Component Models and Component Services: Concepts and Principles

Rainer Weinreich

Johannes Sametinger

Section	I	Authors	Rainer Weinreich, Johannes Sametinger
Chapter	6	E-mail Addresses	weinreich@swe.uni-linz.ac.at, sametinger@acm.org
Pages		Phone Number	+43 70 2468 - 9434, +43 70 2468 - 9435
Date Complete	**12/03/00 3:00 PM RW**	Dates Reviewed	GH (3/21/00 8:51 PM) GH(4/2/00 10:39 AM); BC (4/4/00 10:02 PM); BC (4/5/00 12:05 PM) BC (4/6/00 4:11 PM), GH (8/16/00 2:13 PM) BC (17Aug2000@4:08 CDT PM), BC (20Oct2000 5:50 PM), GH (10/23/00 10:38 PM), GH (10/25/00 9:34 PM), BC (24Nov2000 6:05 PM), BC (25Nov2000 12:05 PM), GH (11/26/00 7:46 PM) BC (28Nov2000 12:58 PM)

Introduction

In this chapter, we describe, comment, and motivate discussions upon basic concepts and principles of component models and component services. We argue that component systems have been used at a coarse-grained level for decades. Discussions and disagreements concerning the term component have appeared because industry and academia have tried to develop technologies for smaller, fine-grained components.

Early component systems

Operating systems are among the first successful component systems (Szyperski 1997). We expect readers to be familiar with at least one operating system; and, thus, we will use operating systems to illustrate the basic ideas behind component models and component model implementations. Operating systems provide an execution environment for software applications. In particular, operating systems present an abstraction of the underlying hardware to applications and regulate their shared access to various resources. They also provide basic services, such as memory management, file management, inter-process communication, process synchronization, and security. Without operating systems, each individual application would have to implement such general functionality. The availability of a wide range of services simplifies application development. Naturally, interfaces have to be defined to let applications use these services. These interfaces are called application programming interfaces (APIs).

To make an analogy, operating systems are component model implementations (see Chapter 4) for applications, which may be viewed as coarse-grained components. Once a

component model implementation is developed and documented, multiple vendors can develop applications that use the low-level services provided by the component model implementation. At the granularity of an application, there is a functioning component market. You can buy applications from different vendors and use them together on a single computer; the applications all adhere to the "standard" defined by an operating system. Often these standards are part of a particular operating system implementation; sometimes they are specified more explicitly and made available to the public, such as the UNIX 98 standard promoted by the Open Group (www.opengroup.org) and the Linux Standard Base (www.linuxbase.org).

Deficiencies of early component systems

Continuing our analogy, we see several shortcomings. Components at the application level can be used, but they do not sufficiently enable widespread software reuse. The lack of reuse results because applications are too coarse-grained, because applications lack composition support, and because operating systems lack domain-specific standards:

1. *Lack of granularity*. Applications are too coarse-grained to improve software reuse. Application developers are often required to design and fully implement common functionality that any application would have. Component-based software engineering (CBSE) seeks to factor out these commonalties into either services provided by the component model implementation or components that could be purchased and integrated into a component infrastructure. A central concept of CBSE is to develop technologies for smaller, fine-grained components and enable a similar degree of reuse on the level of application parts as had been possible at the application level.

2. *Lack of composition support*. While applications have long been units of independent deployment, there has typically been no support for composition, including third-party composition (recall the definition of the term, software component, in Chapter 4). In fact, operating systems ensure that applications execute in complete isolation from each other. Mechanisms such as inter-process communication have been introduced to enable data exchange among applications, but application interfaces are often poorly specified and composition standards are missing. While applications are deployed in the operating system and use its services, they are rarely units of composition.

3. *Lack of domain-specific standards.* The services provided by an operating system are too general to support specific application domains. For example, a simulation system needs other services and APIs than a process control system or a telecommunication application.

The goal of CBSE is to develop software systems by composing reusable components at a finer level of granularity than applications. Naturally, these fine-grained components need standards for interaction and composition, as well as standardized infrastructures and services. The challenge of CBSE is to define component models with such standards

and to provide associated component model implementations to enable components and component infrastructures to be designed, implemented and deployed.

