

## A Taxonomy of Components

**Markus Voelter**, Independent Consultant, Germany

### Abstract

The notion of a component is not really well defined for practical purposes. This is, because the term is used to denote many different things. So, instead of defining the term once and for all, we present a taxonomy that shows the different features of a component.

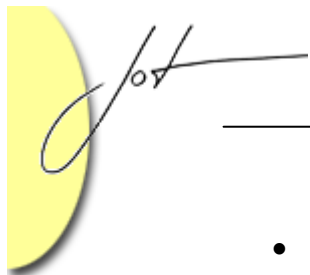
## 1 INTRODUCTION

There exist several definitions for the term component, one of them by Clemens Szyperski:

*“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.” [SZY]*

We will use it as a starting point for our further discussion: Let’s consider some parts of this definition in detail:

- “a unit of composition”: Calling a component a unit of composition actually means that the purpose of components is to be composed with other components. A component-based application is thus assembled from a set of collaborating components.
- “contractually specified interfaces”: To be able to compose components into applications, each component must provide one or more interfaces. These interfaces form a contract between the component and its environment. The interface clearly defines which services the component provides. It thus defines its responsibility.
- “explicit context dependencies only”: Usually, software depends on a specific context, such as available database connections or other system resources being available. One particularly interesting context is the set of other components that must be available for a specific component to collaborate with. To support the composability of components, such dependencies be explicitly specified.



- “can be deployed independently”: A component is self-contained. Changes to the implementation of a component do not require changes (or a reinstallation) to other components. Of course, this is only true as long as the interface remains compatible.
- “third parties”: The people who assemble applications from components are not necessarily the same as those who created a component. Components are intended to be reused - the goal is a kind of component marketplace where people buy components and use them to compose their own applications.

This definition of the term component is very generic and thus it is not surprising that the term is used to describe rather different concepts: subsystems, DLLs, JavaBeans, ActiveX controls, .NET assemblies, Enterprise JavaBeans, COM+ components, CORBA components and more. The purpose of this article is to try to clarify these different views by providing a taxonomy for components. We therefore don't try to give one concise, closed definition. Instead, we will show the different features and characteristics a component must or can have, thereby classifying the different kinds of components as they are used today.

## 2 A FEATURE DIAGRAM FOR COMPONENTS

The following discussion centers around a feature diagram for the concept „component“. Feature diagrams are a notation that is used as part of feature-oriented domain analysis (see [FODA]), a technique used for analysing the different products to be covered by a software product-line. This approach, as well as the diagram notation, is extensively explained in [GP]. For those who don't know the notation, rest assured that the article is written in a way that will clarify the diagram semantics on the fly. The diagram we use as a basis for our discussion is depicted in the illustration.

In this diagram, the features (denotes by the boxes) of the concept *component* are described, which is located at the top of the diagram. The boxes directly connected to *component* are the direct subfeatures of a component. The little circles at the edges connecting the features define the semantics of the edge. A filled circle means „mandatory“. Thus, every component has a feature *service interface*, is of a certain *kind*, requires certain *resources*, and plays a certain *role*, and it is *deployable*. Optionally (denoted by the outlined circle at the edge), it provides *meta information*. Let's elaborate on these issues.

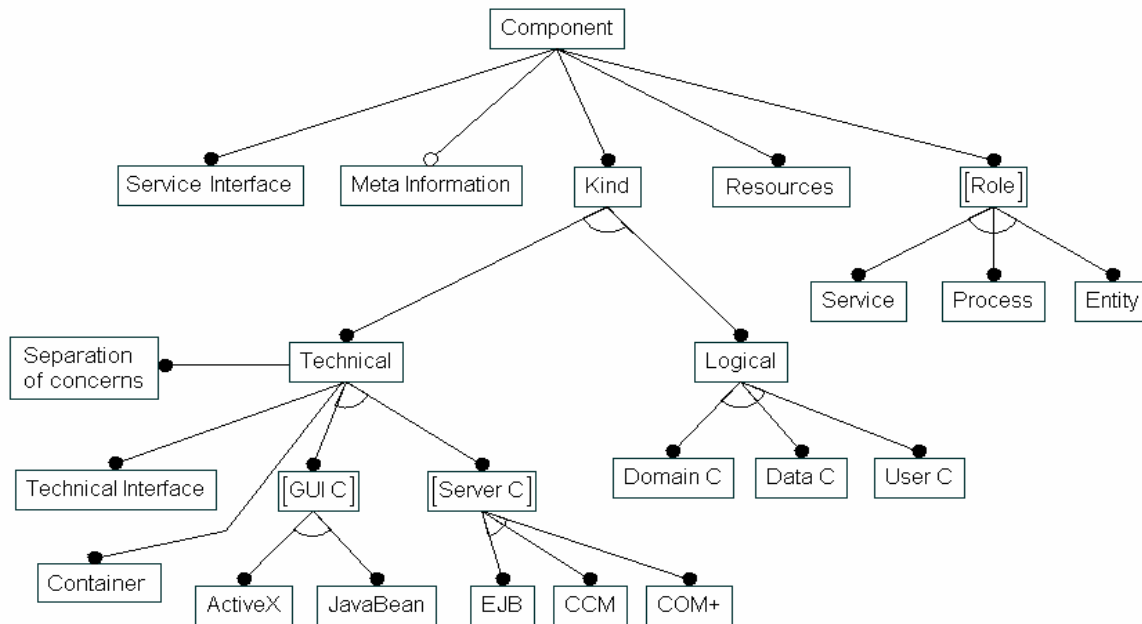
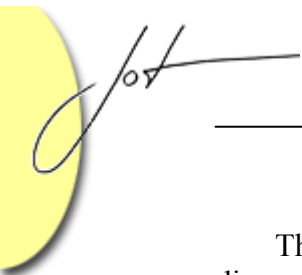


