPONENT that impersonates a certain logical identity. To do this, the pooled COMPONENT must be told to "become" this logical entity, e.g. by retrieving its persistent state from a database. After it has been used by a client, the COMPONENT might be put back into the pool, or it might be passivated. Both activities require the release of MANAGED RESOURCES.

Therefore:

Provide a set of lifecycle callback operations. The CONTAINER calls them whenever it feels it is necessary. They have to be implemented correctly by the COMPONENT programmer, or, in some circumstances, they can be implemented automatically by the CONTAINER.



†††

This pattern allows the CONTAINER to implement very different policies regarding pooling, activation, passivation, reuse, load-balancing, etc. The programmer need not care about these policies as long as he implements the required callback operations semantically correct. To ensure portability across different CONTAINERS (potentially implementing different lifecycle policies), the programmer must not assume any particular policy, implicitly or explicitly. This is not always trivial.

As a variant, some of the lifecycle methods can be default-implemented by a base class, or their implementation can be created using a framework/code generator.

†††

In EJB, several lifecycle callback operations are defined. For example (ENTITYBEANS): setEntityContext() and unsetEntityContext() are used by the CONTAINER to supply the COMPONENT with the COMPONENT CONTEXT after assigning it a logical identity. ejbLoad() and ejbStore() are used to synchronize the COMPONENT with its persistent state in case of entity beans. ejbActivate() and ejbPassivate() are used to handle activation and passivation. ejbCreate() and ejbRemove() handle the creation and deletion of logical identities. Beans can only be deployed when these operations are available (the CONTAINER checks this using COMPONENT REFLECTION), but it is the responsibility of the programmer to implement them correctly.

CCM provides roughly the same operations as EJB. The set of operations of course depends on the type of component. There are operations to set the COMPONENT CONTEXT, the Session Component supports activate(), deactivate() and remove() (see POOLING and PASSIVATION for details). For ENTITY COMPONENTS, CCM provides the same operations (with other names) as EJB.

EJB and CCM provide an interesting interface called Synchronization (SessionSynchronization in EJB). It has two operations *before_completion()* and *after_completion()*. These are used to notify a component about the transaction boundaries. For example, an explicit (in-memory) rollback can be done if the transaction is rolled back.

In COM+, poolable components can be notified before they are deactivated, and after they have been activated again. If they wish to be informed, they have to implement the IObjectControl interface. It provides three operations, of which two (activate() and deactivate()) are related to lifecycle management. The COM+ runtime calls them whenever necessary.

Client Library

You are using a COMPONENT BUS to access your COMPONENTS in the CONTAINER.

†††

To build a client application, the client programmer needs to access the COMPONENT BUS an needs to know the interfaces of the components. In addition, every method invocation must contain security and other information in the INVOCATION CONTEXT. Sometimes, specific marshalling code also needs to be available on the client side.

For example, the client must access the correct NAMING context when its initial context is created. The security principal must be set on the client, in order to allow the security features to work correctly. In general, there is some cooperation on the client side required, in order for the system to work correctly.

In addition, usually, the lower level transport mechanisms (see COMPONENT BUS) require some automatically generated stubs or proxies on the client side. They take care of marshalling, etc.

Therefore:

Create a client library upon COMPONENT INSTALLATION. This library contains all the interface definitions, the automatically generated code for accessing the COMPONENT BUS, and any other parametrization or initialization parameters that are necessary for the client.

†††

Using this pattern allows the client application developer to "transparently" access the components and take part in security, etc. without too much manual programming. Only some settings (like security principal) have to be made manually.

It depends on the COMPONENT BUS whether real code is required as part of the library, or just sime definitions and properties. If the client programming environment supports the COMPONENT BUS directly then, the library does not need to contain code to access it. However, usually, at least some stubs and marshalling code must be provided. The bare minimum is usually header files or other programming language artifacts which define the COMPONENT INTERFACE of the COMPONENT that should be accessed.

†††

In EJB, most deployment tools automatically create a client JAR file which you simply copy into your client's classpath. It is used at runtime, when the client runs. The infrastructure aspects, such as security classes, and the component bus access itself it provided in the form of a JAR file that's generic for the CONTAINER.

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

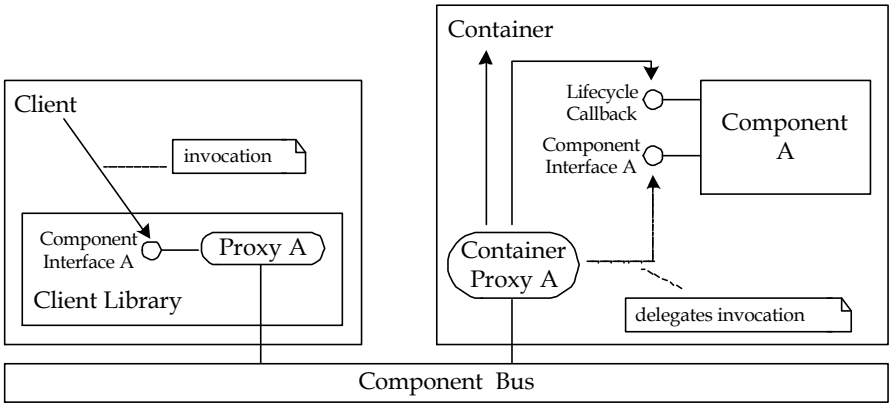
In the case of COM+, a DLL is generated, called the marshalling DLL, that contains proxies and stubs to allow the client to contact the server object. It is (dynamically) linked into the client application.

Collaboration Scenario

This section describes the collaborations between the following patterns:

- COMPONENT
- COMPONENT INTRERFACE
- DISTINGUISH IDENTITIES
- CONTAINER
- LIFECYCLE CALLBACK
- CLIENT LIBRARY

The illustration shows, how the patterns collaborate, explanations follow below:



A client, because it should be independent of any technical concerns on the part of the CONTAINER is programmed exclusively against the COMPONENT INTERFACE of the COMPONENT it wants to work with. Because, as with any remote object access technology, the client should not care about the remoteness of from a programming model point of view, the CLIENT LIBRARY provides a proxy, that implements the COMPONENT INTERFACE. The COMPONENT BUS is responsible for packaging the request and transporting it to the CONTAINER.

