# Component-Based Systems[1]

Bruce W. Weide, The Ohio State University, January 2001[2]

Human-engineered physical devices such as cars and appliances and computers, as well as naturally occurring physical objects such as forests and galaxies and nerve bundles, often are called "systems." Both this word and the thinking that underlies it are crucial features of modern software engineering as well. Software systems, however, are purely symbolic entities that have meaning and intellectual interest even without a physical manifestation. Component-based systems in physical engineering and in software engineering therefore have many common features, yet differ in important and sometimes subtle ways.

The characteristics of component-based software engineering as currently practiced can be approached from two different perspectives. One explanation is based on the relationship to fundamental principles of system design. In this view, the key features of component-based software engineering could have developed simply as inevitable consequences of applying general "system thinking" to software systems. Another explanation is based on observing the history of two related branches of software engineering research. In this view, the key features of component-based software engineering actually have resulted from a melding of the concurrent but curiously separate activities of two communities: advances in software reuse research and common off-the-shelf (COTS) component technology, and advances in object-oriented (OO) technology.

This article begins by exploring the fundamental similarities and differences between component-based systems in the physical world and those in the software world (Weide, 1996; Gibson, 2000). It then uses that analysis and a summary of component-based software research as a basis for understanding the currently unfolding transition from second-generation to third-generation OO technology: the integration of component-based software concepts into modern software practice which relies on the identification and use of "objects".

## Component-Based Physical Systems

The system thesis, briefly stated, is that a *system* is any part of the world that someone wants to treat as an indivisible unit with respect to the rest of the world. Pick anything and surround it with an imaginary boundary or wrapper called an *interface*. The dual claims in defining an interface are:

- Some details of the part of the world lying inside the boundary are important only to an insider, and are unimportant to an outsider.
- Some details of the part of the world lying outside the boundary are important only to an outsider, and are unimportant to an insider.

[1] This is a draft of a chapter to appear in *Encyclopedia of Software Engineering*, J.J. Marciniak, ed., John Wiley and Sons, 2001. Copyright © 2001 by the publisher.

[2] Department of Computer and Information Science, The Ohio State University, 2015 Neil Avenue, Columbus, OH 43210; weide.1@osu.edu.

Deciding to call a physical situation a system, then, inherently involves describing that situation, if for no other reason than to locate its interface. This symbolic description characterizes part of the world as inside the interface and everything else as outside it, also known as in the *environment* of the system. For example, considering a TV to be a system means it is necessary to delineate exactly what part of the world is this thing called “the TV.”

The part of the world inside an interface can be subdivided in a similar way. In fact, this is just how to describe the insider’s view. In such a hierarchical partitioning of the world, some systems enclose or contain others. The smaller systems inside are called *subsystems* or *components* of the enclosing system, because they are at a “lower level” than the enclosing system and are visible only in the insider’s view of it. This is detail an outsider, being on the other side of the interface, cannot see. Figure 1 illustrates that “inside” and “outside” are relative terms — the outsider cannot see into the system, the insider cannot see out.

The colloquial term for a person who views a system as an outsider is a “user” of that system. When discussing human-engineered systems, the term *client* is preferred. This term really describes a role played by a person, not the person. For example, an engineer designing a TV who understands its electrical components through their interfaces plays the role of (or more simply, is a) client of those components. One particular client is the *end user* who benefits from system thinking only after the product is put into service. A person who delves into the internal details and organizes subsystems into a coherent collection during product design and development plays the role of an *implementer* of the larger system. Note that this same person simultaneously plays the role of client with respect to the subsystems.

