**C. Szyperski**

# Components vs. Objects vs. Component Objects [*]

**Abstract**

In the present upswing of component software it is not too surprising to observe a common confusion of terms. It is understandable but not helpful that some promoters of object-oriented programming try to 'sell' their established apparatus to the component world by renaming objects into components. Emerging component-based approaches and tools combine objects and components in ways that hint at the possibility that these are indeed different concepts. This article highlights the key differences between objects and components, points out that they are orthogonal concepts, and provides an integrating conceptual framework that addresses systems supporting objects and components. Seeming conflicts are resolved that result from 'visual assembly' tools, which really assemble objects, not components.

## Introduction

Monolithic software still dominates the world of software products. Substantial technical difficulties in establishing a functioning component approach have been overcome only partially and resulting component technology is not perfect. However, a bigger obstacle is the slow emergence of standards – or *de facto* standards – that carry viable component-based products. To fully explore the component software potential, agreement is required on the precise meaning of terms and concepts.

### Why Components?

Before discussing the underlying concepts it is helpful to understand the core rationale behind component software, leading to an intentional understanding of what components *should be*. Traditionally, closed solutions with proprietary interfaces did address most customers' needs. Among the few examples of components that did reach high levels of maturity are heavyweights such as operating systems and database engines. Manufacturers of large software systems 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.

Attempts to create low-level connection standards ('wiring standards') so far fell into two categories: product or standard driven. The Microsoft standards, resting on its Component Object Model (COM), have always been product-driven and are thus incremental, evolutionary, and to a degree legacy-laden by nature. Other examples exist, predominantly in the area of extensible graphics and multimedia architectures such as Apple's QuickTime, but no other vendor so far has been able to push component software standards with such broad reach as Microsoft has done. Nevertheless, Microsoft still has a lot of ground to cover. As their approach originated on the desktop, it now has to claim both the worlds of Internet/intranet and of enterprise computing.

Standard-driven approaches usually originate in industry consortia. The prime example here is the Object Management Group's effort. However, especially in the component world, OMG hasn't contributed much and is now falling back on JavaSoft's JavaBeans standards for components, although attempting a generalization 'CORBA Beans.' JavaBeans itself still has a long way to go as so far it is not implementation language neutral and bridging standards to Java external services and components are only emerging (CORBA Components; Enterprise JavaBeans). At first it might be surprising that component software is largely pushed by desktop and Internet based solutions. On second thought, this should not surprise at all. Component software is a complex technology to

---

[*] This article is partially based on the author's book *Component Software* (Szyperski 1998).

master and viable solutions will only evolve in this direction if the benefits are clear. There are many benefits that can be stated for traditional enterprise computing, but they all rest on a number of assumptions about the willingness of enterprises to evolve substantially.

In the desktop and Internet worlds, the situation is different. In these worlds centralized control over what information is processed when and where is not an option. Instead, contents (such as web pages or documents) 'arrives' at a user's machine and needs to be processed there and then. With a rapidly exploding variety of content types, monolithic applications have long reached their limits. In an increasingly content-oriented world, monolithic software is no longer solving the problem. Beyond the flexibility of component software it is its capability to dynamically grow to address changing needs that makes it compulsory in the desktop and Internet worlds of today; most likely to be followed by the enterprise-computing world in the near future.

### What a component is and is not

The separate existence and mobility of components, as witnessed by Java applets or ActiveX components, has a tendency to make components 'look' similar to objects. Indeed, 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 are both making their services available through *interfaces*. Components are said to be *whitebox* or *blackbox*, and some even identified *gray-* and *glassboxes*. Language designers add further irritation by also talking about *namespaces*, *modules*, *packages*, and so on. This plethora of terms and concepts needs to be unfolded, explained, and justified. The next section browses the key terms and concepts with relatively brief explanations, relating the concepts to each other. The goal is to establish some degree of order and intuition as a basis for further discussions. Then, a refined component definition is presented and discussed. Building on these terms, some light is shed on the fine line between component-based programming and component assembly.

