Introduction

The many definitions of what a component is—and what it is not—often try to define a component in opposition either to object-oriented programming languages or to architecture description languages (ADLs). This chapter does not intend to provide yet another definition, but tries to clarify the situation showing the models of the different component technologies currently available in the industry: JavaBeans [1], COM+ [2], CCM [3, 4], .NET [5, 6], and the Open Service Gateway Initiative (OSGI) [7].

A difficulty is that industrial component technologies focus on practical problems and are described in technical terms, thus providing many implementation details and making it difficult for users to understand concepts and principles. Available documentation is either oriented toward decision makers and provides mostly marketing arguments, or it is oriented toward programmers or tool implementers and includes pieces of code and complex API specifications.

The ADL community, by contrast, has tried primarily to establish high-level concepts and principles and has produced many research papers. A large amount of work has been dedicated to the analysis and comparison of the different features proposed by the various ADLs [8].
Whereas component technologies focus on the last phases of the life cycle, that is, on implementation, deployment, and execution, ADLs focus on the early stages, mainly on design. Indeed, comparing component technologies and ADLs is difficult because they do not address the same problems and they are not described at the same level of abstraction. It is useful, however, because they share the component concept and their scope is complementary.

The goal of this chapter is to describe component technology in the same terms as ADLs at the same abstraction level. Of course, this chapter does not intend to provide in-depth coverage or present the different facets of these technologies. We shall try instead to identify the component model underlying each component technology. This allows us to identify similarities and differences, both between the various component technologies and between ADLs and current component technologies.

The remainder of this chapter is organized as follows. In the next section, we provide a short historical perspective showing where component technology has its roots. Then we present a broad overview of the main features provided and introduce the notation used in this chapter. The next section briefly introduces Acme as a representative of the main concepts typically found in ADLs. The subsequent sections briefly present industrial component technology, namely, Sun’s JavaBeans, OMG’s CORBA Component Model (CCM), Microsoft’s .NET, and OSGI. Finally, the chapter is summarized.

A Short Historical Perspective

Programming languages, apart from their linguistic aspect, can be seen from either the run-time point of view (run-time support while executing the program) or from the design and reuse perspective (designing new programs and reusing existing programs). The first perspective provides transparent support during program execution, dynamic loaders and binary interoperability being examples of the (extrafunctional) properties added to a program by run-time support. The second perspective, reuse and design, leads to more abstract concepts such as classes, inheritance, and polymorphisms and to methods and notation like UML.

To a large extent, component-based technology is the logical continuation of the first perspective (introspection capability and run-time support), whereas ADLs are the logical follow-up of the second (design and reuse). Take the example of dynamic loading. The association of the language run-time support with the operating system, and the addition of a limited form of
introspection to executables (a table of symbols with their offset), is sufficient to provide new properties for applications (incremental loading, module substitution, partial recompilation, portability, etc.) without any change in the application source code.

It is the development of this idea that is the basis for component-based technology: By relying on the component framework, it is possible to transparently provide new services and remove information from source code. These goals can only be reached by improving source code introspection or by defining new formalisms. Indeed, the approach was used to (1) explicitly define components and their interactions with other components (to perform static connection checking and automate connection control), (2) explicitly define their implementation from native code and interactions with other “native” code (to dynamically instantiate and control instance life cycle), and (3) explicitly define components (extrafunctional) properties and services (to delegate their realization to the component framework). Indeed these three goals were addressed, more or less simultaneously, by different groups, and ended up in different systems.

Component Interface and Connections

ADL
The ADL community focused on the definition of a component, its interface, and relationships with other components. ADLs primarily address the issues related to the early phases of software engineering: design and analysis. They identify a number of concepts such as architecture, configurations, connectors, bindings, properties, hierarchical models, style, static analysis, and behavior (see Chapter 3). Simultaneously, but on a different abstraction level, industrial component technology addresses similar issues, emphasizing more pragmatic aspects.

Improving Code Independence
The compilation techniques used in classic languages like C and C++ are such that even a simple change in a program could result in dramatic recompilations that increase development costs and reduce reusability. The idea of improving code independence was to improve the independence of “programs,” relying on the definition of explicit “interfaces” and on a simple interaction protocol (e.g., asking for a “connection” to an interface).

COM falls into this category. It proposes an improved binary interoperability convention and a few composition techniques: containment
and aggregation. Dassault Systèmes’ component model improved the composition technique with extensions and delegation (see Chapter 19).

Simplifying Composition

JavaBeans focuses on increasing composition capabilities and reuse. The system is based on considerably extended introspection capabilities and on many programming conventions. The result is that beans are highly reusable and can be easily composed, statically or dynamically. Specific assembly tools have been developed, for example, Net Bean [9].

Performing Services Transparently

Performing Distribution

The goal here was to provide a way to realize distributed applications almost transparently. The remote procedure call (RPC) is the ancestor of this very successful line of work. CORBA Object Request Broker (ORB) extended the approach with object-oriented (OO) and language heterogeneity. COM+ relies on RPC.