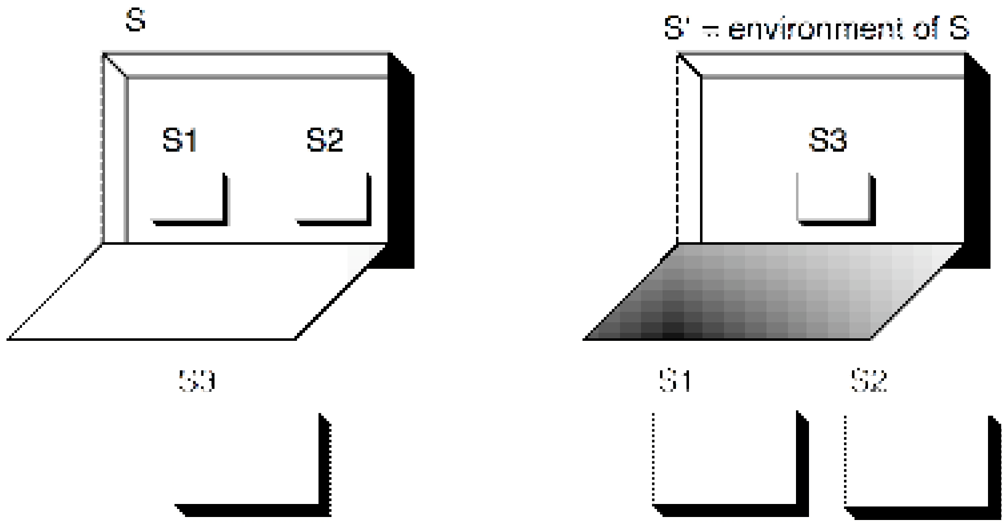

**Figure 1 — Duality between client and implementer views**

*Information Hiding and Abstraction*

The above discussion suggests that an interface is essentially a description of structure: a way of separating what’s inside the system from what’s outside it. But this is only part of the picture. The primary reason for considering a piece of the world to be a system is to enhance the understandability of a situation to the people who play the dual roles of clients and implementers. There is no reason to do this unless there is to be some *information hiding* about what is inside the system. For example, it is helpful to think of a TV as a system so makers of other audio/video systems as well as end users in their living rooms can ignore details like picture tubes, tuners, and other internal technical aspects, and treat the TV as a household appliance with a few external buttons, switches, and connectors that control a picture that appears on a screen and some sound that goes with it.

If none of the complex internal details of the TV are superfluous (a reasonable assumption, since extraneous items probably would add to the TV's cost without any associated benefit) then hiding information about any of them is certain to limit the ability to describe the *behavior* of the TV. Hiding anything about the inside of the TV requires making up some kind of a behavioral cover story to explain the TV's externally visible behavior. This observation leads to the other side of the information-hiding coin, which is called *abstraction.* Abstraction is the technique that permits going beyond merely hiding internal details of a system that are considered inessential from the outside, to "reconceptualizing" the system by explaining its externally visible behavior in terms that are fundamentally different from those that would explain its internal details.

Consider the instruction manual for a TV. It is effectively the TV's interface description for clients, and includes both structural and behavioral aspects. The manual describes what externally accessible buttons, connectors, dials, pictures, and other items are available to connect the TV with other audio/video systems, and their effects in terms of how they operate the TV. Here is a sample statement from an actual TV instruction booklet:

> This TV automatically memorizes the channel being received.

Most people know better! But the implicit anthropomorphic model of the TV helps a client understand how the TV behaves. It does not help anyone understand how that behavior is implemented, but after all, hiding that information is its purpose. Few people could understand the TV's behavior if this part of the manual discussed the circuit that does the "memorizing." And the manufacturer surely wants the flexibility to change that circuit for the next model year without having to go back and rewrite this part of the instructions. So, this is actually an excellent behavioral cover story.

### *Abstract vs. Concrete Descriptions*

The requirement to use abstraction in describing the behavioral aspect of a system's interface leads to the following additional terminology:

- A system's interface description (structural plus behavioral) is called an *abstract description* of the system.
- A description of a system's interior, i.e., the subsystems that comprise it and their relationships to one another and to the interface of the system, is called a *concrete description* of the system.

The instruction manual is an abstract description of a TV. A concrete description of a TV probably includes a wiring diagram that shows the internal connections between external connectors and controls and other internal pieces that are not mentioned in the instruction manual. It explains that there really are electronic circuits involved and how they fit together.

