

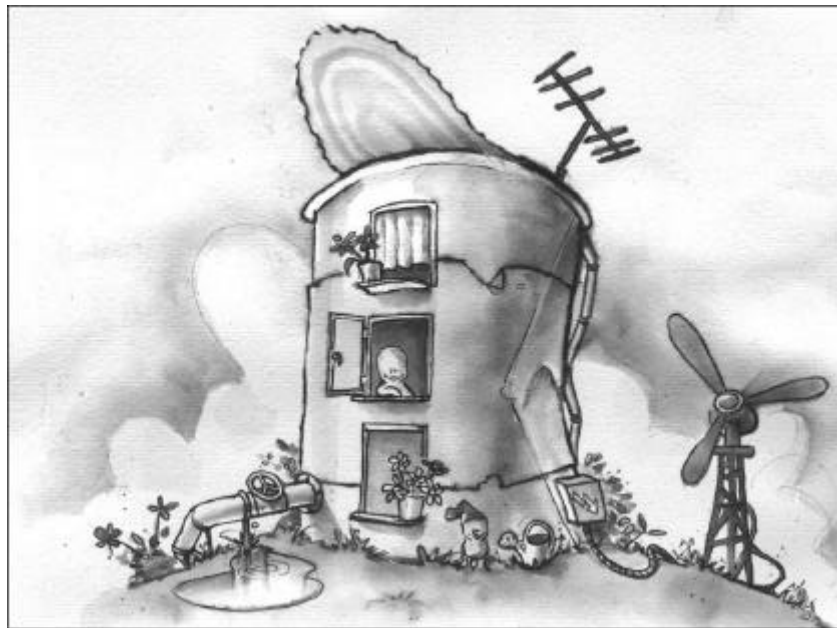
# 1 Core Infrastructure Elements

---

## Container

---

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



† † †

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

**Moreover, you want to be able to reuse the technical concerns effectively.**

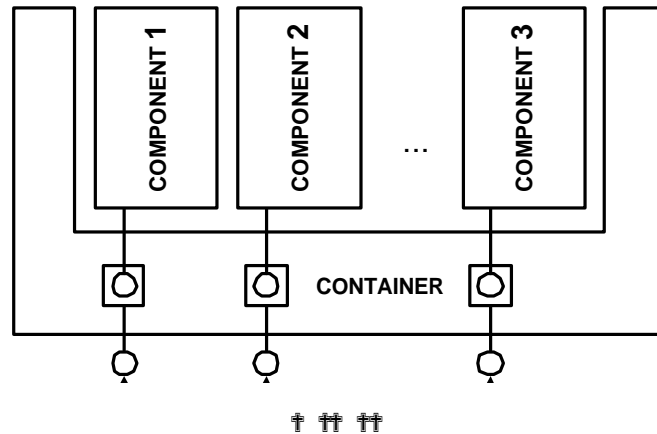
There are two kinds of technical concerns: those that clients require, such as security, transactions, and those that the COMPONENT itself requires, such as concurrency, persistence, and access to infrastructure resources. COMPONENTS not only need to benefit from these technical concerns without being tied to a particular infrastructure or platform, such as a transaction monitor or a security infrastructure. These services must also be provided non-intrusively: they have to be added 'from the outside'. From the client's perspective, the COMPONENT should appear to be a single entity providing their functionality integrated with the required technical concerns. From the COMPONENT's perspective, the technical concerns should be provided 'magically' – the COMPONENT code must not be 'polluted' by code implementing technical concerns.

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

Therefore:

**Provide an execution environment that is responsible for adding the technical concerns to the COMPONENTS. This environment is generally called CONTAINER. Conceptually, it wraps the COMPONENTS, thus giving clients the illusion of tightly-integrated functional and technical concerns. To implement these technical concerns in a reusable**

manner, a CONTAINER uses frameworks and other generic techniques such as code generation.<sup>1</sup>



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

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

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

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

As the different COMPONENT types - SERVICE, SESSION and ENTITY - have different characteristics, you need to provide different CONTAINERS. These are integrated in an APPLICATION SERVER that provides all the services common to the different CONTAINER types.

The management of resources used by a COMPONENT such as a database connection is also a technical concern. The CONTAINER must therefore MANAGE RESOURCES on behalf of the COMPONENTS it hosts. The CONTAINER cooperates with the APPLICATION SERVER over this.

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

† † †

EJB requires special software that plays the role of the CONTAINER. Usually, the CONTAINER is embedded in a larger application, a J2EE APPLICATION SERVER. A J2EE-conforming APPLICATION SERVER has to provide additional services, such as NAMING, accessible through the JNDI API, transactions, or Servlet and JSPs for the dynamic creation of web content. From EJB 2.0 onwards, messaging is integrated with EJBs in the form of Message-Driven Beans, which are Beans that act as 'message receivers'.

Usually, J2EE APPLICATION SERVERS are implemented in Java and provide separate CONTAINERS for each COMPONENT type. EJB CONTAINERS have to conform to the EJB specification to allow seamless exchange of COMPONENTS between different CONTAINER vendors. Nevertheless, they are free to offer different levels of quality of service. For example, some CONTAINERS might offer load balancing, high availability, or advanced caching strategies while other CONTAINERS do not.

CCM also uses the concept of a CONTAINER, although no complete implementations are on the market at the time of writing. CCM explicitly defines different CONTAINER types for the different types of COMPONENTS, thus there are entity, service, process and session CONTAINERS. To implement these different types of CONTAINERS, the facilities already provided by the Portable Object Adapter<sup>2</sup> (POA) are used. This means that a particular CONTAINER is based on settings for threading, life-span, activation, servant retention, and transaction policies. Because all this is already part of 'ordinary' CORBA, it is possible to access a COMPONENT from a client that does not know that the CORBA object to which it has a reference is actually a COMPONENT. Some features such as multiple interfaces are of course not accessible

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

for ordinary CORBA clients. In addition to the technical concerns mentioned above, CCM CONTAINERS also handle persistence in a well-defined way. The persistent state of a COMPONENT can be specified abstractly, and a mapping to logical storage devices can be provided. These logical storage devices are then mapped to real databases with CONTAINER-provided tools.

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