In these approaches, introspection requires “full” knowledge of the interfaces, which are provided in the IDL and processed by a compiler. Runtime support requires more resources, the ORB itself being only one of the required run-time components.

Generalizing Services

The goal here was to generalize the big success of distribution support on a number of other extraneous properties. This produced a proliferation of “services” (see CORBA services [10] and MTS specification) including transaction and persistency. The complexity of each service, and, much worse, of their combination, led to their encapsulation under the concept of container (EJB). The container is an abstraction for the set of services to be applied to a component.

The CCM, adding among other things language heterogeneity, generalized the approach.

Here introspection is more demanding, and the description of the services to apply to a component requires specific formalism(s). The container is an abstraction for the set of services to apply to a component, and its implementation is a mixture of run-time interpreters (code introspection and container information), of objects generated when compiling the container description (stubs, skeletons, etc.), and of general run-time services (ORB, DBs, etc.).
In an abstract way, in these approaches, a component is a box. The box itself is the container that describes and controls the access and properties of its content; and the content is the “functional” or executable code of the component. The box manages the content but the content ignores the box. These approaches emphasize the distinction between the box itself (described in a number of specialized formalisms), and the content (described in classic programming languages, object-oriented or not). The box describes how to manage the content at execution.

Notation and Plan

A component model specifies, at an abstract level, the standards and conventions imposed on software engineers who develop and use components. Compliance with a component model is one of the properties that distinguishes components from other software entities.

In this chapter we call a component framework the set of (1) the formalisms used to describe some aspects of components; (2) a number of tools, including the above formalisms, compilers, design tools and others; (3) the run-time infrastructure that interprets part of the above formalism and enforces specific properties; and (4) a number of predefined components and libraries.

Though imprecise, industrial component models often make assumptions about important steps in an implicit product life cycle. Table 4.1 gives a rough picture of the major steps that could be identified.

In the following discussion, we look at different industrial component models. For each model we will provide the intended goal of the model and describe it from different aspects: (1) interface and assembly, (2) implementation, (3) framework, and (4) the life cycle.

Table 4.1
Majors Steps in CBD Life Cycle

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Phase</th>
<th>Actor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface</td>
<td>Definition</td>
<td>Designer</td>
</tr>
<tr>
<td>Assembly</td>
<td>Assembly</td>
<td>Architect</td>
</tr>
<tr>
<td>Implementation</td>
<td>Implementation</td>
<td>Developer</td>
</tr>
<tr>
<td>Life cycle</td>
<td>Packaging, deployment</td>
<td>Administrator</td>
</tr>
<tr>
<td>Framework, run-time support</td>
<td>Execution</td>
<td>End user</td>
</tr>
</tbody>
</table>
For the purpose of this chapter, we will use a simple graphical formalism, in which a component is a gray box (Figure 4.1). The closed box is used to represent the interface of the component and the concepts provided to model the interaction points of the component with the external world. In all ADLs, as well as in most component models, the unique (meta) entity is the component, and interaction points are only relationships between components. We will instead identify three classes of “entities,” the components (gray), the run-time component framework (black), and the other users programs (white). We emphasize the relationship of a component with (1) the other components (left and right), (2) the component run-time framework (bottom), and (3) the usual programs that are no components (top).

**Acme ADL**

Many architecture description languages have been proposed, sometimes to satisfy the needs of different application domains. Different surveys have been published (e.g., [8]). Although there is little consensus about what exactly an ADL is, there is agreement about what the set of core concepts could be, at least regarding the structural aspect of architectural description. Acme is an architectural description language [11] that incorporates these concepts. In this section, we give a very brief description of Acme in order to present typical ADL concepts. Acme is a second-generation ADL that focuses on the definition of a core set of concepts related to the structural aspects of software architecture in order to make it possible to exchange descriptions between independently developed ADLs. Acme is a generic

---

**Figure 4.1** Component interactions.
ADL that can be used to describe more specific concepts by specializing the core concepts.

Just like other ADLs, Acme concentrates on the design and evaluation of software architecture, emphasizing the notation used as illustrated by the following statement: “An important problem for component-based systems engineering is finding appropriate notations to describe those systems. Good notations make it possible to document component-based design clearly, reason about their properties, and automate their analysis and system generation” [11]. Acme is made of a textual and a graphical notation. In this chapter, we use a very similar graphical notation as an aid to compare the major concepts of ADLs and component models.

A system is described as a graph of components and connectors as shown in Figure 4.2.

**Components and Ports**

Components represent the computational elements and data stores of a system. There are no restrictions on the size of a component. A component can be as small as a button in a graphical user interface or as large as a Web server. For large and complex components, it is essential to be able to hide their internal structures. The interface of a component, that is, its external view, is described as a set of ports. The ports are the points of interaction between a component and its environment.