Fig 1: A feature diagram for components

## Discussion

As already stated in Szyperski's definition, a component „is a unit of composition with contractually specified interfaces“. It is thus necessary that a component specifies the services it provides to clients (which can also be other components). Typically, these services are specified in terms of operations including their parameters and types in a *service interface*. Ideally, this interface also specifies the semantics of the operations. There are several ways how this can be done: the most popular ways are based on design-by-contract [DBC] (i.e. pre- and postconditions for operations) or state machines (which specifies the legal invocation sequences and possibly timing constraints). In most component-based systems, the *service interface* does not specify any semantics however, only the provided operations and their signatures.

Because a component is intended to be (re-)used as a building block during the assembly of complete applications, it is necessary that it specifies the *resources* it requires to run. Typical resources are database connections, message queues or the service interfaces of other components that the current component uses to have some services fulfilled. Ideally, the resources should be specified in a way that allows the environment to detect whether an application – a collaboration of components – *can* run, or whether resources are missing. Enterprise JavaBeans [SCP], for example, specify these resources as part of the XML deployment descriptor, the Small Components prototype [SMC], a component framework for small devices, uses a normal Java interface, together with some naming conventions, to specify a component's resources.



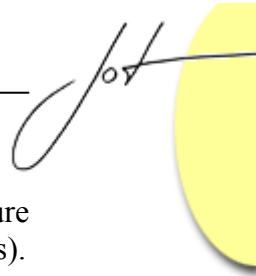
---

The next feature to be discussed is the *role* a component plays. As the feature diagram shows, a component can play one of several roles (a one-of-many selection is denoted in the diagram by the outlined-filled arc connecting the three subfeatures of *role*). The most typical are:

- A component can represent an *entity* such as a person, an employee or a text edit field. It has state, which can be persistent.
- It can represent a *service*. This means it is typically stateless. An example would be a component that provides an operation to calculate the VAT for a specific product or do some mathematical calculations, or a wrapper around some physical device or legacy system.
- A *process* component is one that encapsulates a process like the filling of a shopping cart or a complex workflow. It is typically stateful, usually it is not persistent.

A component also is of a specific *kind*. This is a very important concept because people typically confuse these two *kinds* of components and end up in endless discussions. Components can either be *technical* or *logical*. A logical component is simply a package of related functionality. It can be some kind of subsystem, a DLL or a complete, standalone application that runs as part of a larger system. Logical components are mainly a way to keep the complexity of a system under control, and to organize version control or project management issues. Often, the other features (like service interface, resources, role, and meta information) are specified informally – not necessarily tool-readable. There are basically three kinds of logical components. *Domain* components provide business logic; *data* components provide access (and optionally, validation and conversion) to data; *user* components are part of the client application and typically provide user interface functionality and/or access to domain or data components. There is also the notion of a *business* component – an aggregation of data, domain, and user components that embody a complete subsystem.

On the other side there are *technical* components. These are technical building blocks to assemble applications. Here, the concepts explained above (service interface, role, resources, meta information) are specified „formally“, they can be evaluated and understood by a tool. Such a tool is usually called a *container*. The container provides a runtime environment for the component. The component cannot live (i.e. run) outside a container. Their single purpose in life is to be used inside a container. Why is this so? The reason is, that we want to apply the concept of separation of concerns: we want to keep different aspects of an application separated into different technical artifacts. A container handles the so-called technical concerns for the components. What these technical concerns are, depends on the application domain in which the component architecture is used. In business systems the technical concerns are things like transactions, security, failover or load-balancing. The container handles these technical concerns for the components – the component developer does not need to implement them manually in his code. To allow the container to host a technical component and handle the technical concerns for it, the component has to provide a *technical interface*, with which the



container can access all hosted components in a uniform way, for example to configure them, start them, stop them, query them for their state or whatever (see [SCP] for details).