Components and objects

CBSE is commonly considered the next step after object-oriented programming. Thus, it is not surprising that components are often related to objects and sometimes the term component is simply used as a synonym for object. However, the concepts of components and objects are independent although most component models are based on object-oriented concepts. To avoid further confusion we briefly characterize objects and components and outline their differences.

Objects are entities that encapsulate state and behavior and have a unique identity. The behavior and structure of objects is defined by classes. A class serves multiple purposes. First, it implements the concept of an abstract data type (ADT) and provides an abstract description of the behavior of its objects. Class names are often used as type names in strongly typed systems. Second, a class provides the implementation of object behavior. Third, a class is used for creating objects, that is, instances of the class.

Nearly all modern component models are based on the object-oriented programming paradigm. The basic premise of object-orientation is to construct programs from sets of interacting and collaborating objects; this does not change with component-based approaches. Components are similar to classes. Like classes, components define object behavior and are used for creating objects. Objects created by means of components are called component instances. Both components and classes make their implemented functionality available through abstract behavior descriptions, called interfaces.

Unlike classes, the implementation of a component generally is completely hidden and sometimes only available in binary form. Internally, a component may be implemented by a single class, by multiple classes, or even by traditional procedures in a non-object-oriented programming language. Unlike classes, component names may not be used as type names. Instead, the concept of type (interface) and the concept of implementation are completely separated. Finally, the most important distinction is that software components conform to the standards defined by a component model.

Component models

A component model defines a set of standards for component implementation, naming, interoperability, customization, composition, evolution, and deployment. A component model also defines standards for an associated component model implementation, the dedicated set of executable software entities required to support the execution of components that conform to the model.

There are numerous component models currently available. The main competing component models today are OMG's CORBA Component Model (CCM), Microsoft's (D)COM/COM+ family, and SUN Microsystems' JavaBeans and Enterprise JavaBeans. We need generally accepted component models to create a global component marketplace. It is not necessary to agree on one standard; at the same time, there should

not be too many standards either. The market share of a particular standard has to be large enough to make the development of complying components worthwhile (Szyperski, 1997). In this chapter, we comment on important elements constituting a component model.

Elements of a component model

In a global software component marketplace, components are independently deployed and subject to third-party composition. Such a marketplace requires standards. Standards for communication and data exchange among components from different vendors are rather obvious. Such an interoperability standard — sometimes called wiring or connection standard — is a central element of a component model. Other basic elements of a component model are standards for interfaces, naming, evolution, packaging, customization, and composition (see Table 1).

Table 1: Basic elements of a component model

Standards for	Description
Interfaces	Specification of component behavior and properties; definition of Interface Description Languages (IDL).
Naming	Global unique names for interfaces and components.
Metadata	Information about components, interfaces, and their relationships; APIs to services providing such information.
Interoperability	Communication and data exchange between components from different vendors, implemented in different languages.
Customization	Interfaces for customizing components, needed by customization tools.
Composition	Interfaces and rules for combining components to create larger structures and for substituting and adding components to existing structures.
Evolution Support	Rules and services for replacing components or interfaces by newer versions.
Packaging and Deployment	Packaging implementation and resources needed for installing and configuring a component.

A component model can also have specialized standards for describing domain-specific features required for certain applications. For example, the composition of components in

domains with concurrent activities requires appropriate standardized threading models and synchronization mechanisms. An open distributed processing system requires standards for remote method invocation and security. Three-tiered business applications needs standardized transaction services and database APIs. Finally, a component model for compound documents (like OLE) needs to specify part and container relationships and interfaces. Domain-specific component models offer such special functionality in the component model implementation.

Interfaces, Contracts, and Interface Definition Languages

The main purpose of software components is software reuse. The two main types of software reuse are white-box reuse and black-box reuse. *White-box reuse* means that the source of a software component is fully available and can be studied, reused, adapted, or even modified. White-box reuse plays a major role in some object-oriented frameworks, which rely heavily on inheritance for reusing software implementations (see Gamma et al., 1995). The problem with white-box reuse is that component consumers might depend on the internals of a component and thus be affected adversely if the internals change.