**Connectors and Roles**

Connectors represent interactions between components. Like components, connectors can be of arbitrary complexity. This notion makes it possible to represent, in a unified way, simple concepts such as a method call as well as complex and abstract connections such as an SQL connection between a

![Component, Port, Connector, Role, Attachment](image)

*Figure 4.2* Components, ports, connectors, roles, systems, and attachments in (a) a component interface, (b) a connector interface, and (c) a very simple system.
client and a database server. Indeed, making a connector a first-class entity is one of the important features of many ADLs. Because a connector can be rather complex, it is important to be able to describe its interface independently from its internal structure. The interface of a connector is defined as a set of roles. A role is to a connector what a port is to a component, that is, its interaction points. Many simple connectors are binary. For instance, a method call is a connector with two roles, the caller and the callee; a pipe has a writer and a reader; an event channel has an emitter and a receiver, and so forth. Simple arrows could be seen as suitable notation for binary connectors but this is just a matter of graphical notation.

**Systems and Attachments**

Acme makes it possible to describe components and connectors separately, but the goal is to describe a complete system as an assembly of components and connectors. The structure of a system is specified by a set of components, a set of connectors, and a set of attachments. An attachment links a component port to a connector role. Connections between components are thus modeled as a component–port–role–connector–role–component sequence. This may seem unnecessarily complex, but it is needed to model complex systems. In particular, it allows us to reason about connectors in isolation, which is possible only if connector interfaces are explicitly defined.

**Representations and Bindings**

The concepts presented above make it possible to describe how a system is made of components and connectors, giving an overall view of this system, an example of which is shown in Figure 4.3. To support top-down design of

![Figure 4.3 Representation of a system.](image-url)
software architectures, a refinement feature is needed. One or more component representations can be associated with a component interface. A component representation is made of a system and a set of bindings. Each binding links an internal port to an external port. A component can thus be decomposed hierarchically. Connectors can also be decomposed in a similar way. One important aspect of this approach is that the same formalism is used at each level of decomposition.

**Properties, Constraints, Types, and Styles**

The concepts introduced can be used to describe the structural aspect of software architecture. To improve the description of components, connectors, and systems, each ADL offers additional information to define, for instance, component behavior, protocols of interaction, and functional and extrafunctional properties. Instead of defining a fixed set of features, Acme makes it possible to annotate each entity with an arbitrary set of *properties*. Acme also includes a constraint language based on first-order predicate logic. One or more *constraints* can be attached to any architectural entity. This includes simple constraints defining the range of values an attribute can take, the number of connections accepted by a component, but also sophisticated constraints to control the topology of a system.

Acme also provides a facility for defining new types, including types of properties, types of components, types of connectors, and types of ports. Types are not expressed in terms of the structure of the elements they characterize; they also include constraints that must be satisfied by the elements. One interesting feature of Acme is that it enables us to define the style of architecture, that is, the type of system.

**Discussion**

Acme defines a relatively small set of clearly identified concepts commonly found in other ADLs and can be considered as a typical ADL. Acme does not contain all of the features found in all ADLs, but its generality and focus on structural aspects makes it particularly interesting in a comparison with typical industrial component models.

Indeed, industrial component models concentrate on a subset of the concepts described above and largely ignore advanced features such as behavior specification. Finally Acme, in the same way as other ADLs, emphasizes the notation and design of new systems but does not provide substantial help for implementation software or dealing with existing pieces of code, which
explains why it is unlikely that Acme, and other existing ADLs, will be widely used by software engineers in industry.

**JavaBeans Component Model**

The JavaBeans component model was proposed by Sun in 1997, as the first integration of the notion of a component on top of the Java language: A bean is a component. The main quality of this component model is its simplicity—at least when compared to other industrial component models: The whole specification of the JavaBeans model is a 114-page document [1], which can be compared to, say, 1,172 pages for the CCM specification [4]. The scope of this component model is quite limited and it does not scale up to large CBD. Despite its limitations this component model has been widely used and is influential and popular in the Java community; more than a dozen books have been written on it.

**Key Features**

One of the factors explaining the success of JavaBeans is the Bean Box, a tool delivered with the specification to demonstrate the feasibility of an appealing idea. The first sentence of the specification is, “The goal of the JavaBeans APIs is to define a software component model for Java, so that third party ISVs can create and ship Java components that can be composed together into applications by end users” [1]. The visual assembly or composition of existing components is at the origin of the JavaBean component model: “A JavaBean is a reusable software component that can be manipulated visually in a builder tool” [1]. The reader is invited to contrast this definition with the others provided in this book. JavaBean was designed for the construction of graphical user interfaces (GUIs). Customization of components plays an essential role in JavaBean.

An interesting aspect of this component model is the importance given to the various contexts in which a component can be considered. “First a bean must be capable of running inside a builder tool. Second, each bean must be usable at run-time within the generated application” [1]. JavaBeans is one of the few models in which the component is explicitly tailored to

1. Sun later released a second distinct component model, namely, Enterprise JavaBeans (EJB). These two models are fairly distinct and should not be confused because their names are misleading. EJB is very complex and is similar to CORBA’s CCM.
interact in two different contexts: at composition time, within the builder tool, and at execution time, in the run-time environment. This feature illustrates the fact that component models can take into account the collaboration between components and different contexts, not only execution.