## Terms and Concepts

Some degree of familiarity with most of the terms covered in this section is assumed—and so is some degree of confusion about where one term ends and where another starts. One way to capture the intuitively intended meaning of a term is to enumerate characteristic properties. The idea is: something is an *A* if it has properties $a_1$, $a_2$, and $a_3$. For example, according to Wegner's (1987) definition, a language is called *object-oriented* if it supports objects, classes, and inheritance.

### Components

The characteristic properties of components are:

- A component is a unit of independent deployment.
- A component is a unit of third-party composition.
- A component has no persistent state.

These properties have several implications. Analyzing these implications helps to capture properties that might at first inspection be missed in the above list.

For a component to be independently deployable, the component needs to be well separated from its environment and from other components. A component therefore encapsulates its constituent features. Also, since it is a unit of deployment, a component will never be deployed partially. In this context, a third party is one that cannot be expected to have access to the construction details of all the involved components.

For a component to be composeable with other components by such a third party, the component needs to be sufficiently self-contained. Also, it needs to come with clear specifications of what it requires and provides. In other words, a component needs to encapsulate its implementation and interact with its environment through well-defined interfaces.

Finally, for a component not to have any persistent state, it is required that the component cannot be distinguished from copies of its own. Possible exceptions to this rule are attributes not contributing to the component's functionality, such as serial numbers used for accounting. Not having state, a component can be loaded into and activated in a particular system, but it makes little sense to have multiple instances. In other words, in any given process there will be at most one instance of a particular component. Hence, while it is useful to ask whether a particular component is available or not, it is not meaningful to talk 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 different components related to each other by a versioning scheme.)

In many current approaches, components are heavyweights with just one instance in a system. 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. For example, the database server together with the database might be seen as a module with global state. According to the above definition, 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 were allowed to be mutable, i.e., to have states, 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 curb the problems of manageability, configurability, and version control.

**Objects**

The notions of instantiation, identity, and encapsulation lead to the notion of objects. In contrast to the properties characterizing components, the characteristic properties of objects are:

- ❖ An object is a unit of instantiation; it has a unique identity.
- ❖ An object has state; this state can be persistent state.
- ❖ An object encapsulates its state and behavior.

Again, a number of object properties directly follow. Since an object is a unit of instantiation, objects cannot be partially instantiated. Since an object has individual state, it also has a unique identity that suffices to identify the object despite state changes for its entire 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 a good example for a real-life object: nothing but its abstract identity remained stable over time.

Since objects get instantiated, there needs to be a construction plan that describes the state space, initial state, and behavior of a new object. Also, that plan needs to exist before the object can come into existence. 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 sufficiently close to the object to be created, and that can be cloned. Such a preexisting object is called a *prototype object* (Lieberman, 1986; Ungar and Smith, 1987; Blaschek, 1994).

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.

**Object references and persistent objects**

The identity of an object 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 (a triple of identity, state, and implementing class) and an object reference (just holding 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 that defines Interoperable Object References (IORs) as stable entities (really objects in their own right). Storing an IOR makes the pure object identity persist.

**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 even any classes at all. A component could contain traditional procedures and even have global (static) variables; or it may in its entirety be realized using a functional programming approach; or using assembly language, or any other approach. Objects created in a component, or references to such objects, can leave the component and become visible to the components clients, usually other components. If only objects become visible to clients, there is no way to tell whether a component is 'all object-oriented' inside, or not.

A component may contain multiple classes, but a class is necessarily confined to be part of a single component; partial deployment of a class would not 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 itself. Where a class has a superclass in another component, the inheritance relation crosses component boundaries. Whether or not inheritance across components is a 'good thing' is the focus of a heated debate between different schools of thought. The deeper theoretical reasoning behind this clash is interesting and close to the essence of component orientation (Szyperski, 1998).

**Modules**

From the discussions so far, it should be clear that components are rather close to modules, as introduced by modular languages in the early 80's. 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 truly modular approaches is the support of *separate compilation*, including the ability to properly type-check across module boundaries.