Components optionally provide *meta information*, which means that they can provide information describing themselves. The specification of resources is often part of that meta information. Meta information can be available for *runtime* or *build time*, or both. Build time meta information is important for application assembly tools, tools that support the assembly of applications from components. Runtime meta information can be used by clients to gain insight into the features a component can provide to them.

Today, there are two main uses for technical components. They are either used in client applications, or, in multi-tier systems, on the server. The container for client components is typically an IDE where the components are configured at development-time. These client components can either be visible or invisible. Server components usually encapsulate business logic in multi-tier systems. The container is typically a part of an application server.

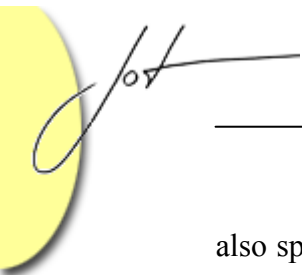
There is an important relationship between logical and technical components: in many cases, logical components are aggregations of technical components – i.e. technical components are used as building blocks for logical components.

As a last point, components are separately *deployable*. In a component-based application, it is possible to replace a component with a new one, as long as it provide the same, or a compatible service interface, usually at runtime. This significantly simplifies deployment, maintenance and operation.

### 3 TECHNOLOGY EXAMPLES

Let's look into some concrete examples for the concepts we explained above. Let's start with client-side components. The most popular examples are ActiveX controls and JavaBeans. Both are typically used as building blocks in IDEs, usually they are visible. They are used as text fields, timers, or more complex widgets, or provide access to databases, etc. ActiveX controls as well as JavaBeans provide meta information at build-time and at runtime. In case of JavaBeans for example, the meta information can be accessed either by Java's reflection or by querying the associated *BeanInfo* class. In multi-tier systems, these components are used as user components, in client/server or standalone applications they are used as user, domain and data components.

In case of server-side components, there are three mainstream examples: Enterprise JavaBeans (EJB), Microsoft's COM+ and CORBA Components. They are used in enterprise business applications. All reside in a container which takes care of transactions, security, load-balancing, failover and other features. Again, they provide meta information, mainly for use at build time (or deployment time). In case of EJB, this is an XML file called a deployment descriptor, in case of COM+ these are attributes stored in the COM+ registry. In all cases, the components provide a technical interface for the container (called *SessionBean* or *EntityBean* in case of EJB). The container uses it to control the (not always trivial) lifecycle of component instances. These components



also specify the resources they need in order to allow the container to provide them to the components. In the CORBA Component Model, for example, they are specified as part of the configuration XML file. These server components are never used as client components, because the containers are rather complex and not available at the client site. Usually, they are used as domain components, sometimes also as data components. There are different specifications for service, process and entity components. They typically differ in their lifecycle and in the services the container provides for them. For example, the EJB container provides concurrency synchronization for Entity Beans, but it does not provide this feature for SessionBeans.

Windows DLLs are thus not really technical components. While the exported operations can account for the interface, but they don't really provide meta information. They also don't specify the resources they require. There is also no container that provides services for them. A technical interface is also not available of course. Their purpose is a way to enhance reuse, and to allow the dynamic loading of application functionality. .NET assemblies are roughly in the same category, although they do feature a sophisticated metadata facility. Standard metadata (which types are in the assembly, which modules, and which additional resources) is added by the respective compiler and it is even possible for the developer to add additional metadata items to types, operations and fields using .NET attributes.

## 4 SUMMARY

The goal of this article was to define the term component as it is used today in the industry. Of course, this is also just the opinion of some people, those with whom I discussed the ideas during the writing of this article. If you have another opinion or any comments, I'd be glad to hear from you at [voelter@acm.org](mailto:voelter@acm.org).

## REFERENCES

- [DBC] ISE; *An Introduction to Design by Contract*  
<http://www.eiffel.com/doc/manuals/technology/contract/page.html>
- [FODA] SEI; *Feature-Oriented Domain Engineering*,  
<http://www.sei.cmu.edu/domain-engineering/FODA.html>
- [GP] Ulrich Eisenacker, Krzysztof Czarnecki; *Generative Programming*, Addison-Wesley 2000
- [SCP] Markus Voelter, Alexander Schmid, Eberhard Wolff; *Server Component Patterns*, Wiley and Sons, 2002
- [SMC] *Small Components Project*, <http://www.voelter.de/smallComponents>



[SZY] Clemens Szyperski; *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley, 1999

### About the author



**Markus Völter** works as an an independent consultant for software technology and engineering. He focuses on the architecture of large, distributed systems. Markus is the author of several magazine articles and patterns, a regular speaker at conferences and co-author of Wiley's *Server Component Patterns*. He can be reached at [voelter@acm.org](mailto:voelter@acm.org) or at <http://www.voelter.de>