Because of the DISTINCTION OF IDENTITIES the client side proxy does not directly talk to the COMPONENT in the CONTAINER. Instead, a server side proxy is used. It does three different things:

- It uses to COMPONENTS business interfaces (those from the COMPONENT INTERFACE) to forward the calls from the client side proxy to the implementation that does the work.
- Before this, it uses the LIFECYCLE CALLBACK operations provided by the implementation to prepare the invocation.
- It uses hooks in the CONTAINER to check security or to manage transactions.

Component Home

You have decomposed the functionality into COMPONENTS. Your application is assembled of collaborating, loosely coupled components, i.e. one COMPONENT uses the services of another COMPONENT.

†††

A COMPONENT needs to get in contact with (find old or create new) instances of other COMPONENTS. You might not know the exact procedure of how to create or find such COMPONENTS, because this depends largely on the technical concerns (which you do not care

about because they are the responsibility of the CONTAINER.) You do not want to code this knowledge into your COMPONENTS.

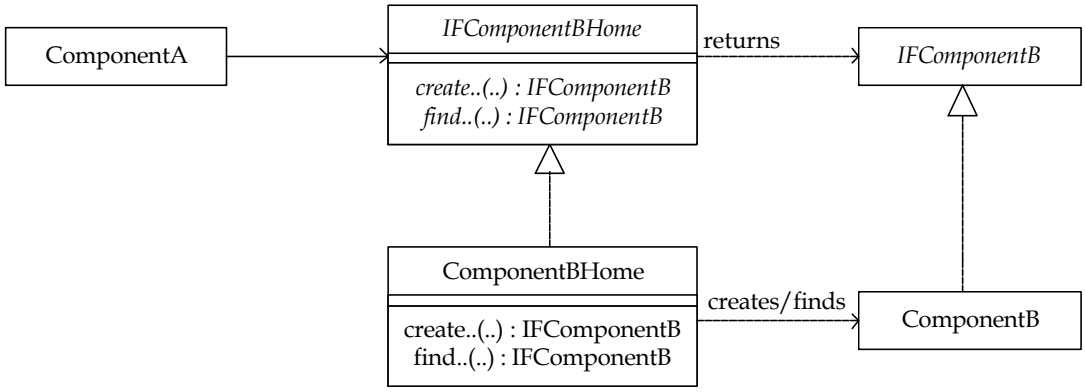
For example, an instance of a required COMPONENT can live in another CONTAINER which possibly resides on another machine. Depending on the component technology, the COMPONENT might even be implemented in another programming language.

In addition, because of the DISTINCTION OF IDENTITIES, you do not know whether really a new instance is created when you require one, or whether only a previously existing one is reused (see INSTANCE POOLING). All this is the responsibility of the CONTAINER.

For Component that have no state, there is no use in the notion of "finding" the same COMPONENT instance has you had before, thus, conceptually, you can use a new instance every time. But for stateful COMPONENTS, there is a difference in creating a new instance or finding a previuos one. Both must be possible.

Therefore:

For each COMPONENT, provide a management interface (a *factory*) that can be used by clients to create and find COMPONENT instances of the respective type. It can provide different ways (operations) how to create and find component instances, depending on the COMPONENT type (ENTITY, SERVICE, PROCESS).



†††

The component home is also an interface. I.e. the CONTAINER is free to implement it, in a way that suits its technical requirements. The steps required to implement the interface can be arbitrarily complex. This is usually done during COMPONENT INSTALLATION.

The component home is actually an implementation of the *Factory* and *Finder* patterns. As such, it is the clients' one and only access point for component instances, and servers as the lifecycle management interface for component instances.

To make the programming model simpler, you should make sure that the home has the same "look and feel" for the programmer as any other COMPONENT. I.e. it should be possible to invoke operations on a home just like on any other COMPONENT, etc.

The provided operations in a component home vary depending on the type of component. ENTITY COMPONENTS need to be created and 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. Operations to remove instances ar are also necessary, although they can also be provided as part of the components themselves.

Because a SERVICE COMPONENT has no persistent state and therefore no logical identity, no finder operations are required. Only new (logical) instances can be created. Removing instances also has no meaning here. For SESSION COMPONENTS the same is true. However, a HANDLE to a specific instance (with non-persistent state) can be stored by the client.

Of course, there must be a way to access the component home. NAMING can be used for that. Because the component home is a single access point for all instances of a component type, the global NAMING context is not polluted with entries for each component instance, instead, only the component home needs to be registered there. It can therefore be seen as a way to customize the behavior of the NAMING service.

†††

When creating a component (bean) in EJB, the programmer has to define a home interface in addition to the bean's interface. Depending on the component type, he can (must) specify several create, remove, and find operations. The create and find operations can be overloaded with different signatures, they serve as constructors for the component. Note that the programmer never really implements the home interface. Depending on the bean type and whether container managed or bean managed persistence is used, the programmer has to implement some of these LIFECYCLE CALLBACK operations in the bean class itself. The implementation of the home itself is done by the CONTAINER, to allow for INSTANCE POOLING and PASSIVATION.

In CCM, you also have to declare home interfaces for components. A home manages exactly one component type, as in EJB. 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 a create operation, homes for components with primary keys also have find (based on PK) and destroy operations. In addition to the default create and find operations, the programmer can create factory and finder operations. If such operations are present, the home interface has to be explicitly implemented by the programmer.

In COM+, a component (a CoClass) is the implementation of one or more interfaces. For each component (not interface!) the programmer has to create a factory. Programmatically, the factory is itself 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 has to use this operation to create a new component instance.Upon

creation, he can specify which of the interfaces the component implements he would like to receive.