With the introduction of the language Eiffel the claim was promoted that a class is a better module (Meyer, 1988). 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 indeed 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.)

In recent language designs, such as Oberon, Modula-3, Component Pascal, and Java, the notions of modules (or packages) and classes are kept separate. Also, a module can contain multiple classes. Where classes inherit from each other, they can do so across module boundaries. As an aside: In Smalltalk systems it was traditionally acceptable to modify existing classes to build an application. Attempts have been made to define 'module' systems for Smalltalk that capture components that cut through classes, e.g., Fresco (Wills, 1991). Composition of such modules from independent sources is not normally possible though and this approach is therefore not further followed here.

Unlike classes, modules can indeed be seen as minimal components. Even modules that do not contain any classes can function as components. A good example is a traditional math library that

could be packaged into a module and that is of functional rather than object-oriented nature. Nevertheless, one aspect of full-fledged components is not normally supported by module concepts. There are no persistent immutable resources that come with a module, beyond what has been hardwired as constants in the code. Resources parameterize a component. Replacing these resources allows to version a component without a need to recompile; localization is an example. Modification of resources may look like a form of mutable state of a component. Since components are not supposed to modify their own resources, this distinction remains useful: resources fall into the same category as the compiled code that forms part of a component.

Modularity is not a new concept and indeed a prerequisite for component technology. Unfortunately, the vast majority of software solutions today are not even modular. For example, it is common practice that huge enterprise solutions operate on a single database, allowing any part of the system to depend on any part of the data model. Adopting component technology requires adoption of principles of independence and controlled explicit dependencies. Component technology unavoidably leads to modular solutions. The software engineering benefits can be sufficient to justify initial investment into component technology, even if component markets are not foreseen.

### Whitebox versus blackbox abstractions and reuse

*Blackbox vs. whitebox abstraction* refers to the visibility of an implementation 'behind' its interface. Ideally, clients of a blackbox know no details beyond the interface and its specification. For a whitebox, the interface may still enforce encapsulation and limit what clients can do (although implementation inheritance allows for substantial interference). However, the whitebox implementation is available and can be studied to enhance the understanding of what the box does. (Some authors further distinguish between whiteboxes and glassboxes, where a whitebox allows for manipulation of the implementation, while a glassbox merely allows studying the implementation.)

*Grayboxes* are those that reveal a controlled part of their implementation. This may seem to be a dubious notion since a partially revealed implementation could be seen as part of the specification. A complete implementation would merely have to ensure that, as far as observable by clients, the complete specification performs as the abstract partial one. This is the standard notion of *refinement* of a specification into an implementation (Büchi and Weck, 1997; Morgan, 1990).

*Blackbox reuse* refers to reusing an implementation without relying on anything but its interface and specification. For example, typical application-programming interfaces (APIs) reveal no implementation details. Building on such an API is thus blackbox reuse of the API's implementation. In contrast, *whitebox reuse* refers to using a software fragment, through its interfaces, while relying on the understanding gained from studying the actual implementation. Most class libraries and frameworks are delivered in source form and application developers study the classes implementation to understand what a subclass can or has to do.

For an analysis of the serious problems of whitebox reuse see Szyperski (1998); here it suffices to note that whitebox reuse renders it unlikely that the reused software can be replaced by a new release. Such a replacement will likely break some of the reusing clients, as these depend on implementation details that may have changed in the new release.

## A Definition: Component

From the above characterization, the following definition can be formed:

- ❖ 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.

This definition was first formulated at the 1996 European Conference on Object-Oriented Programming (ECOOP) as one outcome of the Workshop on Component-Oriented Programming (Szyperski

and Pfister 1996). The definition covers the characteristic properties of components discussed before. It has a technical part with aspects such as independence, contractual interfaces, and composition, and also a market-related part with aspects such as third parties and deployment. It is the unique property of components, not only of software ones, to combine technical and market aspects.

A purely technical interpretation of this view maps this component concept back to that of modules:

- A *component* is a set of normally simultaneously deployed atomic components.