Black-box reuse is based on the principle of information hiding (Parnas, 1972), which states that a component should reveal as little about its inner workings as possible. Users of a component may only rely on interfaces, which are descriptions or specifications of component behavior. By using interfaces, components may be changed internally so long as they continue to satisfy the responsibilities defined by their interfaces. Changes to interfaces are made explicit and tools, such as compilers, can statically verify compatibility with client components.

An interface is not a constituent part of a component, but rather serves as a *contract* between a component and its clients. An interface specifies the services a client may request from a component and which services a component has to provide. Additionally, an interface may include constraints on the usage of these services that have to be considered by both the component and its clients.

Interface specifications are a central element of a component model. A component model defines how a component's behavior is described by means of interfaces, other (non-functional) specifications, and appropriate documentation. A component model defines the elements that may constitute an interface as well as the semantic meaning of these elements. Well-known elements of an interface are:

- Names of semantically related operations
- Their parameters
- Valid parameter types

Interfaces may also include exceptions that may be raised, preconditions and postconditions that have to be met when using individual operations, and even partial specifications of the expected behavior of a component implementing the interface (Büchi and Weck, 1999). Many component models have an interface definition language (IDL) for describing interfaces and their elements using an implementation-independent notation.

The component model may define a set of specific interfaces that need to be implemented by components that conform to that model. In general, these interfaces will be used by the component model implementation to provide dedicated services expected by the components, such as transactions or security.

Naming

A global marketplace requires uniquely identifiable components and interfaces. Name clashes (when two different components are mistakenly assigned the same name) have to be avoided or at least should be highly unlikely. Thus, a standardized naming schema is a necessary part of a component model. The two main approaches to such a naming schema are unique identifiers and hierarchical namespaces.

- *Unique IDs.* Unique identifiers are generated by dedicated tools (e.g., compilers), which use a combination of specific data to guarantee the uniqueness of each generated identifier. An example of unique IDs are Global Unique IDs (GUIDs), which are used by Microsoft's COM/DCOM/COM+ family. A GUID is a 128-bit number that combines a location identifier (e.g., the address of an Ethernet card), the time of creation, and a randomly generated number. GUIDs were introduced by OSF/DCE, where they were called *Universally Unique IDs* (UUIDs).

- *Hierarchical name spaces.* Hierarchical namespaces are guaranteed to be unique if the top-level names are uniquely registered with a global naming authority. Most Java-based component models use hierarchical namespaces (although there is no global naming authority). SUN Microsystems advises manufacturers to adhere to a registered Internet domain name as the root name for their components (Gosling et. al., 1996, p.125f).

Metadata

Metadata is information about interfaces, components, and their relationships. Such information provides the basis for scripting and remote method invocation and can be used by composition tools and reflective programs (see Maes [1987] and Kiczales [1991] for a good treatment of reflection). A component model must specify how metadata is described and how it can be obtained. Component model implementations must provide dedicated services allowing the metadata to be retrieved. There are many ways in which metadata can be provided, such as interface and implementation repositories of CORBA-based systems, type libraries in COM-based systems, and introspection in Java-based systems.

Interoperability

Software component composition is possible only if components from different vendors can be connected and are able to exchange data and share control through well-defined communication channels. Component interoperability or wiring standards are thus a central element of any component model.

An operating system executes applications in separate and isolated process address spaces, but communicating components may reside in the same process address space. If the component model allows the implementation of components in different programming languages, calling conventions must be standardized at the binary level to ensure interoperability of these components. Even if component implementations share the same language, the binary layout of interfaces and parameter types may still be different. Interoperability of components within a process address space is possible if the component model defines the binary interface structure and calling conventions.

A component model may also support communication of components across processes on the same computer or over the network. Remote interoperability is based on remote method calls (RMCs), an extension of the concept of remote procedure calls (RPCs) introduced by Birell and Nelson (1984). An RMC consists of a client invoking a method of a remote server. To the client, an RMC appears similar to a local method invocation because the client actually invokes a method of a local proxy object that offers the same interface as the remote component. The proxy transforms the method invocation (including parameters) into a linearized network format (a process called *marshaling*) and sends the data to a corresponding *stub* object on the remote machine. The stub receives the data, reconstructs the invocation (un-marshaling) and forwards the invocation request locally to the component instance for which it was intended. Proxy and stub are called stub/skeleton in CORBA-based systems.