This pattern is an implementation of the *Factory* pattern, described in [GHJV94].

Naming (aka Lookup)

You have provided a COMPONENT HOME for your components to manage the instances of a specific component type. You are using MANAGED RESOURCES in the CONTAINER.

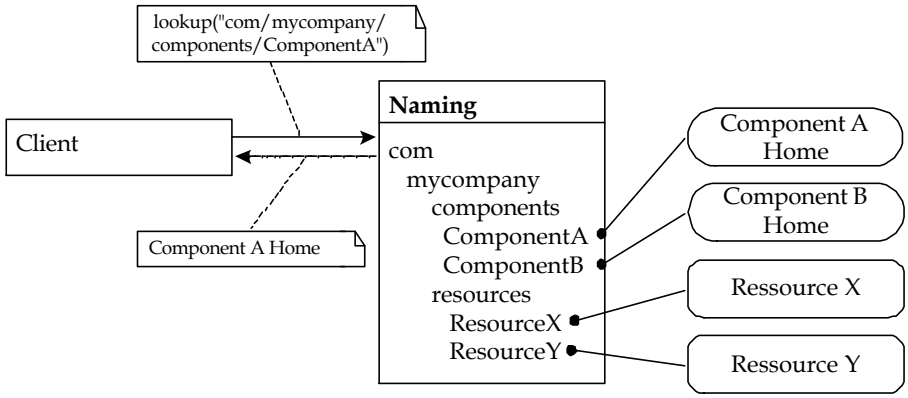
†††

To access the COMPONENT HOME or MANAGED RESOURCES, you need to get the reference of the home or the resource. How can you initially get in touch with home or service object references (bootstrapping)?

As in every system, you have the problem of bootstrapping. Before you can talk to an object, you need to have its object reference. This is true for COMPONENT HOMES as well as for any other service or resource in the CONTAINER. COMPONENT HOME provides a central access point for all instances of a COMPONENT. But to use the COMPONENT HOME, you need to have its own reference.

Therefore:

Provide a naming service which maps names to object references. Naming can be used uniformly to look up any kind of object/component/resource reference. It can be accessed by clients using a well-known object reference, which is resolved by the COMPONENT BUS.



†††

Naming provides a white pages service. That means, it maps names to object (or component) references. Usually, the name can be structured, just like file names in a file system using NAMING *Contexts*. Contexts are hierarchically structured and serve as a container for other contexts or bindings. Thus, a legal name for a COMPONENT HOME could be

/myCompany/mySystem/myComponentHome. In component based systems, the naming service has to provide access especially to COMPONENT HOMES and MANAGED RESOURCES. Usually, the naming service is provided by the CONTAINER, and by the COMPONENT BUS for the clients.

To make lookup for Components and their homes simple, you should enforce a uniform naming convention. For example, for each COMPONENT called *X*, the HOME could be called *XHome*. To avoid name clashes, you should also standardize a context structure, such as */myCompany/mySystem/…*

Naming usually provides two sets of operations:

- One set is responsible for registering and unregistering objects. Operations are usually called *bind(name, objectRef)* or *unbind(name)*.
- The other set of operations can be used to do the actual lookup. Operations are often called *lookup(name)*, and they return the reference to the remote COMPONENT (HOME) or MANAGED RESOURCE. Listing the content of a context is usually also possible through an operation called *list(contextName)*.

Usually, naming services can be federated, i.e. one naming context can be embedded in another one, just like in Unix file systems where you can mount different physical hard drives or partitions in one global directory tree.

If a suitable (non-standard) Naming implementation is used, it can also provide a simple form of load balancing. This works by returning different references for a lookup if the Naming service knows about several COMPONENTS that provide the same COMPONENT INTERFACE. Ideally, they are located on different machines.

†††

As mentioned above, every CONTAINER in EJB is usually part of an application server. This application server provides an implementation of JNDI, the Java Naming and Directory Interface. For every COMPONENT HOME of deployed beans, every service, and every MANAGED RESOURCE registered in the CONTAINER, you have to provide a name which will be used for registration in the CONTAINER's naming context. A component instance running in the CONTAINER can access this naming context using the COMPONENT CONTEXT provided to the Component instance when it is initialized and use it for lookup.

In addition, EJB provides an additional step of indirection to enhance portability. Imagine, a Component Customer uses another component Order, i.e. at runtime, it has to create or find instances of Order, using Order's home interface. Therefore, it needs to lookup Order's home in the naming context. As a consequence, Customer depends on the name, under which Order is registered in the naming context. If this Order's home name is changed, e.g. because the Component's home is relocated to another CONTAINER, the implementation of Customer has to be changed. To avoid this, EJB provides a way to map a local name to a real name. In the bean's implementation, only the local name is used. During COMPONENT INSTALLATION, this local name is mapped to a real name in the naming context. If the real name changes, only the

mapping has to be adapted. The bean's implementation does not need to be touched, and it does not need to be redeployed.

In CCM, the CORBA Naming is used for this purpose. A client looks up the Home of a component (or other services) using normal CORBA Naming lookup operations. The reference to the Naming service can be obtained using CORBA's built-in resolve_initial_references() operation, which is used to lookup well-known names, such as Naming.

There is another possibility to get in touch specifically with home interfaces based on Home Finders. A home finder is a CORBA object that can be used to retrieve references to home objects, either by specifying the repository ID of the component or by passing the repository ID of the factory.

In COM+ there is no real naming service. However, it is of course possible to locate the home for a specific component. There is a COM+ library operation called CoGetClassObject to which you pass the GUID (Globally Unique ID) of the component whose factory you want to have. The Service Control Manager, who is responsible for getting this object, uses the windows registry to find the server machine which hosts the object. It uses the same mechanism to find the server DLL that provides the object, an calls a standard operation in the DLL which returns the class object. This object's reference is then returned.