This distinction of components and atomic components caters for the fact that most atomic components are not deployed individually, although they could. Instead, atomic components normally belong to a set of components and a typical deployment will cover the entire set.

- An *atomic component* is a pair of a module and a set of resources.

Atomic components are the elementary units of deployment, versioning and replacement; although not usually done, individual deployment is possible. A module is thus an atomic component with no separate resources. (Java packages are *not* modules: the atomic units of deployment in Java are class files. A single package is compiled into many class files – one per class.) The above technical component definition is in line with the broader intentional component definition above, but there is room for other technical definitions that nevertheless respect the broader definition.

- A *module* is a set of classes and possibly non-OO constructs, such as procedures or functions.

Obviously, in order to work a module may statically require the presence of other modules. Hence, a module can only be deployed if all modules that it depends on are also 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.

- 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 those generated by a compiler compiling a module or package. In a 'pure objects' approach, resources are serialized immutable objects. Immutable, because components have no persistent identity: duplicates cannot be distinguished. (Component instances have identity.)

**Interfaces**

For the purposes of this discussion it suffices to view a component's interfaces as defining the component's access points. These points allow clients of a component, usually components themselves, to access the services provided by the component. Normally, a component will have multiple interfaces corresponding to different access points. Each access point may provide a different service, catering for different client needs. Emphasizing the contractual nature of the interface specifications is important: since the component and its clients are developed in mutual ignorance, it is the standardized contract that forms a common ground for successful interaction. What are the non-technical aspects that contractual interfaces have to obey to be successful?

First, the economy of scale has to be kept in mind. A component can have multiple interfaces, each representing a service that the component offers. Some of the offered services may be less popular than others, but if none are popular and the particular combination of offered services is not either, the component has no market. In such a case, the overheads involved in casting the particular solutions into a component form may not be justified.

Notice, however, that individual adaptations of component systems may well lead to the development of components that themselves have no market. In this situation, extensions to the component system should build on what the system provides, and the easiest way of achieving this may well be the development of the extension in component form. In this case, the economic

argument applies *indirectly:* while the extending component itself is not viable, the resulting combination with the extended component system is.

Second, undue fragmentation of the market has to be avoided as it threatens the viability of components. Redundant introductions of similar interfaces have thus to be minimized. In a market economy, such a minimization is usually the result of either early standardization efforts among the main players 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 non-technical nature of market forces.

Third, to maximize the reach of an interface specification, and of components implementing this interface, there need to be common media to publicize and advertise interfaces and components. If nothing else, this requires a small number of widely accepted unique naming schemes. Just as ISBN (International Standard Book Number) is a worldwide and unique naming scheme to identify any published book, a similar scheme is needed to refer abstractly to interfaces 'by name.' Just as with an ISBN, a component identifier is not required to carry any meaning. An ISBN consists of a country code, a publisher code, a publisher-assigned serial number, and a checking digit. While it reveals the book's publisher, it does not code the book's contents. Meaning may be hinted at by the book title, but book titles are not guaranteed to be unique.

**Explicit Context Dependencies**

Besides the specification of provided interfaces, the above 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 (Olafsson and Bryan, 1997). 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, there are several *component worlds* that partially coexist, partially compete, and partially 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 themselves fragmented by the various computing and networking platforms. This is not likely to change soon. Just as the markets so far tolerated a surprising multitude of operating systems, there will be room for multiple component worlds. In a situation where multiple such worlds share markets, a component's specification of context dependencies must include its required interfaces *and* the component world (or worlds) it has been prepared for.

There will, of course, also be secondary markets for cross-component-world integration. In analogy, consider the thriving market for power-plug adapters for electrical devices. Thus, bridging solutions, such as OMG's COM/CORBA Interworking standard, mitigate chasms.

