1

Component Software and the Way Ahead

Clemens Szyperski
Microsoft Research
One Microsoft Way, Redmond, WA 98053 USA
cszypers@microsoft.com

Abstract

Components capture the deployment nature of software; objects capture its runtime nature. Components and objects together enable the construction of next-generation software. However, as discussed in this chapter, many problems still need to be solved before component software can become ubiquitous. One important step to be taken is to move from component introversion to component extroversion and to adopt component-based software architecture on a much broader basis. To avoid the many traps on that way, it is useful to emphasize: Components are units of deployment and versioning but the atoms of configuration. To control the complexity explosion of peer-to-peer component architectures, component frameworks need to be pursued beyond their current weak foundation.

1.1 Introduction

From the early days of discovering that software construction ought to be an engineering discipline it has been recognized that the one key concept to learn from other engineering disciplines is the notion of prefabrication of more generic parts and their assembly to form more specific parts. Such "parts" can figure at different levels and stages of a production process—but only if they form isolatable parts of deployed solutions should they be called software components. Parts used to generate the parts that are eventually deployed, should not be called software components to avoid diffusing the component concept to a useless level of generality. In the remainder of this chapter, where unambiguous, software components are simply called components.

The restrictive notion of software components just sketched is, of course, a pragmatic take. From an ontological point of view, everything that can be composed

into a composite is a component. Since plenty of compositional concepts are used on the road towards deployable software, all these could be subsumed under the component umbrella. Examples of such compositional concepts are expressions, functions, statements, procedures, classes, and modules. Of these, only modules—and only if carefully institutionalized—form the basis of software components, but merely primordial ones.

Notably, there is a second space of compositional concepts that deserves separate attention: the space of run-time instances. Composition in this space covers concepts such as co-allocation of variables, address indirection (references and pointers), objects, and object composition via forwarding, aggregation, or delegation. Especially in the space of objects, it is a common simplification to not thoroughly distinguish objects from classes—that is, instances from their generating description. However, this distinction is fundamental to the understanding of what software components should be.

Components are on the upswing, while objects have been around for a while. It is therefore understandable but not helpful to see object-oriented programming sold in new clothes by simply calling objects "components." The emerging component-based approaches and tools combine objects and components in ways that show they are separate concepts. In this chapter, some key differences between objects and components are examined to clarify these blurred areas.

The remainder of this chapter is organized as follows: First, some motivating arguments are given for the use of components and the need to have standards. Then, some key terms are unfolded, explained, and justified. Based on this, a refined component definition is reviewed. Following, some light is shed on the fine line between component-based programming and component assembly. In particular, it is shown that approaches based on assembly really assemble objects, not components, but that they create new components when saving the finished assembly. Taking steps beyond objects, concepts of component frameworks and component system architecture are then introduced.

1.1.1 Why Components?

Reference to more mature engineering disciplines cannot and should not be the key argument for pushing software component technology. So, what is the rationale behind component software? Or rather, what is it that components should be? This section introduces a set of key arguments that motivate the use of components, intentionally preceding later sections that cover software components in detail.

Traditionally, closed solutions with proprietary interfaces addressed most customers' needs. Heavyweights such as operating systems and database engines are among the few examples of components that have reached high levels of maturity. Large software-systems manufacturers often configure delivered solutions by combining modules in a client-specific way. However, the interfaces between such

modules tend to be proprietary—at most, open to highly specialized independent software vendors (ISVs) that specifically produce further modules for such systems. In many cases, these modules are fused together during a linking step and are no longer distinguishable in deployed solutions.

There are three sets of arguments in favor of component software:

(i) *Baseline argument:* Component-based solutions can combine acquired and purpose-created components. By combining "make" and "buy", components offer to reduce cost by focussing on core competencies and by avoiding excessive reinvention of the wheel. Hence, strategic components are made while non-strategic ones are bought, perhaps even off-the-shelf (COTS). This way, organizations can maintain their competitive edge—an issue of increasing importance as software dominates more and more aspects of modern organizations.

(ii) *Enterprise argument:* Components can be assembled in a variety of different ways. If the component factoring is performed skillfully, then several products of a *product line* can be covered by configuring a core set of components, plus perhaps some more product-specific ones. Product creation is then largely an issue of configuration. Furthermore, by versioning individual components and reconfiguring systems the evolution of products can be controlled.

(iii) *Dynamic computing argument:* Modern software systems are increasingly challenged by an open and growing set of content types to be processed. A web browser is a good example. If well architected, such systems can be dynamically extended to meet new requirements on demand.