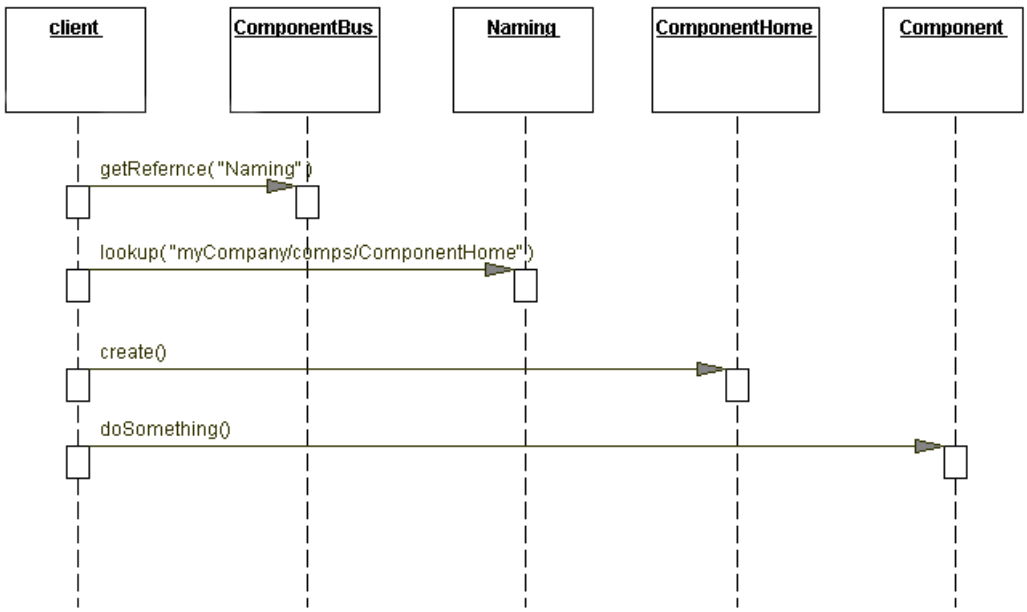
Naming is described under the name *Lookup* in [KJ00]. It is part of the *Jini Pattern Language* [JiniPL].

Collaboration Scenario

This section describes the collaborations between the following patterns:

- COMPONENT HOME
- NAMING

The COMPONENT HOME and NAMING collaborate in order to get in touch with a specific Component instance. NAMING stores a reference to the HOME, which is then used to create or retrieve the COMPONENT instance. The following seuquence diagram illustrates this process.



Usually, the COMPONENT BUS provides a way to get in touch with important infrastructure services such as NAMING. Once the client as a reference to NAMING, it can look up the COMPONENT HOME for a specific COMPONENT. The home, in turn, can be used to manage the instances of the COMPONENT.

Annotations

You provide a CONTAINER, the task of which is to take care of the technical requirements such as transactions, security, NAMING registration, etc. As a consequence, the programmer is not required to code these concerns again and again for each application, reducing bugs and enhancing the chances for reuse.

†††

However, you will need a way to control the behavior of the COMPONENTS in the CONTAINER. It is for example not enough, that the CONTAINER will handle transactions for you – you have to tell it *how* it should handle transactions for a specific COMPONENT because usually there are different possibilities how such technical concerns can be realized. You need a way to tell the CONTAINER, how it should handle certain aspects of its responsibilities, regarding a specific COMPONENT.

Take security as an example. Using a COMPONENT PROXY, it is relatively simple for the CONTAINER to enforce security requirements on COMPONENT instances. Every operation invocation is is intercepted, and the permissions are checked against a security database. But you need to tell

the container, what these permission requirements are; i.e. who is allowed to create a COMPONENT instance or invoke an operation on it. The same is true for transactions. You have to tell the CONTAINER, which operations *require* or *support* transactions, or which operations should run within a *new* transaction, etc.

Such technical concerns typically have the following characteristics:

- ✍ There is only a limited number of possible options for each feature (e.g. a transaction is *required*, *allowed*, or *forbidden* for an operation) or the options can be defined with regards to a set of data items and options (e.g. user *XY* is allowed to invoke the operation *abc()*).
- ✍ Once the above point is specified, the implementation of the requirement follows simple, predefined rules.

Therefore:

Allow the developer to annotate the components. For each COMPONENT, the programmer has to write an ANNOTATIONS file specifying the behavior of the CONTAINER when it enforces the technical requirements. The CONTAINER is free to decide how it realizes these annotated requirements, as long it is ensured that they comply with the predefined semantics expected by the ANNOTATIONS-creator.

✝✝✝

Annotations do not relieve the programmer from thinking about these concerns and making reasonable decisions about what the CONTAINER should do. For example, transaction attributes can be very important regarding the correctness of an application. But the programmer is not required to *code* these things – reducing the chances for errors

Because of these characteristics mentioned above, it is usually possible to use a GUI tool that helps the programmer to specify the annotations, providing the predefined options for selection. To facilitate the creation of such a tool, COMPONENT REFLECTION can help to access the defined operations, etc.

The CONTAINER can implement the requirements either by code generation, or by "plugging in" specific INTERCEPTORS at runtime. The code generation or the configuration of the CONTAINER is usually done during COMPONENT INSTALLATION. Because the programmer only specifies what the container should do, and not how, this leaves a lot of freedom for optimizations on the part of the container.

There are other things that are often part of annotations:

- ✍ Registration of a COMPONENT in the NAMING system
- ✍ Used MANAGED RESOURCES
- ✍ Dependencies on other COMPONENTS

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, transaction, and other information, such as whether a session bean is stateful or stateless. In EJB 1.0 this has been a serialized object, in EJB 1.1 (and succeeding versions) the deployment descriptor is an XML file. It must be supplied together with the COMPONENT implementation for deployment. The deployment descriptor can be created by hand, although all EJB server vendors (i.e. CONTAINER implementers) provide a tool to facilitate the process. These tools use reflection (see COMPONENT REFLECTION) to gain information about the component.