**Component 'weight'**

Obviously, a component is most useful if it offers the 'right' set of interfaces and has no restricting context dependencies at all. That is, if it can perform in all component worlds and requires no interface beyond those whose availability is guaranteed by the different component worlds. However, only very few components, if any, would be able to perform under such weak environmental guarantees. Technically, a component could come with all required software bundled in, but that would clearly defeat the purpose of using components in the first place. Note that part of the environmental requirements is the *machine* the component can execute on. In the case of a virtual machine, such as the Java VM, this is a straightforward part of the component world

specification. On native code platforms, a mechanism such as Apple's 'Fat Binaries,' that packs multiple binaries into one file, would still allow a component to run 'everywhere.'

Instead of constructing a self-sufficient component with everything built in, a component designer may have opted for 'maximal reuse.' To avoid redundant implementations of secondary services within the component, the designer decided to 'outsource' everything but the prime functionality that the component offers itself. Object-oriented design has a tendency towards this end of the spectrum and many object-oriented methodists advocate this maximization of reuse.

Although maximizing reuse has many oft-cited advantages, it has one substantial disadvantage: the explosion of context dependencies. If designs of components were, after release, frozen for all time, and if all deployment environments were the same, this would not pose a problem. However, as components evolve, and different environments provide different configurations and version mixes, it becomes a showstopper to have a large number of context dependencies. With each added context dependency it becomes less likely that a component will find clients that can satisfy the environmental requirements. To summarize:

❖ Maximizing reuse minimizes use.

In practice, component designers have to strive for a balance. When faced with requirements that specify the interfaces that a component should at least provide, a component designer has a choice. Increasing the context dependencies usually leads to leaner components by means of reuse, but also to smaller markets. Additionally, higher vulnerability in the case of environmental evolution must be expected, such as changes introduced by new versions. Increasing the degree of self-containedness reduces context dependencies, increases the market, and makes the component more robust over time, but also leads to 'fatter' components. Figure 1 illustrates the optimization problem resulting from trading leanness against robustness.

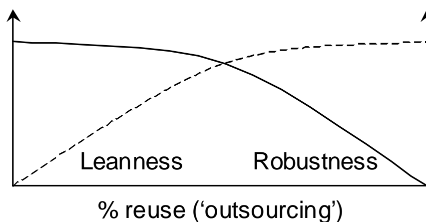


**Figure 1 — Opposing force fields of robustness (limited context dependence) and leanness (limited 'fat'), as controlled by the degree of reuse of a component.**

The effective costs of making a component leaner, versus making it more robust, need to be estimated to turn this qualitative diagram into a quantitative optimization problem. There is no universal rule here; the actual costs depend on factors of the producing organization and of the target markets. The markets determine the typical deployment environment and the client expectations, including component 'weight' and expected lifetime.

## Component-based programming vs. component assembly

Component technology is sometimes seen as a synonym for 'visual assembly' of pre-fabricated components. Indeed, for relatively simple applications surprising productivity can be reached by wiring 'components' – as an example, JavaSoft's BeanBox allows a user to connect beans visually and displays such connections as pieces of pipework: plumbing instead of programming!

It is useful to take a look behind the scenes. When 'wiring' or 'plumbing' components, the visual assembly tool registers event listeners with event sources. For example, if the assembly of a button and a text field should clear the text field whenever the button is pressed, then the button is the event source of the event 'button pressed' and the text field is listening for this event. While details are of no importance here, it is clear that this assembly process is *not* primarily about components. The button and the text field are obviously *instances*, i.e., objects, and not components. (When adding the first object of a kind, an assembly tool may need to locate an appropriate component.)

However, there is a problem with this analysis. If the assembled objects are saved and distributed as a new component, how can this be explained? The key here is to realize that it is not the graph of

particular assembled objects that is saved. Instead, the saved information suffices to generate a *new* graph of objects that happens to have the same topology (and, to a degree, the same state) as the originally assembled graph of objects. However, the newly generated graph and the original graph will *not* share common objects: the object identities are all different. More formally, the objects are saved *modulo their embedding graph*. Orthodox persistent-object approaches store the objects *modulo the universe*, i.e., preserve universal identity instead of identify relative to a confined graph.