The baseline argument is not as strong as it seems: source-level reuse may often be sufficient for this purpose. Likewise, the enterprise argument still allows for final product integration. In both cases it is thus possible to get away without investing into component technology that supports deployment-time components. However, the dynamic computing argument strictly requires components that are units of deployment. It is interesting to observe that the first two scenarios do benefit from deployable components. In the baseline case the argument is increased robustness: source code is a fragile basis for reuse. In the enterprise case the argument is increased flexibility: deployed systems can be upgraded and adapted to changing needs without requiring clients to install a whole new system.

1.1.2 Component Standards

Deployable components need to be shipped in a "binary" form, that is, a form that is machine processable and does not require any further human intervention. Traditionally, this form was a loadable image holding machine code. Alternatively, and long known, such a form can be lifted to match some virtual machine. Java

bytecode is the presently most popular binary form for a virtual machine. In addition to binary form, a component standard needs to be established that—at this binary level—defines interoperation between components from independent providers. Traditionally, this task was performed by operating systems that defined a *calling convention*. With components that should go beyond simple procedural interfaces, new standard infrastructure is needed to effectively standardize more general calling conventions.

Attempts to create low-level connection standards or wiring standards are either product- or standard-driven. The Microsoft standards, resting on its Component Object Model (COM) [Mic97, Box98, Cha96], have always been product-driven and are thus incremental, evolutionary, and to a degree legacy-laden by nature. Standard-driven approaches usually originate in industry consortia. The prime example here is the Object Management Group (OMG)'s effort. However, OMG's Common Object Request Broker Architecture (CORBA) [Obj98] hasn't contributed much in the component world and is now falling back on JavaSoft's Enterprise JavaBeans (EJB) [Jav98] standards for components, although attempting a CORBA Beans generalization in the CORBA 3 revision. The JavaBeans standard still has a way to go and it is not implementation language-neutral. Re-introducing such neutrality in CORBA Beans—while remaining sufficiently compatible with the evolving EJB specification—is a major challenge. To summarize, the situation is somewhat paradoxical in that it seems there is a choice between platform independence (EJB) and language independence (COM). The CORBA promise of offering both yet remains unfulfilled in the components arena.

At first, it might be surprising that standards for component software are largely pushed by desktop- and Internet-based solutions. On second thought, this should not be surprising at all. Component software is a complex technology to master—and viable, component-based solutions will only evolve if the benefits are clear. Traditional enterprise computing has many benefits, but they all depend on enterprises that are willing to evolve substantially.

In the desktop and Internet worlds, the situation is different. Centralized control over what information is processed when and where is not an option in these worlds. Instead, contents (such as web pages or documents) arrive at a user's machine and need to be processed there and then. With a rapidly exploding variety of content types—and open encoding standards such as XML, monolithic applications have long reached their limits. Beyond the flexibility of component software is its capability to dynamically grow to address changing needs: Components are key to the extensibility and evolvability of software systems. This is the dynamic computing argument introduced above.

1.2 Terms and Concepts

1.2.1 What a Component Is and Is Not

The terms "component" and "object" are often used interchangeably. In addition, constructions such as "component object" are used. Objects are said to be instances of classes or clones of prototype objects. Objects and components both make their services available through interfaces. Language designers add more confusion by discussing namespaces, modules, packages, and so on. It is, therefore, useful to unfold and explain these terms. To remain goal-oriented, here is a first definition of components:

Component A component's characteristic properties are that it is a unit of independent deployment; a unit of third-party composition; and it has no persistent state.

These properties have several implications. For a component to be independently deployable, it needs to be separated from its environment and from other components. A component, therefore, encapsulates its constituent features. In addition, since a component is a unit of deployment, it is never deployed partially.

If a third party needs to compose a component with other components, the component must be self-contained. (A third party is one that cannot be expected to access the construction details of all the components involved.) In addition, the component needs to come with clear specifications of what it provides and what it requires. In other words, a component needs to encapsulate its implementation and interact with its environment through well-defined interfaces and platform assumptions only. It is also generally useful to minimize hard-wired dependencies in favor of externally configurable providers. If all hard-wiring is avoided, then a component is fully connectable (or "pluggable"): it can be used in any context where the required and provided interfaces can be properly connected. See Section 1.4 for a detailed discussion of connections and assembly.

Finally, observe that a component without any persistent state cannot be distinguished from copies of its own—an important and useful property. (Exceptions to this rule are attributes not contributing to the component's functionality, such as serial numbers used for accounting.) Without state, a component can be loaded into and activated in a particular system—but in any given process, there will be at most one copy of a particular component. So, while it is useful to ask whether a particular component is available or not, it is meaningless to ask about the number of copies of that component. (Note that a component may simultaneously exist in different versions. However, these are not copies of a component, but rather related components.) Not copying components does not mean that components cannot support multiple instances. For example, a button component would exist only once in a deployment context, but it could support any number of button instances (see Section 1.2.2 below).