**Interface of a Component**

A bean can be described from an external view as a set of ports though this terminology is not used in JavaBeans. This model defines four types of ports: methods, properties, event sources, and event sinks called listeners. In Figure 4.4, we introduce a graphical notation to ease the understanding of the model. Other component models will also be described in a similar way.

The notion of *property* can be used both to parameterize the component at composition time (as described above) or as a regular attribute (in the object orientation sense of the term) during run time. From a client point of view, three main operations are associated with a property: getting the value of a property, setting a new value, and editing the property. This last operation is the one used typically by the builder tool to support the customization of the component. A “bound property” can generate an event whenever it changes its values. A “constrained property” is a property that can be modified only if no other components reject the proposed change after a validation protocol.

The notion of method is directly bound to the notion of method in the Java programming language. From the client point of view, calling the method is the only operation associated with ports.

![Component interface](image)

**Figure 4.4** Interface of a bean component.
Event-based communication is also allowed and plays a major role in this model. *Event sources* generate events of a certain type, while *event sinks* receive events. Multicast event sources can be connected to an arbitrary number of event sinks, while unicast only supports one sink at the most. From a client point of view, the only operations supported by the event sources are the connection and disconnection of an event sink.

### Implementation of a Component

The information described in the previous section represents the interface of a component; nothing else is visible from other components. Most bean components are implemented by a simple Java object, the object being encapsulated in the component [Figure 4.5(a)]. Mapping between object methods and component ports is implicit thanks to the use of naming conventions. Sometimes more sophisticated implementations are required as suggested in Figure 4.5(b):

- **Wrapping a legacy object.** It is possible to wrap an existing object that cannot be changed and does not follow the standard naming convention. This wrapping is done through a set of explicit bindings between object methods and component ports.

- **Multiple-objects implementation.** A component may encapsulate a collection of objects collaborating in the realization of the component.

![Image](image.png)

**Figure 4.5** Implementations of bean components: (a) simple implementation and (b) more complex implementation.
These objects are shown inside the component in Figure 4.5(b). In particular, additional objects can be attached to the core functionality of the component to support its extrafunctional aspects such as the customization service; for instance, a property editor of a given property, as shown in the figure in the top-left corner, or a “customizer” as shown in the bottom part.

- **Dependency on traditional entities.** A bean object may call another object outside the component boundary and it thus depends on objects outside of it.

The implementation of a component is described here from a conceptual point of view. A component is built by grouping different objects and binding them to the components ports. From a technical point of view, this is achieved by means of nontrivial APIs.

### Components Assembly

Although assembly is one of the key features of JavaBeans, this component model does not provide any specific solution. Instead, the component model has been designed to support different ways of assembling components: “Some builder tools may operate entirely visually, allowing the direct plugging together of JavaBeans. Other builders may enable users to conveniently write Java classes that interact with and control a set of beans. Other builders may provide a simple scripting language to allow easy high-level scripting of a set of beans” [1].

The JavaBeans distribution includes a toy prototype, the Bean Box, to illustrate the notion of interactive and visual assembly of beans components. This approach has been successfully integrated in various commercial programming environments such as Sun’s Net Beans, IBM’s Visual Age, and Borland’s JBuilder. Whatever the builder used, from a conceptual point of view, a bean assembly can be seen as a graph of components as suggested in Figure 4.6(a).

Nevertheless, it is important to stress that JavaBeans neither specifies an assembly language nor the kind of connections that can be used to connect components. Obviously event sources can be directly connected to event sinks, but the range of connections available depends on the assembly tool. For instance, the Bean Box supports only connections between event sinks and methods with no arguments. Commercial builder tools allow the user to
express much more sophisticated connections and generate code behind the scene when necessary.

Although these tools have proven useful in the construction of GUIs, currently most existing systems combine component-based and traditional technology [Figure 4.6(b)]. This smooth integration has been planned: “Note that while beans are primarily targeted at builder tools they are also entirely usable by human programmers. All the key APIs, such as events, properties, and persistence, have been designed to work well both for human programmers and for builder tools” [1].

**Packaging and Deployment**

JavaBeans define a model for packaging components into archives. When the archive is loaded into a builder tool, all packaged beans will be added to the directory of available components.

Packaging is required in practice because most bean components depend on resources (e.g., icons and configuration files) and on other Java classes. To avoid duplication of items shared by different beans, it is therefore better to package all of these items together. To handle packaging issues, the JavaBeans definition includes the definition of dependency relationships between the package items.

The customization code associated with a component may be potentially quite large. For example, if a component includes a “wizard customizer” guiding a user through a series of choices, then the customization code can be more complex than the component itself. It is therefore important to be able to eliminate this code if necessary. Each package item can be marked “Design Only,” so that they can be removed in a final application.
COM, DCOM, MTS, and COM+

These technologies come from Microsoft. COM (1995) is typical of early attempts to increase program independence and allow programming language heterogeneity, but for C/C++ like languages, in a centralized context, and on Windows platforms. COM relies on binary interoperability conventions and on interfaces. DCOM extends COM with distribution, and MTS extends DCOM with persistency and transaction services. Together they constitute COM+, which is discussed here.