A component model supports distributed components by defining common data representations and invocation semantics. Often component models also standardize the network protocols used for communicating among different components based on the same component model. Examples for remote method specifications are the Simple Object Access Protocol (SOAP) for Windows .NET platforms (msdn.microsoft.com/soap), Remote Method Invocation (RMI) for Java-based platforms (java.sun.com/products/jdk/rmi), and Internet Inter-Orb Protocol (IIOP) for CORBA-based systems (www.omg.org). SOAP, for example, uses the eXtensible Markup Language (XML) for data encoding and the HyperText Transfer Protocol (HTTP) as standard transport protocol.

Interactive development environments (IDEs) supporting a particular component model usually provide dedicated tools for automatically generating proxies and stubs for remote communication. Some component model implementations support on-the-fly proxy and stub generation (or generic proxies) based on metadata from component interfaces.

Different component models may support incompatible remote method specifications. A component model should explicitly define how to “bridge” communication among implementations of different component models. For example, the CORBA Specification (OMG, 1999) defines how to access Microsoft COM objects from CORBA environments and vice versa.

Customization

Interoperability standards and metadata about components and interfaces provide the basis for component customization and composition. We define component customization as *the ability for a consumer to adapt a component prior to its installation or use*. Since components are treated in black-box fashion, revealing as little as possible of their implementation, components can only be customized using clearly defined customization interfaces. A customization interface enables customization and deployment tools to modify simple properties, or even complex behavior by providing instances of other components as parameters to customization functions. Customization tools may learn about the customization interfaces of components using metadata services.

Composition

Component *composition* or *assembly* is the combination of two or more software components yielding a new component behavior. A component composition standard supports the creation of a larger structure by connecting components and the insertion or substitution of components within an existing structure (see Chapter 4). Such an existing structure is a component infrastructure, sometimes called component framework.

The two basic types of component interactions are client/server and publish/subscribe. Components may act as clients, requesting information from, or method invocations of, other components. A component may register itself with another component or a dedicated service and receive notifications of interesting events. The component model must define how to design interfaces to support such composition. Metadata about imported and exported interfaces of a component is required for composition tools and languages.

Various approaches to component composition at different levels of abstraction have been identified (Weinreich, 1997). Components may be connected using all-purpose programming languages, scripting or glue languages, visual programming or composition tools, or component infrastructures. Glue languages, such as VisualBasic, JavaScript and TCL, support component composition at a higher level of abstraction than all-purpose programming languages, such as C++ and Java. Composition through visual programming further raises the level of abstraction, but there are drawbacks of visual approaches, such as the lack of density and structure of graphical representations and the needed extra effort for graphic editing and layout operations (Petre, 1995).

The disadvantage of composition languages and tools is that the glue code has to be written or graphically specified from inception. Maximum reuse is achieved with component infrastructures designed for a specific domain, where the interaction among component instances is already predefined. Composition with a component infrastructure is a matter of inserting and substituting components conforming to the interaction standards defined by the component framework. Interaction standards specify which interfaces participating components have to implement along with rules governing component interaction.

Component infrastructures or frameworks enable not only the reuse of individual components but also of an entire design. For example, Weinreich (1997) describes a trader-based component infrastructure for graphic editors; Szyperski and Pfister (1999) describe a component infrastructure that supports compound documents; Praehofer et al. (1999) describe a component infrastructure, based on JavaBeans, for simulation systems. Only a well-designed component infrastructure enables the effective and efficient assembly of components.

Evolution Support

Component-based systems require support for system evolution. Components acting as a server for other components might have to be replaced by newer versions providing new or improved functionality. A new version may not only have a different implementation but may also provide modified or new interfaces. Existing clients of such components ideally should not be affected or should be affected as little as possible. In addition, old versions and new versions of a component might have to co-exist in the same system. Rules and standards for component evolution and versioning are thus an important part of a component model.