In many current approaches, components are heavyweights. For example, a database server could be a component. If there is only one database maintained by this class of server, then it is easy to confuse the instance with the concept. In the example, the database server, together with the database, can be seen as a component with persistent state. According to the definition described previously, this instance of the database concept is not a component. Instead, the static database server program is and it supports a single instance: the database object. This separation of the immutable plan from the mutable instances is key to avoid massive maintenance problems. If components could be mutable, that is, have state, then no two installations of the same component would have the same properties. The differentiation of components and objects is thus fundamentally about differentiating between static properties that hold for a particular configuration and dynamic properties of any particular computational scenario. Drawing this line carefully is essential to curbing manageability, configurability, and version control problems.

1.2.2 Objects

The notions of instantiation, identity, and encapsulation lead to the notion of objects. In contrast to the properties characterizing components, an object's characteristic properties are that it is a unit of instantiation (it has a unique identity), it has state that can be persistent, and it encapsulates its state and behavior.

Again, several object properties follow directly. Since an object is a unit of instantiation, it cannot be partially instantiated. Since an object has individual state, it also needs a unique identity so it can be identified, despite state changes, for the object's lifetime. Consider the apocryphal story about George Washington's axe, which had five new handles and four new axe-heads—but was still George Washington's axe. This is typical of objects: nothing but their abstract identity remains stable over time.

Since objects get instantiated, a construction plan is needed that describes the new object's state space, initial state, and behavior before the object can exist. Such a plan may be explicitly available and is then called a class. Alternatively, it may be implicitly available in the form of an object that already exists, that is close to the object to be created, and can be cloned. Such a preexisting object is called a prototype object [Lie86, US87, Bla94].

Whether using classes or prototype objects, the newly instantiated object needs to be set to an initial state. The initial state needs to be a valid state of the constructed object, but it may also depend on parameters specified by the client asking for the new object. The code that is required to control object creation and initialization could be a static procedure, usually called a constructor. Alternatively, it can be an object of its own, usually called an object factory, or factory for short.

1.2.3 Object References and Persistent Objects

The object's identity is usually captured by an object reference. Most programming languages do not explicitly support object references; language-level references hold unique references of objects (usually their addresses in memory), but there is no direct high-level support to manipulate the reference as such. (Languages like C provide low-level address manipulation facilities.) Distinguishing between an object—an identity, state, and implementing class—and an object reference (just the identity) is important when considering persistence. As described later, almost all so-called persistence schemes just preserve an object's state and class, but not its absolute identity. An exception is CORBA, which defines Interoperable Object References (IORs) as stable entities (which are really objects). Storing an IOR makes the pure object identity persist.

1.2.4 Components and Objects

Typically, a component comes to life through objects and therefore would normally contain one or more classes or immutable prototype objects. In addition, it might contain a set of immutable objects that capture default initial state and other component resources. However, there is no need for a component to contain only classes or any classes at all. A component could contain traditional procedures and even have global (static) variables; or it may be realized in its entirety using a functional programming approach, an assembly language, or any other approach. Objects created in a component, or references to such objects, can become visible to the component's clients, usually other components. If only objects become visible to clients, there is no way to tell whether a component is pure object-oriented inside, or not.

A component may contain multiple classes, but a class is necessarily confined to a single component; partial deployment of a class wouldn't normally make sense. Just as classes can depend on other classes (inheritance), components can depend on other components (import). The superclasses of a class do not necessarily need to reside in the same component as the class. Where a class has a superclass in another component, the inheritance relation crosses component boundaries and objects created by instantiating such a class are effectively instances carried by multiple components. Whether or not inheritance across components is a good thing is the focus of heated debate. The theoretical reasoning behind this clash is interesting and close to the essence of component orientation [Szy98], but it is beyond the scope of this chapter.

1.2.5 Modules

Components are rather close to modules, as introduced by modular languages in the early 1980s. The most popular modular languages are Modula-2 and Ada. In Ada,

modules are called packages, but the concepts are almost identical. An important hallmark of modular approaches is the support of separate compilation, including the ability to properly type-check across module boundaries.

With the introduction of the Eiffel language, the claim was that a class is a better module [Mey88]. This seemed justified based on the early ideas that modules would each implement one abstract data type (ADT). After all, a class can be seen as implementing an ADT, with the additional properties of inheritance and polymorphism. However, modules can be used, and always have been used, to package multiple entities, such as ADTs or classes, into one unit. Also, modules do not have a concept of instantiation, while classes do. (In module-less languages, this leads to the construction of static classes that essentially serve as simple modules.)

Recent language designs, such as Oberon, Modula-3, and Component Pascal, keep the modules and classes separate. (In Java, a package is somewhat weaker than a module and mostly serves namespace control purposes.) Also, a module can contain multiple classes. Where classes inherit from each other, they can do so across module boundaries. Modules can be seen as minimal components. Even modules that do not contain any classes can function as components. It is important though that the modules that should function as components are individually and separably present at deployment time.