There are, then, two quite distinct kinds of symbolic descriptions of systems. The differences in purpose and nature between abstract and concrete descriptions induce the need for different features in the languages used to write these descriptions. Writing an abstract description demands a language in which to express not just structure, but properties, behaviors, guarantees, obligations, assumptions, etc., that might be needed in a cover story about the system. Depending on the type of system and the nature of the cover story, a language for abstract descriptions must be quite powerful because it might be needed to say nearly anything. Natural languages such as English are commonly used for this purpose, but they are too ambiguous and imprecise for some purposes such as sound and rigorous reasoning about the behavior of a system from its abstract description alone. Engineers therefore routinely use mathematical models to characterize externally visible system behaviors of interest, although these models seldom appear undisguised in end-user instruction manuals.

Writing a concrete description is quite different. It must explain the inventory of subsystems used inside a system and how they are related to each other. It also must say enough about how these subsystems are related to the abstract description of the enclosing system in order to reason about why the cover story is legitimate. Circuit diagrams, free-body diagrams, blueprints, and associated mathematical and natural-language notations are examples of languages used in traditional engineering to write concrete descriptions of systems.

### *The Client-Implementer Contract*

Recall from Figure 1 that an interface is not like a one-way mirror. It is more like a firewall — a symmetric barrier that hides information both ways. Because it serves both clients and implementers, an abstract description has other names. Sometimes it is called the system's *interface specification*, or its *behavioral specification*, or simply its *specification*. This document records a *contract* between a client and an implementer of the system (Meyer, 1997). It simultaneously defines two complementary aspects:

- the structural and behavioral requirements the system places on its clients, or equivalently, the only assumptions a person implementing the interior of a system may make about the clients of that system; and
- the guarantees the system gives in return to its clients, or equivalently, the only assumptions a person using the system as a client may make about the components of that system.

### *Substitutability and Modular Reasoning*

The inability of clients to breach the interface means that inside a system may be any coordinated combination of subsystems that results in satisfying the interface specification. This *substitutability* property is what offers the option of buying any of a large number of TV models from any of a number of TV vendors, plugging your TV into the wall and connecting it to the same remote speakers and other audio/video systems, and expecting the whole combination to work as advertised. Systems such as modern audio/video systems that satisfy the substitutability property are sometimes called *component-based systems* because various vendors can supply subsystems to meet *standard* interface specifications. These systems generally do differ in internal details and, of course, in other ways (e.g., price) that are inessential in meeting the standard interface specifications.

The dual abilities of a client to reason about a system's behavior without peeking inside its interface, and of an implementer to reason about subsystem-to-subsystem behavior without peeking outside its interface, together are called the *modular reasoning* property. Another term for this is *compositionality*; but this term can be misleading because it suggests that merely structural, not behavioral, composition is the key question.

Achieving the modular reasoning property is the central technical objective of using component-based design in engineering. A common misconception is that the motivation for component-based design is economic. Indeed there are economic benefits when parts are standardized and can be supplied by different vendors to deliver the same functionality with different auxiliary characteristics (e.g., price). But these economic benefits accrue *only* when it is legitimate to replace one vendor's concrete component with another's, without worrying that the rest of the system might break as a result of the substitution. For an excellent discussion of economic issues in the context of modular computer system design, and insightful commentary on the theoretical importance of "modularity", see (Baldwin and Clark, 2000).

## Component-Based Software Systems

Many of the terms widely used by software engineers to talk about software systems are identical to those introduced above for physical systems, e.g., system, component, interface, abstract, concrete, contract, etc. The parallels between component-based systems in traditional engineering and in software engineering are most natural if one views a software system as a set of documents describing the behavior of a particular kind of physical system, namely some computer(s) and attached devices. Some of these documents are ordinary computer programs, e.g., procedure and function and class headers, and their code bodies: the concrete descriptions that explain the data structures and algorithms that produce the externally visible behavior. Other documents are specifications: the abstract descriptions that explain the structural and behavioral aspects of the interface.