Packaging and Deployment

Widely accepted component model standards, as well as high-bandwidth Internet connections, will change the deployment of what is now called shrink-wrapped software. It will become superfluous to bundle big software systems and its documentation to sell off-the-shelf. In addition to well-defined component model implementations, only small components will be needed to construct applications. Fast Internet connections will allow component consumers to conveniently download packaged components with documentation to develop comprehensive software systems.

A component model must describe how components are packaged so they can be independently deployed. A component is deployed, that is, installed and configured, in a component infrastructure. Thus, the component must be packaged with anything that the component producer expects will not exist in the component infrastructure. This may include the program code, configuration data, other components, and additional resources. A deployment description provides information about the contents of a package (or of a number of related packages) and other information that is necessary for the deployment process. This description is analyzed by the target component infrastructure and used for installing and configuring a component properly.

The deployment standard specifies structure and semantics for deployment descriptions and it may also define the format of packages. A component model may also define processes for deployment, including component registration.

Component Model Implementations and Services

An important part of a component model is the standardization of the run-time environment to support the execution of components. This includes the specification of interfaces to both general and more domain-specific run-time services. General services

to support object-based component systems include object creation, life-cycle management, object-persistence support, and licensing. Component models for distributed systems additionally have to define services for:

- Other forms of communication, such as message queues
- Remote event-based notification
- Locating remote services
- Security

Component models supporting the construction of multi-tiered information systems may specify data access APIs and services for transaction management and load balancing.

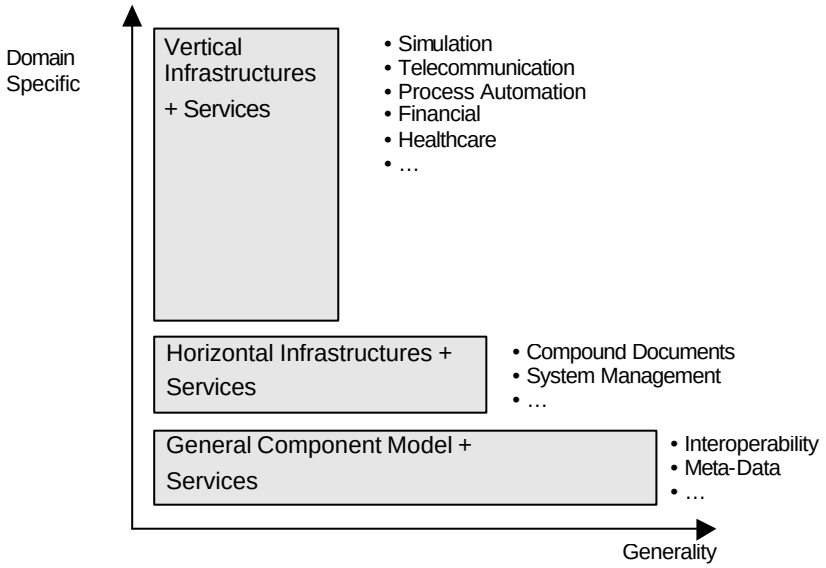

Figure 1. From general to domain-specific standards

Typically, a component-based design will reflect the standardization process from general to more domain-specific services (see Figure 1). For example, a general component model for distributed systems may form the base on top of which additional, more domain-specific, component infrastructures and services may be defined. Horizontal services and infrastructures provide additional functionality across multiple domains. Typical examples of such services include user interface management services, such as compound documents, and system management services. Vertical services and infrastructures support a particular domain. Examples are financial, healthcare and telecommunication services.

An example for such a family of standards that is built on a general component model is the object management architecture (OMA). The OMA is defined by the Object Management Group (OMG), a non-profit organization with about 800 industrial and academic members (www.omg.org). At the heart of this model is the common object request broker architecture (CORBA), an interoperability standard for distributed object-based applications supporting various implementation languages. CORBAservices and CORBAfacilities are specifications built upon CORBA. CORBAservices are a standard for general services of distributed object-systems while CORBAfacilities standardizes horizontal services. The most specialized standards of the OMA are vertical services for various application domains.

Other well-known component models, such as Microsoft's COM family and Sun's JavaBeans, define similar services that are useful for systems in multiple domains. All major component model implementation vendors are also developing domain-specific interaction and composition standards.