Interfaces and Assembly

A COM interface is seen as a C++ virtual class and takes the form of a list of data and function declarations without associated code. The binary convention is based on the C++ technique to support polymorphism: a double indirection for the access to the interface elements (see Figure 4.7). COM does not provide an assembly language; it instead provides a simple protocol that COM objects can use to dynamically discover or create objects and interfaces. All interfaces are descendants of the IUnknown interface, which defines the basic functions: QueryInterface, AddRef, and Release.

Implementation

A COM object is a piece of binary code that can be written in any programming language as long as the compiler generates code following the binary interoperability convention. These objects can be packaged in executables or DLL dynamic libraries that contain the minimum information (introspection) required for dynamic link and COM object identification.

COM supports two composition techniques: containment and aggregation. Containment is the operation by which a COM object contains other COM object(s). The outer object declares some of the inner object interfaces. The implementation of these interfaces is simply a call (delegation) to the same function of the inner object. Of course, clients see a single object (the outer one) as implementing all interfaces.

Aggregation is more complex. The outer object can expose interfaces of the inner object as if the outer object implemented them, but without the

2. Microsoft terminology is very obscure; these “definitions” come from [2].

3. An interface can be declared using C++ alone, or using the MIDL language, an extension of OSF DCE IDL; MIDL is required only for DCOM.
need to implement them (it avoids the call indirection). Unfortunately, the process of aggregation requires the source code of both the inner and outer objects to be changed so as to handle the IUnknown interface properly.

Framework

The COM framework consists of standard interfaces (IUnknown, IDispatch, etc.) and a simple run time that interprets, in collaboration with the dynamic library run time, the calls for creating COM objects, returning interface handles, and managing the reference count for releasing objects. Fortunately, the COM framework proposes a set of tools, like Visual Studio, that vastly simplifies the implementation of COM objects; indeed many COM programmers ignore the COM component model.

DCOM (Distributed COM, 1996) extends COM with distribution, based on the DCE RPC mechanism. The component model is unchanged, but the framework is extended. The MIDL language and compiler are required for interface definition; the introspection capability is much enhanced by means of type libraries.

MTS (Microsoft Transaction Server, 1997) extends DCOM with the container approach, already presented in CCM, to produce COM+ (1999). MTS introduced the distributed transactional approach and added services, similar to those of CORBA.

Life Cycle

COM and COM+ are strictly execution-time and binary component models. No life-cycle issues are explicitly supported. The .NET component model is a departure from this approach, as described later in this chapter.
CCM

The CCM is the most recent and complete component specification from OMG. It has been designed on the basis of the accumulated experience using CORBA service, JavaBeans, and EJB.

Like many component models, CCM focuses on the developer who builds applications by assembling available parts, but also explicitly the component designer, assembler, and deployer. The major goal behind the CCM specification was to provide a solution to the complexity reached by CORBA and its services: “CORBA’s flexibility gives the developer a myriad of choices, and requires a vast number of details to be specified. The complexity is simply too high to be able to do so efficiently and quickly” [4]. The choice was to define a component model generalizing the standard defined by EJB, in which the number of details to be specified and the risk of inconsistent mixtures are vastly reduced. CCM focus is “a server-side component model for building and deploying CORBA applications” [4].

One of the advantages of CCM is its effort to “integrate” many of the facets involved in software engineering. As a consequence, a software application is described in different formalisms along two dimensions: the time dimension (the life cycle, from design to deployment) and the abstract dimension (from abstractions to implementation). Altogether, this makes a rather complex specification. As of early 2002, no complete implementation of CCM had been accomplished.

For CCM, a component is “a self-contained unit of software code consisting of its own data and logic, with well-defined connections or interfaces exposed for communication. It is designed for repeated use in developing applications, either with or without customization”[4].

Interface and Assembly

The external view of a component is an extension of the traditional CORBA IDL language; a component interface is made of ports divided into these pieces (Figure 4.8):

- **Facets**, which are distinct named interfaces provided by the component for client interaction;

• **Receptacles**, which are named connection points that describe the component’s ability to use a reference supplied by some external agent;

• **Event sources**, which are named connection points that emit events of a specified type to one or more interested event consumers, or to an event channel;

• **Event sinks**, which are named connection points into which events of a specified type may be pushed;

• **Attributes** are named values, exposed through accessor and mutator operations. Attributes are primarily intended for use in component configuration, although they may be used in a variety of other ways [4].

CCM considers ports as named and typed *variables* (most models consider only interfaces, i.e., a type), thus different facets of the same component can have the same type. Components are typed and can inherit (simple inheritance) from other components (i.e., it inherits its interface).

_Homes_ are component factories that manage a component instance life cycle: creation, initialization, destruction, and retrieval of (persistent) component instances. Homes are also typed and can inherit from other home types. A home type can manage only one component type, but a component type can have different home types. Homes are defined independently from components, which allow component life-cycle management to be changed without changing the component definition.
CCM uses the term *navigation* to describe the framework operations that can be called by a component to dynamically discover and connect to other components and ports.