It is thus appropriate to view the stored graph as *persistent state* but not as persistent objects. Therefore, what seems to be assembly at the instance rather than the class level, and thus something fundamentally different, turns out to be a matter of convenience. In fact, there is no difference in outcome between this approach of assembling a component out of sub-components and a traditional programmatic implementation that 'hard-codes' the assembly. Indeed, visual assembly tools are free to not save object graphs, but to generate code that when executed creates the required objects and established their interconnections. The main difference is the degree of flexibility left in theory. The saved object graph could be easily modified at run-time of the deployed component, while the generated code would be harder to modify. This line is much finer as it may seem –the real question is whether components with 'self-modifying code' are desirable. Almost always they are not, since the resulting management problems immediately outweigh possible advantages of flexibility.

It is interesting that persistent objects in the precise sense of the word are only supported in two contexts: object-oriented databases, still restricted to a small niche of the database market, and CORBA-based objects. In these approaches, object identity is indeed preserved when storing objects. However, for the same reason, these approaches cannot be used when the intention is saving state and topology *but not* identity. An expensive *deep copy* of the saved graph would be needed to effectively undo the initial effort of saving the universal identities of the involved objects.

The two primary component approaches, COM and JavaBeans, on the other hand both do not immediately support persistent objects. Instead, the emphasis is on only saving the state and topology of a graph of objects. The Java terminology is 'object serialization.' While object graph serialization would be more precise, this is much better than the COM use of the term persistence in a context where object identity is not preserved. Indeed, saving and loading again an object graph using serialization (or COM's 'persistence' mechanisms) is equivalent to a deep copy of the object graph. (This equivalence is used in many object-oriented systems to implement deep copying.)

While it might seem like a major disadvantage of these approaches when compared against CORBA, it should be noted that persistent identity is a heavyweight concept that can always be added where needed. For example, COM supports a standard mechanism called *monikers*, objects that resolve to other objects. A moniker can be used to carry a stable unique id (a *surrogate*) and the information needed to locate that particular instance. The resulting construct is about as heavyweight as the standard CORBA Object References. Java does not yet offer a standard like COM monikers, but one could be added easily.

**Component objects**

Components carry instances that act at run-time as prescribed by their generating component. In the simplest case a component is simply a class and the carried instances are objects of that class. However, most components (whether COM or JavaBeans) will consist of many classes. A Java bean is externally represented by a single class and thus a single kind of object representing all possible instantiations (uses) of that component. A COM component is more flexible. It can present itself to clients as an arbitrary collection of objects, where clients only see sets of interfaces that are unrelated. In JavaBeans or CORBA multiple interfaces are in the end always merged into one implementing class. This prevents proper handling of important cases such as components that support multiple versions of an interface, where the exact implementation of a particular method shared by all these versions needs to depend on the version of the interface the client is using.

### Mobile components vs. mobile objects

Surprisingly, mobile components and objects are just as orthogonal as regular components and objects are. As demonstrated by the Java applet or the ActiveX approach, it is useful to merely ship a component to a site and then start from fresh state and context at the receiving end. Likewise, it is possible to have mobile objects in an environment that isn't component-based at all. For example, Modula-3 Network Objects can travel the network, but do not carry their implementation with them – instead, it is expected that all required code is already available everywhere. To achieve certain effects it may be necessary to support both. For example, a mobile agent (a mobile autonomous object) that travels the Internet to gather information should be accompanied by its supporting components. A recent example are Java Aglets (agent applets).

## Conclusions

While components capture the static nature of a software fragment, objects capture the dynamic nature. Simply treating everything as dynamic can eliminate this distinction. However, it is a time-proven principle of software engineering to try and strengthen the static description of systems as much as possible. Dynamics can always be superimposed where needed and modern facilities such as meta-programming and just-in-time compilation simplify this soft treatment of the boundary between static and dynamic. Nevertheless, as a principle it is always advisable to initially capture as many static properties of an architecture or a design explicitly. In other words, it is advisable to turn arising invariants into explicitly declared invariants, i.e., 'invariants by chance' into invariants by construction. Capturing such crystallized invariants in reusable units of economically viable granularity is the purpose of components. Capturing the dynamic nature of the arising systems built out of components is that of objects. Component objects are then objects carried by identified components. Clearly, this can be specialized for objects of specific nature, such as autonomous objects, often called *software agents*. The resulting notion of *component agents* is indeed emerging.