Following this direction, a *software system* is defined as a set of abstract and concrete descriptions of behavior of some computer(s) and attached devices. A software system is *component-based* if the concrete descriptions it contains include the identification of subsystems or components that are likewise described, down to some level where subsystems are considered primitive.

The abstract descriptions are called *abstract components*; similarly, the concrete descriptions are called *concrete components*. The "components" in a component-based software system, then, are the documents that describe what subsystems and systems do and the source code that describes how they do it. In an OO software system, then, a class is a component, but a class instance (i.e., a run-time object) whose behavior is described by that class is *not* a component.

### *Limits of Traditional System Diagrams*

Consider any set of abstract and concrete descriptions of a component-based physical system. The hallmark of a component-based system design is the potential to substitute any of a number of different concrete descriptions for some abstract description. The restriction is that the concrete description and the abstract description should fit together properly, or be compatible, in the sense that there is some physical system that might have both of those descriptions. Substitutability, then, inevitably leads to the need to distinguish between a system that exists, and the other systems that might be obtained from it by compatible substitutions.

The impact of this observation becomes apparent when considering diagrams like Figure 1 that often are used to convey information about the nesting structure of physical systems. In a diagram of a specific physical system, such as a particular TV, the original notion of a system makes sense: a piece of the physical world with an exterior, an interface, and an interior. But a system diagram of this TV is misleading as a claim about the *design* of the TV, because it ignores the substitutability principle. Thinking of the interior of a particular system as a fixed thing, and not simply as one of the many possible interiors that might be compatible with the abstract description, simply overlooks the key reason for component-based system design.

This means that while nested-box diagrams like the top left of Figure 1 are adequate for informally describing particular instances of physical systems — the kinds of physical systems they were intended to illustrate — these diagrams are inadequate for depicting physical systems that are explicitly component-based. In the latter there might be many compatible interiors for a given system interface, i.e., many potential particular systems. The only way to use traditional system diagrams to show the combinatorially many potential systems that can be built from a given set of subsystems is to exhibit individually the particular systems that could be constructed from them. This is tedious at best. So traditional system diagrams are not really satisfactory from the standpoint of component-based engineering design. They tend to be better for analysis tasks where a single fixed system is to be described and studied.

*Design-Time Relationships*

To overcome this problem, software engineers seem to be the first engineers to have identified the important relationships that might hold between the various documents describing a component-based system — not between the physical systems they describe, but between the descriptions themselves. They depict these relationships using quite different diagrams than those in Figure 1. An example of such a diagram is shown in Figure 2. It uses the notation of a *class diagram* in the current *de facto* industry standard Unified Modeling Language, or UML (Fowler and Scott, 1999). This diagram shows concisely and simultaneously both the relationships among abstract and concrete descriptions and the potentially very large number of particular physical systems that could have (or, in a constructive view, could be built using) those descriptions.

The language of UML class diagrams has an alphabet of symbols that is similar to that used in traditional system diagrams. Rectangles stand for abstract and concrete descriptions. Arrows connecting rectangles stand for certain *coupling*, or *dependence*, relationships between the corresponding descriptions. But there is no use of nesting (i.e., boxes inside other boxes) to suggest physical enclosure, as in traditional diagrams of physical systems.

An open triangular arrow with a dashed line from a concrete description *C* to an abstract description *A* that is (optionally) labeled <<*implements*>> means that a physical system with the concrete description *C* can legitimately have *A* as its abstract description, i.e., that the two descriptions are compatible both structurally and behaviorally. A simple arrow with a dashed line from a concrete description *C* to an abstract description *A* that is (optionally) labeled <<*uses*>> means that a system with the concrete description *C* is a client of a system with the abstract description *A*.

There are other important dependence relationships, but these are the most important for explaining the underpinnings of component-based software engineering and how it has evolved.