**Assemblies**

CCM simply defines the concept of connection as “an object reference”; thus CCM, like all other industrial component models, does not provide a connector concept. Nevertheless, components are connected by linking facets to receptacles and event sources to event sinks. Connections are binaries and oriented, but the same port can handle multiple connections.

Connections can be explicitly described (in the assembly descriptor, an XML file) and established by the CCM framework at initialization. A component assembly describes the initial configuration of the application (i.e., which components make up the assembly), how those components are partitioned, and how they are connected to each other. The assembly does not address architecture evolution during execution.

It is interesting to note that the CCM specification explicitly refers to the Rapide ADL: “The provides and uses statements in this submission are similar to the Interface Connection Architecture implemented in Rapide [12] and discussed in [13]. The Rapide Interface Connection Architecture applies provides and requires statements to individual functions in a class declaration … the difference being that we specify dependencies with respect to interfaces rather that individual methods” [4].

**Implementation of a Component**

The implementation of a component is a set of *segments* or executors. Segments are executable code written in any programming language, implementing at least one port. Segments are loaded only when required. No (segment) composition operators are defined. Nothing prevents a segment from calling conventional programs; conversely, the concept of “equivalent interface” was defined to turn a program into a component and to make a component appear as a classic program.

Because implementation is always a set of executable code, the CCM model is not hierarchical. The same applies to homes which are directly bound to executors.

CCM proposes a Component Implementation Definition Language (CIDL) that describes the segments, the associated home executor, the type
of storage, and the class of container (see discussion of containers). Different implementations can be associated with the same (abstract) component.

**Framework: The Container Approach**

Like CORBA, MTS, and EJB, CCM emphasizes the fact that many services can be made available to components without having to change that component’s source code. This approach increases component reusability, reduces complexity, and significantly improves maintainability. These reasons are at the origin of the many CORBA services; unfortunately, the multiplication of services has (re)introduced a level of complexity incompatible with widespread use of that technology.

Like EJB, CORBA components use a container to implement component access to system services using common design patterns gleaned from experience in building business applications using object technology and CORBA services. Containers are defined in terms of how they use the underlying CORBA infrastructure and thus are capable of handling services like transactions, security, events, persistence, life-cycle services, and so on. Components are free to use CORBA services directly (component-managed service), but CCM emphasizes container-managed service (i.e., the container managing services automatically); the component code itself ignores the associated services. To do so, the container intercepts communications between components and calls, if needed, framework services. In Figure 4.9, the container is represented as the part of the run-time framework located between the ports and the component implementation. Components may have to implement callback operations defined by the framework if they need to manage their own persistent state.

![Figure 4.9 CCM run-time framework.](container)
Life Cycle

A package descriptor is a file, in XML, describing a set of files and descriptors, including the component assembly descriptor. CMM does not define the set of formalisms from which this XML file is produced; thus the CCM specification does not describe the actual packaging and deployment. Vendors, in the future, should provide assembly, packaging, and deployment design tools, whose purpose will be to help users in the design and specification of these topics, as well as deployment tools, which, by interpreting the information found in the packaging, assembly, and component implementation descriptors, will perform actual deployment and assembly.

CCM is the best effort to date to gather the advances made in different fields, to include a wide spectrum of life-cycle activities, while still claiming efficiency and heterogeneity capabilities, but the whole does not provide the feeling of being as “simple” as claimed.

.NET Component Model

.NET [5], the latest component model from Microsoft, represents a discontinuity—it no longer relies on COM, because binary interoperability is too limited. .NET relies instead on language interoperability and introspection. To do so, .NET defines an internal language Microsoft Intermediate Language (MSIL), which is very similar to Java Byte Code and its interpreter with introspection capabilities: the Common Language Runtime (CLR), which is very similar to a Java Virtual Machine.

Interfaces and Assembly

.NET represents the programming language approach for component programming. It means that the program contains the information related to the relationships with other “components,” and that the compiler is responsible for generating the information needed at execution. This (proprietary) approach contrasts with the Object Management Group (OMG) (open) approach where separate formalisms (and files) are used to indicate component-related information, with languages and compilers being unchanged.

What most resembles a component is an assembly. The manifest is the component descriptor. It gathers in a single place all information about an assembly: exported and imported methods and events, code, metadata, and resources.
Because of the programming language approach, the corresponding programming language, C#, which is very similar to Java, includes some features of a component model: (first-class) events and extensible metadata information. The compiler not only produces MSIL byte code but also generates, in the *manifest*, the interface description of the component (called assembly), in the form of a list of import and export types. There is no explicit concept of connection but rather the traditional list of imported and exported resources. .NET relies on a specific dynamic linker to realize the connections, during execution, between the provided and required resources.

**Implementation**

A component (assembly) consists of *modules*, which are traditional executable files or dynamic link libraries (DLLs). Following the programming language approach, the list of modules composing an assembly is provided in the compiler command line when compiling the main module. The compiler thus generates the manifest in the same file as the main module executable. Modules are loaded only when required. Modules cannot be assemblies, thus the .NET model is not hierarchical. Figure 4.10 shows the .NET interface and component implementation.