Nevertheless, module concepts don't normally support one aspect of full-fledged components. A module does not come with persistent immutable resources, beyond what has been hardwired as constants in the code. Resources seem to parameterize a component—replacing these resources allows one to make a new version of a component without needing to recompile it, for example, for purposes of localization. Modification of resources may look like a form of a mutable component state. Since components are not supposed to modify their own resources (or their code), the component definition remains useful: resources fall into the same category as the compiled code that forms part of a component.

Component technology unavoidably leads to modular solutions. The software engineering benefits can thus justify initial investment into component technology, even if component markets are not foreseen.

1.3 Components Beyond Modules

It is possible to go beyond the technical level of reducing components to better modules. To do so, it is helpful to define components differently.

A Definition: Component "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." (Workshop on Component-Oriented Programming at ECOOP 1996 [SP97].)

This definition covers the characteristic properties of components as discussed. It covers technical aspects such as independence, contractual interfaces, and composition, and also market-related aspects such as third parties and deployment. It is the unique property of components, not only of software components, to combine technical and market aspects. One possible—and purely technical—interpretation of this view maps this component concept back to that of modules, as illustrated in the following.

- A *component* is a set of simultaneously deployed atomic components. An atomic component is a module plus a set of resources.

This distinction of components and atomic components caters to the fact that most atomic components are not deployed individually, although they could be. Instead, atomic components normally belong to a set of components, and a typical deployment will cover the entire set. Atomic components are the elementary units of deployment, versioning and replacement; although it is not usually done, individual deployment is possible. A module is thus an atomic component with no separate resources.

ActiveX, JavaBeans, and CORBA components come packaged in a form that satisfies this component definition. While none of these approaches uses the term "module", constructs of the same meaning are easily identified. For example, in Java packages are not modules. But the atomic units of deployment aren't modules either; they are class files and resources. A single package is compiled into many class files—one per class. However, a set of class files and a set of resources are combined into a JAR (Java archive) file to be shipped as a Java component.

- A *module* is a set of classes and possibly constructs that are not object-oriented, such as procedures or functions.

Modules may statically require the presence of other modules in order to work. Hence, a module can be deployed if all the modules it depends on are available. The dependency graph must be acyclic or else a group of modules in a cyclic-dependency relation would always require simultaneous deployment, violating the defining property of modules. Also, modules that underly components should not have a persistent state, as argued in Section 1.2.1.

- A *resource* is a frozen collection of typed items.

The resource concept could include code resources to subsume modules. The point is that there are resources besides the ones generated by a compiler compiling a module or package. In a pure-objects approach, resources are serialized immutable objects. They're immutable because components have no persistent identity. Duplicates cannot be distinguished.

1.3.1 Interfaces

A component's interfaces define its access points. These points let a component's clients, usually components themselves, access the component's services. Normally, a component has multiple interfaces corresponding to different access points. Each access point may provide a different service, catering to different client needs. It is important to emphasize the interface specifications' contractual nature. Since the component and its clients are developed in mutual ignorance, the standardized contract must form a common ground for successful interaction. What nontechnical aspects do contractual interfaces need to obey to be successful?

First, undue market fragmentation must be avoided, as it threatens the viability of components. The redundant introduction of similar interfaces should also be minimized. In a market economy, such a minimization is usually either the result of early standardization efforts in a market segment, or the result of fierce eliminating competition. In the former case, the danger is suboptimality due to committee design; in the latter case, it is suboptimality due to the nontechnical nature of market forces.

Second, to maximize the reach of an interface specification, and of components implementing this interface, common media are needed to publicize and advertise interfaces and components. If nothing else, this requires a small number of widely accepted unique naming schemes.

1.3.2 Explicit Context Dependencies

Besides specifying provided interfaces, the previous definition of components also requires components to specify their needs. That is, the definition requires specification of what the deployment environment will need to provide, such that the components can function. These needs are called context dependencies, referring to the context of composition and deployment. If there were only one software-component world, it would suffice to enumerate required interfaces of other components to specify all context dependencies [OB97]. For example, a mail-merge component would specify that it needs a file system interface. Note that with today's components even this list of required interfaces is not normally available. The emphasis is usually just on provided interfaces.

In reality, several component worlds coexist, compete, and conflict with each other. Currently there are at least three major worlds emerging, based on OMG's CORBA, Sun's Java, and Microsoft's COM. In addition, component worlds are fragmented by the various computing and networking platforms. This is not likely to change soon. Just as the markets have so far tolerated a surprising multitude of operating systems, there will be room for multiple component worlds. Where multiple such worlds share markets, a component's context dependencies specification must include its required interfaces and the component world (or worlds) it has been prepared for.