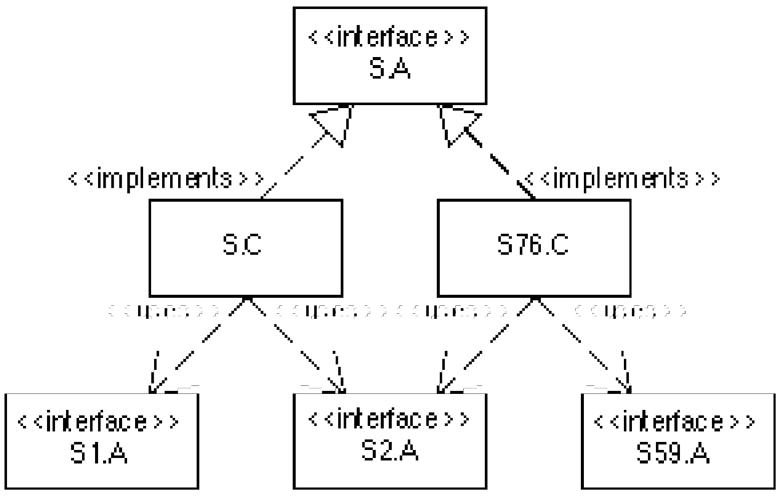

**Figure 2 — A UML class diagram showing important design-time relationships**

To recap, the top left diagram in Figure 1 shows a hierarchy of actual systems culminating in a single actual system called *S*. Figure 2 also shows a hierarchy, but of all the possible systems that could be built from the components whose descriptions are mentioned in the diagram. Figure 2 summarizes important information about substitutability, e.g., that the systems whose internal details are explained in concrete descriptions *S.C* and *S76.C* are substitutable for each other in any client that needs the behavior specified in abstract description *S.A*. This sort of information is simply not represented in diagrams like those of Figure 1, making something like Figure 2 an important piece of design documentation.

## *Dependence Is Bad*

Component *X depends* on component *Y* if it is necessary to understand *Y* in order to understand *X*. In this case, changing *Y* even slightly implies the need to understand the impact of this change on *X*, and possibly means changing *X*, too. Dependence is transitive, i.e., if *X* depends on *Y* and *Y* depends on *Z*, then *X* depends on *Z*.

Some dependencies are unavoidable and important. But the kind of situation software engineers do not want to face is a long *dependence chain*, or path of dependence arrows, among a set of components. Understanding a component requires understanding all the components it depends on, either directly or (by transitivity) indirectly. These are all the components that are reachable from a component by following arrows out of it in its class diagram. Similarly, changing a component might necessitate changing any or all of the components from which the changed component is reachable along a dependence chain.

The intuitive objective that you should minimize such dependencies suggests a general rule-of-thumb for component-based system design:

> Design each component so it depends, either directly or indirectly, on as few existing components as possible. Design each component so as few future components as possible will need to depend, either directly or indirectly, on it.

The best way to observe this rule is to design each component so it depends, directly or indirectly, only on components that contain information which is absolutely required to understand the new component.

## *Concrete-to-Concrete Dependence Is Worse*

The ill effects of long dependence chains arise whether the components involved are abstract components or concrete components. But the effects of dependence chains are more serious when they involve concrete components. To see why, consider the <<uses>> relation, e.g., in Figure 2, which was introduced as a dependence relation between a concrete component and an abstract component. System thinking says it is advantageous for a designer to make each concrete component depend only on the abstract descriptions of the subsystems needed to build it, and not on particular concrete components that implement those abstract descriptions. But the <<uses>> relation also can hold between two concrete components — and in many software systems this is exactly what happens. Rephrased now in terms of dependence chains, the problem with concrete-to-concrete dependencies is that if a concrete component directly uses another concrete component instead of an abstract component, and that concrete component uses another concrete component, and so on, then this can introduce long dependence chains.

In fact, it gets worse. If a designer does not consciously try to limit dependencies between concrete components, then a typical concrete component depends on more than one other component, so the dependencies branch out. Every time there is a dependence from a concrete component to two others, there is a doubling of the number of dependence chains. The resulting chains together form a *dependence graph*. The component at the root of this graph depends on all the components in all the dependence chains in the entire graph — and the number of chains in the graph typically grows very quickly with the lengths of the chains because of the typical branching factor for a concrete component. Figure 3 illustrates how a single concrete component like the one on the top can depend (directly and indirectly) on a very large number of other concrete components and participate in several dependence chains. If the same kinds of dependencies continue for another few levels of arrows, it is clear that changing such a system is complicated. Branching in, e.g., to a highly reusable component, also means that changes to that component cause ripple-effect problems in very many other components. In short, maintenance and evolution of a system with many long dependence chains is more difficult than it would be if those chains were shorter.