**References**


Box D. (1998) *Essential COM*. Object Technology Series, Addison-Wesley.

Blaschek G. (1994) *Object-Oriented Programming with Prototypes*. Springer-Verlag.

Büchi M. and Weck W. (1997) A plea for gray-box components. *Technical Report No. 122*, Turku Centre for Computer Science, Turku, Finland.

Chappel D. (1996) *Understanding ActiveX and OLE*. Microsoft Press.

Crelier, R. (1994) *Separate Compilation and Module Extension*. Ph.D. Thesis No. 10650, Swiss Federal Institute of Technology, Zurich.

Gates B. (1988) VBA and COM. BYTE, 23:3, pp. 70-72, March 1998.

IBM (1997) NetRexx 1.0, www2.hurley.ibm.com/netrexx/, May 1997.

IONA Technologies (1998) *Orbix COMet*, www.iona.com.

Intermetrics (1997) AppletMagic Ada 95 to Java bytecode translator. www.appletmagic.com, February 1997.

Lieberman H. (1986) Using prototypical objects to implement shared behavior in OO systems. *OOPSLA'86*. SIGPLAN Notices, 21(11), 214-223.

Meyer B. (1988) *Object-Oriented Software Construction*, 2nd edn. 1997, Series in Computer Science. Prentice-Hall, Englewood Cliffs, NJ.

Microsoft (1997) *The Component Object Model Specification*, www.microsoft.com/com/.

Morgan C (1990) *Programming from Specifications*. Prentice-Hall, Englewood Cliffs, NJ.

Oberon microsystems (1997) BlackBox Component Builder. www.oberon.ch.

Olafsson A. and Bryan D. (1997) On the need for Required Interfaces of components. *Intl. Workshop on Component-Oriented Programming (WCOP96)* at ECOOP96, Linz, *Special Issues in Object-Oriented Programming—ECOOP'96 Workshop Reader*, pp. 159-171, dpunkt Verlag.

OMG (1998) *CORBA 2.2 Specification*. Object Management Group, www.omg.org.

Orfali R., Harkey D., and Edwards J. (1996) *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, New York.

Szyperski C. and Pfister C. (1997) Workshop on Component-Oriented Programming (WCOP'96), Summary. In Mühlhäuser M. (ed.) *Special Issues in Object-Oriented Programming—ECOOP'96 Workshop Reader*, pp. 127-130, dpunkt Verlag, Heidelberg.

Szyperski C. (1988) *Component Software: Beyond Object-Oriented Programming*. ACM Press Books, Addison-Wesley, Harlow, UK.

Ungar D. and Smith R.B. (1987) Self: the power of simplicity. *OOPSLA'87*, SIGPLAN Notices, 22(12), 227-241. (Revised: *Lisp and Symbolic Computation*, 4(3), pp. 187-205, 1991.)

Wallace E. and Wallnau K.C. (1996) A situated evaluation of the Object Management Group's Object Management Architecture (OMA). *OOPSLA'96*, ACM SIGPLAN Notices, 31(10), 168-178.

Wills A. (1991) Capsules and types in Fresco—program verification in Smalltalk. *ECOOP'91*, LNCS, 512, 59-76, Springer-Verlag.



**Author:**

Prof. Dr. C. Szyperski

Queensland University of Technology, GPO Box 2434, Brisbane, QLD 4001, Australia

Tel.: +61 7 3864 5222, Fax: +61 7 3864 1801

e-mail: c.szyperski@qut.edu.au, web: http://www.fit.qut.edu.au/~szypersk/