A CORBA component descriptor serves the purpose of annotations in CCM. It is XML based. It provides information on a component and all its supported and provided interfaces (among other things) as well as threading and transaction policies. The used interfaces are referenced (using their interface repository id), so that additional details about the interfaces can be retrieved from the repository.

COM+ also uses annotations for several aspects of a component's behaviour, in this context it is known as attribute based programming. These "attributes" are stored in a special configuration database called the COM+ catalog, and they can be specified when the component is installed using the Component Services Explorer. The aspects that can be configured by attributes are (among others) transactions, synchronization, pooling, the construction string, just-in-time activation, security, etc.

Component Installation

You express your technical requirements regarding a COMPONENT using ANNOTATIONS and your CONTAINER is capable of implementing these ANNOTATIONS.

†††

Your ANNOTATIONS state the technical requirements declaratively. However, before a COMPONENT should be made available for clients, the consistency of the ANNOTATIONS needs to be ensured, and usually code needs to be generated to realize these annotated requirements and to adapt your COMPONENT to the CONTAINER and the COMPONENT BUS.
Also, the COMPONENT HOMES need to be registered in the NAMING service before the COMPONENT can actually be used.

Using ANNOTATIONS to specify aspects of the COMPONENT's behaviour has the disadvantage, that the compiler cannot check that the ANNOTATIONS are syntactically and semantically correct and consistent with the component's implementation. But to avoid runtime errors, this is necessary. However, you do not want to force the developer to use a special tool to create the ANNOTATIONS because such a (often GUI-based) tool usually cannot be integrated easily into a make/build process.

In addition, at runtime, the most important concern is performance. Flexibility for the technical requirements is usually not needed anymore. Therefore, you do not want to

interpret the ANNOTATIONS at runtime to determine the specified behaviour. Instead, you might want to use code generation upon COMPONENT INSTALLATION.

There are other requirements for code generation. Usually, the component architecture requires the COMPONENT programmer to follow some programming guidelines/conventions which cannot be enforced by the compiler. These guidelines have to be checked, and if they are correct, glue code has to be generated which adapts the COMPONENT to the requirements of the CONTAINER.

Another problem you have is that the CONTAINER needs to publish the COMPONENT HOMES in its NAMING service, so that they can be looked up by clients.

Therefore:

Include an explicit installation step for the COMPONENTS in the CONTAINER. You provide your COMPONENT's code and the ANNOTATIONS to the CONTAINER, and the CONTAINER can do everything that's required to successfully install the COMPONENT in the CONTAINER.

†††

An explicit installation step allows your CONTAINER to introduce any glue code that is necessary to adapt your COMPONENTS functional requirement's to the technical infrastructure provided by the CONTAINER, taking into account the declarative requirements in the ANNOTATIONS. Component Installation is usually the time when the COMPONENT HOMES are published in the NAMING service.

A disadvantage is that testing becomes more time consuming, because the installation takes up additional time and effort. You should make sure that most of the functionality is tested before your COMPONENT is installed, for example by delegating complex behavior to a "normal" Java class, which can be tested and debugged without the surrounding infrastructure.

†††

EJB requires an explicit installation step as described in this pattern, it goes under the name of deployment. A COMPONENT is packaged in a .jar file. It contains the COMPONENT'S INTERFACE, the COMPONENT HOME, the bean's implementation and the deployment descriptor. When a component is deployed, the container creates a lot of code to house the component.

Just like EJB, CCM also requires an explicit installation step. Because CCM also uses languages that do not provide reflective capabilities, code generation is even more important than with Java's EJB. COMPONENT REFLECTION can help here.

In COM+ a component has to be explicitly installed. You install a component by using the Component Services Explorer. You can define several attributes (ANNOTATIONS) for the component using this tool. The information is stored in the COM+ catalog, and, partly, in the system registry. To deploy a component to another machine without setting the attributes again, manually, you can create an export file that contains all the attributes. This can be reimported on another machine.

Component Context

Your application is decomposed into COMPONENTS which are executed in a CONTAINER.

†††

Your components cannot exist completely on their own. They need to access resources outside themselves, i.e. in the CONTAINER. Or they might need to control *some* aspects of their CONTAINER, such as the transaction state. In addition, it might be necessary to provide the COMPONENT with some instance-specific information, perhaps even about itself.

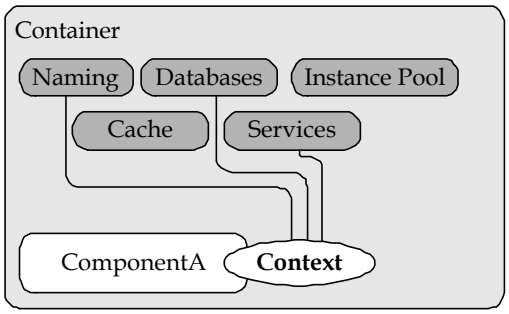
Imagine a COMPONENT that wants to store some log messages into a logging service. It needs to access this logging service somehow. Of course, NAMING can help. But the COMPONENT still has to get access to the NAMING service, so there must be a way to "bootstrap".

Because a COMPONENT lives in a CONTAINER, it is the container's job to let the COMPONENT access some controlled aspects of its environment. As an example take transactions. The programmer should not need to care, but he must be able to signal to the CONTAINER that the current transaction (no matter who started it, etc.) cannot be finished, because of an application level problem.

In addition, a COMPONENT might want to access some information about itself, such as its state (regarding lifecycle), its own load, or its own identity (PRIMARY KEY), which could change over time if POOLING is used. etc.

Therefore:

Supply each component instance with a COMPONENT CONTEXT at the beginning of its lifecycle. This context provides operations, with which the COMPONENT can access the environment, i.e. some aspects of the CONTAINER, or instance-specific information about itself. The data in the context can be updated by the CONTAINER transparently.