**Framework**

.NET relies on the traditional programming approach in which the framework is seen as the language run-time support: “The .NET Runtime is designed to give first-class support for modern component-based programming—directly in the Runtime” [6]. Extrafunctional aspects such as distribution, security, confidentiality, and version control are delegated at

![Figure 4.10](image)

Figure 4.10 .NET component (a) interface and (b) implementation.
execution to the operating system and loader (see the life-cycle section below). Transaction control relies on Microsoft Transaction Server (MTS).

**Life Cycle**

Unlike when using traditional DLLs, the .NET model includes visibility control, which allows assemblies (and their modules) to be local to an application, and thus different DLLs with the same name can run simultaneously. Further, each assembly keeps track of versioning information about itself and about the assemblies it depends on, provided either in the form of attributes in the code source or as command-line switches when building the manifest.

Version control is delegated to the dynamic loader, which selects the “right” version, local or distant, based on the assembly’s version information and on a set of default rules. Both at the machine-wide level and at each application level, the default rules can be altered using XML configuration files.

These features significantly improve application packaging and deployment control (with respect to traditional Windows application deployment). The early life-cycle phases (design, analysis) appear not to have received corresponding attention to date.

**The OSGI Component Model**

The OSGI was founded in 1999 with the mission of creating “open specifications for the delivery of multiple services over wide area networks to local networks and devices” [7]. The OSGI emphasis is on a lightweight framework that can be executed in low-memory devices. Actually, the OSGI targets products such as set-top boxes, cable modems, routers, consumer electronics, and so on. The OSGI relies on Java to ensure portability on different hardware. An important characteristic of this technology is that it has been tailored to support dynamic evolution of the system architecture. Components can be downloaded, updated, and removed dynamically, without even stopping the system. Moreover, the OSGI allows for remote administration of the system via the network.

**Two Levels of Components**

The OSGI is based on two main concepts that can be interpreted as components: bundles and services. “Developers should design an application as a set of bundles that contain services, with each service implementing a segment of the overall functionality. These bundles are then downloaded on
demand” [7]. The term *component* is used alternatively for each concept in the specification.

While some authors define components as a unit of composition and as a unit of deployment, these two concepts can be distinguished in the OSGI. A *service* is a unit of composition; a *system* is a set of cooperating services that are connected. A *bundle* is a unit of deployment that groups a set of services that can be deployed as a unit. The following discussion centers on bundle components because the OSGI provides many more features at this level than at the level of service components. Indeed, a service component is merely defined as an implementation and as a set of interfaces.

**Interface of a Bundle Component**

A bundle packages a set of software entities that collectively form a piece of software. These entities may depend on entities packaged in other bundles, therefore creating dependencies between bundles. The OSGI manages these dependencies. From an external point of view, the interface of a bundle could be represented as shown in Figure 4.11. As suggested in the figure, a bundle uses three kinds of ports to express its interactions, (1) with traditional technology, (2) with others components, and (3) with the run-time environment.

First, the OSGI clearly recognizes the importance of handling traditional technology. In particular, a bundle may require and provide one or more Java packages. It declares this information statically by means of appropriate ports depicted on the topside of the interface.

Second, bundles manage dynamic connections between services. At any time, and for any reason, a bundle may display or remove a service interface. Similarly, at any time, and for any reason, a bundle may require or release the use of a service interface. Service interfaces can therefore be attached and detached dynamically on the left and right sides.

Third, bundles can interact with the run-time environment through the ports depicted on the bottom side. Bundles may listen to events published by the framework such as the insertion of a new component in a system or the publication of a new service. In this way bundles can take appropriate action as the architecture evolves.

**Assembly of Bundle Components**

Most ADLs and component models are based on the notion of static assembly: The set of participating components is known and components are connected by a human to form a valid system. The OSGI is based on a radically
Figure 4.11 (a) Interface of a bundle component and (b) dynamic connection.
different approach. A system is an evolving set of bundle components. Components can connect to each other dynamically based on their own decisions. Components cannot assume that the interfaces they use will be available at all times since the component displaying these interfaces may be uninstalled or may decide to remove this interface from their ports. This means that components may have some knowledge about how to connect and disconnect.

The process of dynamic connection is illustrated in Figure 4.12. When a bundle component publishes a service interface, it can attach to it a set of properties describing its characteristics. When a component requires an interface for its use, it will select one via a query expression based on these properties. The destination of a connection is never given explicitly. The result of the query depends on the actual state of the system. This contrasts with the traditional approach in which each connection statically links ports of identified components. At a given point in time, an OSGI system may look like that shown in Figure 4.12.

The set of connections between components (in the middle of the figure) evolves dynamically. This flexibility also has its counterpart. Once a connection is established with a bundle, there is no guarantee that the service will remain available. Each bundle component must listen to events generated by the OSGI run-time environment and must take appropriate action as the system evolves. These connections are displayed in the bottom of the figure. On the top of the figure, the connections between Java packages are shown. The framework automatically creates these connections on the basis

![Figure 4.12 Implementation of a bundle component.](image_url)
of the information provided in the bundle interface. Various bundles may provide the same package but in different versions. In this case, the framework will ensure that only one will be selected on the basis of the preference expressed by the bundles requiring the packages and on a set of default rules specified by the OSGI. A bundle cannot be activated until all of the packages it imports are present in a system.