Conclusion

We introduced the concept of component models using an analogy comparing operating systems with component model implementations. Operating systems provide the basic component model for applications through interfaces and services and provide integration and communication mechanisms. CBSE makes it possible to consider fine-grained components to enable more flexible component composition. CBSE will enhance the level of software reuse and provide necessary component models for establishing component marketplaces at this level. Standardized component models are necessary to realize this vision.

We have presented the basic concepts and principles of component models and component model implementations. Component models define standards for interfaces, naming, interoperability, customization, composition, evolution, packaging and deployment. Additionally, specifications of run-time environments and services are needed to standardize component models. Typically, component model implementations exist on top of an operating system. However, some operating systems, such as MS-Windows™, have already begun to incorporate component model implementations. Eventually, operating systems may directly serve as component model implementations for CBSE.

References

A.D. Birell and B.J. Nelson, "Implementing Remote Procedure Calls", *ACM Transactions on Computer Syst*ems, Vol. 2, No. 1, February, 1984, pp. 39-59.

M. Büchi and W. Weck, "The Greybox Approach: When Blackbox Specifications Hide Too Much", Technical Report No 297, Turku Centre for Computer Science, ISBN 952-12-0508-3.

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

J. Gosling, B. Joy, G. Steele, *The Java Language Specification*, Addison-Wesley, 1996.

G.Kiczales, J.deRivieres, D.G.Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

Linux Standard Base, "Standardizing The Penguin", http://www.linuxbase.org/, 2000.

Pettie Maes, *Computational Reflection*, PhD Thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Bruessel, Belgium, January 1987.

Object Management Group, "A Discussion of the Object Management Architecture", http://www.omg.org, January, 1997.

Object Management Group, "The Common Object Request Broker: Architecture and Specification", Rev. 2.3.1, http://www.omg.org, October, 1999.

D. L. Parnas, "A technique for software module specification with examples. *Communications of the ACM,* Vol. 15, No. 5, May, 1972, pp. 330-336.

M. Petre, "Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming", *Communications of the ACM,* Vol. 38, No. 6, June, 1995, pp. 33-44.

H. Praehofer, J. Sametinger, A. Stritzinger, "Concepts and Architecture of a Simulation Framework based on the JavaBeans Component Model", *Journal of Future Generation Computer Systems*, Special Issue on Web-based Simulation, Elsevier Science, Vol. 16, 2000.

C. Szyperski, *Component Software: Beyond Object-Oriented Programming,* Addison-Wesley/ACM Press, 1997.

C. Szyperski, C. Pfister, "BlackBox: A Component Framework for Compound User Interfaces", *Implementing Application Frameworks: Object-Oriented Application Frameworks at Work* (edited by M. Fayad, D.C. Schmidt, R. Johnson), John Wiley & Sons, Inc., New York, 1999.

R. Weinreich, A Component Framework for Direct Manipulation Editors, *Proceedings TOOLS-25*, Melbourne, Australia, IEEE Computer Society Press, November, 1997, pp. 93-101.

Side-Bars

Operating systems provide
- An abstraction of the underlying hardware (infrastructure)
- An execution environment, and
- Basic services

for applications.

Component models define standards for
Naming, metadata, component behavior specification, component implementation, interoperability, customization, composition, and deployment.

Component model implementations are based on a particular component model. They provide
- A run-time environment
- Basic services
- Horizontal services that are useful across multiple domains
- Vertical services providing functionality for a particular domain

for software components.

Biographies

Rainer Weinreich is assistant professor at the Johannes Kepler University in Linz, Austria. His main research interests lie in the area of component-based and distributed software architectures. He is the lead architect of several frameworks for object-oriented, component-based, and distributed software systems and currently is leading a project for agent-based remote diagnosis and supervision of process automation systems. Information about his research activities can be found at http://www.swe.uni-linz.ac.at/weinreich.

Johannes Sametinger is associate professor at the Johannes Kepler University in Linz, Austria. He currently holds a position at the University of Regensburg, Germany . His research interests include software engineering, software documentation, software maintenance, software reuse, object-oriented programming, component-based programming, and programming environments. Information about his research activities can be found at http://www.swe.uni-linz.ac.at/sametinger.