†††

Using this pattern allows you to let your COMPONENTS access selected aspects of its environment in a controlled manner. Only the operations provided in the context can be invoked by the COMPONENT. This is in contrast to just letting the COMPONENT access the CONTAINER's internal

state. Examples for resources accessible through a context object include the root NAMING context, database connections, the current transaction, the invoker of an operation (security principal), other components, etc.

If the COMPONENT uses the context to access information about itself, such as its own identity, you must be careful when programming COMPONENTS, that you you do not store information from the context in the instance which might be changed by the CONTAINER without the COMPONENT knowing.

†††

In EJB, the SessionContext and EntityContext interfaces are an implementation of the Component Context pattern. A bean has to implement operations, which the CONTAINER can call to set the context whenever necessary (setEntityContext() and getSessionContext). I.e. the context is available to a COMPONENT whenever it is active.

CCM defines a set of interfaces. Some are called internal and define the communication path between the component and the container. Examples are Transaction, Security, Storage. One interface, the ComponentContext interface serves as the bootstrap interface to access the others. There are different specialized contexts depending on the type of component. The container passes the Component Context to a component when it is instantiated. The mechanisms are basically the same as in EJB.

In COM+, a component can obtain its context using the CoGetObjectContext(). From a using point of view, this is also a component which provides several interfaces, among others IContextState, used to manage deactivation and transaction voting, and ISecurityProperty to determine the security ID of the caller. Information which is specific to a particular invocation can be obtained using a (formally) different context, the call context, which is available via CoGetCallContext(). Using ISecurtityCallContext, the component can access fine-grained security and caller/role information. IServerSecurity allows the component to access DCOM security information.

Configuration Parameters

You want to reuse your COMPONENTS. To achieve reuse, you need a certain degree of variability in your COMPONENT implementation. This variability can be technical or functional.

†††

To achieve the variablilty, you need a way to "pass (configuration) information to the COMPONENT" during COMPONENT INSTALLATION or later. This information must be accessible from within the COMPONENT, e.g. using NAMING or the COMPONENT CONTEXT, to allow it to adapt its behaviour.

For an example of a functional configuration, take a look at a component which manages adresses. A part of an address is the ZIP code. In each country, the ZIP code has a different

format. The format can be validated easily by using a regular expression. In Germany, for example, ZIP codes are five digit numbers, which can be validated by ^99999. To make the format of the ZIP code flexible, the regular expression must be configured during COMPONENT INSTALLATION.

For an example of a technical configuration, consider the initial size of a list. You don't want to hardcode this into your COMPONENT if the optimal size depends on the system in which the COMPONENT is used.

If MANAGED RESOURCES do not provide the additional indirection between COMPONENT-local resource name and the name of the real resource in the NAMING system, you can also specify resource names or security principal information in such paramters.

Therefore:

The COMPONENT CONTEXT should allow the COMPONENT to access configuration parameters which are defined for the COMPONENT during COMPONENT INSTALLATION, or they are part of the ANNOTATIONS. These parameters are usually name-value pairs, both of a string type.

†††

This pattern allows the user (installer) of a COMPONENT to change parts of the structure of behaviour without changing its implementation. Of course, it is only possible to configure these aspects of the COMPONENT, which the developer has made "accessible" through such a parameter.

On the downside, it is usually not easily possible to force the installer to provide values (or even correct values) for those parameters. ANNOTATIONS created by the implementer can help here, if the architecture permits. To make sure that only correctly configured COMPONENTS are used by clients, the LIFECYCLE CALLBACK operations can be used. For example, the operation invoked upon COMPONENT instantiation can verify the configuration, and in the case of problems, it can signal an error.

†††

In EJB, environment parameters can be defined in the deployment descriptor (ANNOTATIONS). They are essentially name-value string pairs, and the bean can access them using NAMING. The parameters reside in a specific context called *java:comp/env/*. The ZIP code example could be configured in the deployment descriptor as a parameter ZIPRegEx=^99999, and the bean would access it using *java:com/env/ZIPRegEx*. Please note, that you do not need this configurability for MANAGED RESOURCES, because EJB provides the indirection between the COMPONENT-local name and the real name in the NAMING system.

CCM is a little bit more advanced here. It explicitly supports the concept of component configuration. The lifecycle of a component instance has two parts: the configuration phase and the operational phase. Once the configuration phase is completed (signaled by a call to *configuration_complete()* on the component instance) the configuration cannot be changed anymore. The configuration is expressed as a set of values for the components attributes (those defined directly on the component, not on facets), thus the configuration is executed as a series of calls to attribute setters. The component can raise an exception, if the configuration

is invalid. The configuration is done by a component Configurator, an interface that provides an operation *configure(Component)*. Specific configurators can be associated with creation operations (factories) on the COMPONENT HOME.

COM+ also provides a standard way to pass configuration information from the server to a newly created component instance, which allows you to define values for variable aspects at instance creation time. This mechanism is based on the use of the IObjectConstruct interface. The deployer can define a configuration string, usually in the form name=value;name=value;… which is passed to the component instance.

Instance Pooling

You run your COMPONENTS in a CONTAINER. You DISTINGUISH IDENTITIES and provide LIFECYCLE CALLBACKS operations in your COMPONENT.

†††

Physical instance creation in the CONTAINER is expensive, because a COMPONENT instance is not just simple object, it implies a significant overhead. It is therefore useful to minimize the number of creations and destructions, especially in the case of ENTITY COMPONENTS.

To provide an extreme example, imagine a client "batch" application which changes the format of telephone numbers in all employee COMPONENT instances. It accesses every logical instance and changes the slash between area code and number into a hyphen. In a naïve implementation, for each logical component, a physical component instance will be created and destroyed afterwards, which is unacceptable. (Of course, there are better ways of doing this, e.g. directly accessing the database for bulk operations).