**Implementation of a Bundle Component**

The section above described a bundle from an external point of view. Actually, from a concrete point of view, a bundle is represented as a JAR archive containing (1) service components, (2) Java packages, and (3) other resources files such as configuration files and images (Figure 4.12).

The set of entities contained in a bundle component can be connected to form a running system. If a bundle provides an “activator” object, the framework will call this object to activate and passivate the bundle. The entities contained in the bundle are not visible from outside as long as the bundle does not export them. They are independent from the rest of the system as long as they do not require a service provided by another bundle.

One wonders then if every service component must handle the complexity of dealing with the dynamic apparition and retirement of external services. The OSGI specification does not provide any information with respect to this question, but in some situations it is possible to regroup this behavior in a piece of code providing a default value when a connection cannot be resolved or when a resolved connection later becomes unavailable. In this case, communication with the outside of the bundle can be made totally transparent to the internal components; the connection manager can be seen as a container providing a connection service. This method does not prevent individual components from registering directly with framework events if they want to achieve a specific behavior.

**Summary**

The following briefly summarizes the main differences among the presented component models, along our four dimensions.

**Interface**

Although interfaces appear to be similar, a number of differences exist in the concept and in the way they are defined. In the programming language
approach, JavaBeans and .NET, the structure is a language one: virtual
classes (.NET) or interfaces (Java), and the compiler are in charge of generat-
ing the required information. In EJB and COM, an interface corresponds to
a set of methods; it is a type. In CCM, ports are typed variables but in all
cases interfaces are described in a separate formalism.

In ADLs, interfaces and ports are carefully designed but abstract con-
cepts. The connector, as a first-class concept, is typical of ADLs. Connectors
may have ports, are typed, can inherit, and can be associated with potentially
complex behavior. Connections and necessary connectors do not yet exist in
today’s industrial component models.

Assembly

JavaBeans is original in that assembly is a step directly supported by the
framework. Beans can include code that helps in customizing and assembling
components. EJB and COM do not provide specific features. CCM makes
explicit and automates the initial connections, .NET relies on the dynamic
loader, and the OSGI focuses on the dynamic discovery of services.

In ADLs, assemblies (often called configurations) and the analysis of
their properties are important facets, but assembly is seen from a static point
of view, during design.

Implementation

Surprisingly, all models are flat, but most models consider a component as a
set of executables: segments (CCM), modules (.NET), and classes. From this
point of view, components provide a higher level of structuring and packag-
ing than programming languages. In all models, components are seen as a
composition unit; in many cases they are also seen as the packaging and
deployment unit.

In all models, component implementations can call the usual pro-
grams, which are not components. In ADLs, all models are hierarchical, but
the relationship with executables is seldom considered.

Framework

In the language approach, the operating system and loaders perform most of
the work in a transparent way. In JavaBeans and COM no added formalism
is required, but added (extrafunctional) services are not provided either. In
.NET the loader has been extended to provide distribution, security,
deployment, and versioning services, but attributes and configuration files are required. In MTS, EJB, and CCM, the container concepts make explicit a set of services, described in a specific formalism, and supported at execution by the framework.

In most models the infrastructure is limited to the run time, JavaBeans shows that the infrastructure can provide support to the assembly phase, but the same idea can be extended to all phases: design, deployment, and so on. In ADLs, this aspect is not explicitly addressed. Extrafunctional properties, such as distribution and transaction, and the way they can be implemented are not usually considered important.

**Conclusion**

The major differences between ADLs and component technologies are seen in their focus. Component technology addresses primarily execution and technological issues, including interoperability, heterogeneity, distribution, efficiency, fast life cycle, deployment, and multiprogrammer development. In component technologies, pragmatism and realism require that an application is always a mixture of components (which satisfy the model and protocols) and of the usual programs (classes and procedures). The component support system manages the former and simply ignores the latter. Component technology often makes the distinction between the component model specification (e.g., CORBA) and the implementation of the standards (many companies, such as Visigenic and Iona, market their CORBA implementations), the in-house developed component, and the components available in the marketplace. ADLs do not make these distinctions. Component technology emphasizes the relationships with operating systems and the run-time support, not ADLs. Conversely, component technologies do not have a hierarchical model, propose very limited formalisms, do not propose any connector concept, have no design formalism, and do not propose any analysis tool. To say the least, there is room for improvement.

Component-based technologies and ADLs are the logical results of evolution from the previous generation of software engineering technology.

---

5. Which makes difficult, or useless, many of the analyses ADL systems can perform.

6. The OSGI is one of the very few exceptions, see the section on the OSGI component technology.
Thus drawing a definitive boundary between them is difficult and irrelevant. Trying to contrast ADLs and component-based technology is also irrelevant since, initially, the former focused on early life-cycle phases (analysis and design) while the latter focused on the execution phase and its technological issues. Their natural fate is to be merged to provide next-generation software engineering environments, tools, techniques, and methods.

References