Therefore:

In the CONTAINER, use instance pooling to keep a number of component instances ready, which "become" different logical instances at different times. Use LIFECYCLE CALLBACKS to notify a physical COMPONENT instance of upcoming state changes, e.g. the "impersonation" of a new logical instance. Usually, the COMPONENT PROXY is used to invoke LIFECYLCLE OPERATIONS before the invocation itself is forwarded to the physical instance.

†††

Please note, that the CONTAINER must require LIFECYCLE CALLBACKS in the COMPONENTS to be able to manage their lifecycle, and they have to be implemented correctly. Pooled instances will have to behave very differently from components, which currently have a logical ID. The instance can use the COMPONENT CONTEXT to determine whether it is currently active or pooled.

†††

Because EJB provides the lifecycle methods ejbLoad() and ejbStore() for entity beans, it is easy for the CONTAINER to implement pooling, this is the reason why these operations are here. For

example, when an entity bean is taken from the pool to impersonate a new logical entity, then the CONTAINER first ejbLoad() to cause the bean to load the state associated with the primary key in the context.

In CCM, instance pooling is also a feature which is already enabled by the POA. The necessary lifecycle operations are present in the components.

In COM+ you can either leave objects alive all the time, or you can use just-in-time (JIT) activation, a form of PASSIVATION. If JIT-activation is used, many component instances are created and destroyed over time. To get rid of this performance problem, COM+ provides instance pooling for JIT-activated objects. To be poolable, a COM+ component must meet a couple of requirements, among others, it must be stateless. Pooling can be configured with a minimum and maximum instance count for the pool.

Passivation

You run your Components in a CONTAINER. You DISTINGUISH IDENTITIES and provide LIFECYCLE CALLBACKS operations in your COMPONENT.

†††

SESSION COMPONENTS need to be accessible as long as a client does not destroy them (or a timeout is reached). However, they might not be used for a very long time in between invocations. This might fill the memory available to the CONTAINER with SESSION COMPONENTS, which are not really used.

A typical example is a shopping cart. There are periods where the user does not explicitly use it, but it still must retain its state. Imagine a shopping cart would be empty after you have read the description of another product in the shop for ten minutes. But during these ten minutes, the shopping cart instance is not really used, its resources could be used otherwise.

Therefore:

Allow the CONTAINER to remove unused COMPONENT instances temporarily from memory. The state of such as passivated COMPONENT must be stored persistently and it must be reloaded upon reactivation. The programmer who develops the COMPONENTS need not care about this, as long as he implements the LIFECYCLE CALLBACK operations correctly.

†††

Note that this is not the same as INSTANCE POOLING. Although both techniques have the same goal (optimization of resource consumption) the techniques are different. Pooling will create a fixed number of COMPONENT instances which "impersonate" different logical identities during their lifetime. In case of Passivation, a COMPONENT which is not used for a certain amount of time is evicted from memory until it is needed again. Note that both techniques require a COMPONENT PROXY to make the logical COMPONENT instances available when a request for them comes in.

Passivation usually involves significant overhead, comparable to paging in operating systems. Therefore, the need to passivate should be avoided whenever possible. The server should have enough memory to keep the instances active, at least for short-lived instances. Heavy passivation is usually a clear indication for not enough server resources.

LIFECYCLE CALLBACK should be used to give the COMPONENT a chance to do something before they are passivated and after they are activated again. Examples are typically concerned with MANAGED RESOURCES.

†††

EJB uses passivation for stateful session beans. Some LIFECYCLE CALLBACK operations are there explicitly for this purpose, such as ejbActivate() and ejbPassivate(). They are used by the CONTAINER to inform the bean that it has been activated or is about to be passivated, respectively. EJB provides a neat feature: You do not have to manually return and reacquire certain kinds of MANAGED RESOURCE in ejbPassivate() and ejbActivate(), this is handled by the CONTAINER automatically. Examples are database connections. For stateless session beans passivation is not necessary, because they have no state – the next invocation can use a new (or pooled, different) instance. In entity beans, passivation is also not an issue, because they write their state to persistent storage and are managed in pools – resource usage is therefore limited.

CCM uses the same technique as EJB.

COM+ provides a feature called just-in-time activation, which means that components are instantiated only when requests arrive for the component. When the request has finished (or more correctly: the current transaction), the component is removed from memory.

Managed Resource

You run your COMPONENTS in a CONTAINER. Using the COMPONENT CONTEXT, the COMPONENTS access certain resources outside themselves, such as database connections, message queues, etc.

†††

Because the CONTAINER uses INSTANCE POOLING and PASSIVATION, YOU do not know how many physical component instances you will have, so you don't know how many resources you have to provide. Also, your components will run in different environments when they are deployed. You do not want to limit the portability of your components by depending on the type or location of a specific resource. Also, the CONTAINER must be able to integrate the resources with its security and transaction policies.

Obviously, it is not very useful to let a COMPONENT manage its own resources (technical aspect), such as database connections, for several reasons:

- Since you never know, how many physical instances of a COMPONENT the CONTAINER will create, you don't know how many instances of a resource you will need. In addition, a

COMPONENT might be subject to PASSIVATION or INSTANCE POOLING, in which case a resource must be acquired and released based on their current state (active, passive, pooled). Acquisition of resources such as database connections usually takes too long to do it just in time.

- To access a resource, you usually need to know the driver type, the location (or an identifier), a user to log in with, a password, etc. But if you install the component in another machine's CONTAINER, there might be other types of resources available, or they might reside at another location, or you need to use another driver or user/password. You do not want to limit your component's portability by hard-coding these things in your source code.
- The CONTAINER must be able to interact with the resource, e.g. using INTERCEPTION. This is necessary to allow the CONTAINER to manage transtions and security as specified in the ANNOTATIONS. If the CONTAINER does not know about resources which are accessed by the COMPONENT, this is not possible.

Therefore:

Let the CONTAINER manage resources for the COMPONENTS. This management should include

? **creating pools for every configured resource. The pool is configured externally (type, location, and other configuration such as user/password**

? **providing a resource factory to the component. The resource factory is the component's interface to acquire a resource.**

? **make the resource factory accessible through the COMPONENT CONTEXT and/or NAMING. The COMPONENT acquires the resource, uses it, and then releases it immediately afterwards to allow efficient management by the CONTAINER.**

? **The Component uses logical names to access a managed resource. The logical name is mapped externally to the resource's real NAMING name.**

†††

Using this pattern allows the CONTAINER to allocate a preconfigured set of resource when it starts up, creating the pool. When a resource is needed by a COMPONENT, there is no latency due to connection buildup, they are just taken from the pool. The (usually scarce) resources can be used more efficiently, because a component blocks a resource only while it is really needed.

Because all resources are known to the CONTAINER, it can intercept any call to a resource, and insert code to manage transactions and security as specified in the ANNOTATIONS.

The mapping from a logical name to an actual resource hides the details of the exact type and location of a resource. Deploying the COMPONENT in another environment where the resource has another NAMING name would otherwise require either a change of the name of the resource in the code, or a renaming of the resource in the NAMING system. Both is not a good solution. Using the indirection, only requires to change the mapping during COMPONENT INSTALLATION. However, in the case of a database, this is not always an advantage, because the

interface to a database, usually SQL, is not completely specified, therefore the component needs to know the type of database to create the correct SQL. In practice, a specific component can only create SQL for one specific type of database which severely limits portability.

Resource pooling also has some disadvantages. Either, a component cannot specify, with which user to log into a database, because all connections use the same user (all connections are created equal). Or, if the component specifies user and password, then the pools are usually very small and the advantages are reduced.

In addition, to make this pattern work well, the Component programmer must release a resource correctly, and as quickly as possible. Otherwise, dangling or inefficiently used resources will be the consequence (in C++ this can be achieved by using smart pointers).

†††

EJB CONTAINERS let you define connection pools. They are configured in the CONTAINER using proprietary tools or syntax. However, from the COMPONENT'S point of view they are all accessed using DataSources. A data source serves as a resource factory for the resources it manages; in addition, it manages the pool. All configured data sources are registered in the NAMING service, allowing the COMPONENTS to access them uniformly. Access can be based on logical names, which are mapped to DataSources in the component's environment parameters.

Persistence is usually handled by the Container in CCM. CCM provides a quite sophisticated persistence service definition, so connection pooling is managed by the container, too. However, because persistence support might be provided by a different vendor that the container itself, a standardized interface from container to persistence provider is defined. A component can access the persistence provider using the Storage interface. Containers are expected to manage pools of connections to persistence providers.

COM+, provides resource pooling through its OLEDB service. OLEDB is the successor to ODBC.

Invocation Context

You are using a CONTAINER to take care of the technical requirements. Among other things, the CONTAINER's job is to manage transactions and security.

†††

To be able to manage transactions and security as well as other things, the CONTAINER needs to know more than just the name of the invoked method and the arguments when an operation is called. This additional information cannot be supplied with a normal method call, because it is not possible in the syntax and semantics of the programming language.

For transactions, the so-called transaction context must be transported, which identifies the current transaction when an operation is invoked. This is especially important, when an operation does not start its own new transaction, but runs in a transaction that has been started by another operation. Note that a COMPONENT PROXY alone does not help here, information has to be transported, not just processed.

In case of security, we have the same problem. The security context must be available, which includes the caller id and its security privileges.

Note that it is not suitable to add the required information as parameters to the operation invocation, for two reasons. First, you don't want the developer to take care of these things when he programs its functional COMPONENTS, and second the kind of required information is perhaps not even known at build/compile time. In addition, it clutters the operation signatures.

Therefore:

Include an invocation context with each operation. It can be inserted or created by a stub provided by the CLIENT LIBRARY. The context can include any kind of information, only the CONTAINER must know how to handle it. Make sure the Component Bus can transport this additional data.

†††

To make this pattern work, of course, the transport protocol (i.e. the message format for the remote invocations) must allow the CONTAINER to transport this information. Note that this invocation context is only present under the hood. The real functional COMPONENT instance is not required to be able to handle such contexts (after all, in local method invocations you definitely cannot add such information to a method call). If parts of the information are necessary for the COMPONENT itself, make it accessible using the COMPONENT CONTEXT.

A disadvantage is, that this additional information makes the remote messages larger and consumes performance, and is therefore not suited for applications, which are overly constrained in these respects. And, purely local, direct invocations on instances are not possible anymore.

The proposed solution is also called *Implicit Context Propagation*, as opposed to *Explicit Context Propagation,* where the context information is explicitly handled by the application programmer. For the reasons outlined above, explicit context propagation is not suitable for components.

†††

In EJB, security and transaction information is transported to the COMPONENT. The transport protocols (JRMP or IIOP) both allow to attach this kind of information to method invocation messages. The component can access the information using the COMPONENT CONTEXT.

CCM uses IIOP as a transport protocol. It allows to attach context information.

COM+ transport protocol, DCE RPC, allows to attach context information, such as security and transaction state. The affected component can obtain this information using the COMPONENT CONTEXT.

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.

Klaus Marquart, my EuroPLoP shepherd for this paper, has worked very hard to supply me with many useful and constructive comments. Many thanks to him!

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, 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

[AOP] *The Aspect Oriented Programming Homepage*, http://www.parc.xerox.com/csl/projects/aop/

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

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

[HDSC] IBM Corp, *Hyperspaces*, http://www.research.ibm.com/hyperspace/

[HS99] Herzum, Sims; *Business Component Factory*; Wiley, 2000

[ISE] ISE, *An introduction to Design by Contract*, http://www.eiffel.com/doc/manuals/technology/contract/index.html

[JiniPL] *The Jini Pattern Language*, http://www.cs.wustl.edu/~mk1/AdHocNetworking/

[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

[PS99] Peter Sommerlad, *Configurability*, EuroPLoP 99 conference, Irsee, Germany

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