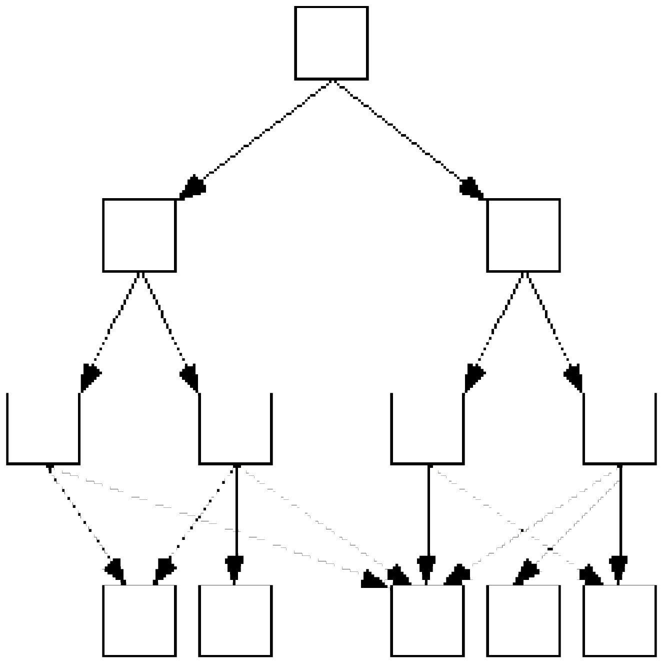

**Figure 3 — Dependence chains cause maintenance problems**

Successful component design minimizes such problems by introducing dependencies only to abstract components (Weide, 1996). Merely by observing system thinking, it should be possible to avoid dependencies from any component to a concrete component. Specific design techniques can be followed to keep the remaining dependence chains among abstract components fairly short. If this is achieved, then dependence of a concrete component on an abstract component effectively damps out the dependence chain that starts there. This observation means there are only two categories of dependencies in well engineered component-based software system designs: abstract-to-abstract dependencies and concrete-to-abstract dependencies. There are no long dependence chains.

In summary, it is possible to explain the current state-of-the-practice in component-based software systems as a natural — in fact, virtually inevitable — consequence of applying system thinking to software systems. But the present situation actually arose via a series of developments over at least 30 years from two related communities: the component-based software research community and the OO software community. The system-thinking framework and terminology also helps present this historical perspective, which starts in the next section with a review of some important technical developments from the first community.

## Advances in Component-Based Software

Component-based software ideas are evident from the earliest days of computing, when "programming" meant connecting wires. Commercial Fortran subroutine libraries for numerical applications became popular in the 1960s. The vision that multiple vendors might supply software components just like electronic and mechanical components (McIlroy, 1976) was suggested at a NATO-sponsored conference in 1968 that marked the beginning of the field of software engineering. A similar notion of "software ICs" (i.e., software integrated circuits) was presented again many years later (Cox, 1986), still as a vision because the original idea had not yet made its way into software practice. While the basic idea of software componentry seems natural and with hindsight can be elaborated in some detail in a programming-language neutral way (as in the first half of this article), it has proved surprisingly difficult to achieve in large part because of programming language shortcomings. So, while the component-based software community investigated many other technical and non-technical impediments to software reuse over the years, this section focuses on language mechanisms related to component-based systems.

In the 1970s, the Unix operating system featured its own style of software components that could be hooked together to form more complex software using "little languages" (Bentley, 1986). But studies of principles for design and implementation of software components did not materialize until the 1980s, when mainstream programming languages began to have features that made it possible to write reasonable software components. The most important of these languages was Ada, which had the backing of the U.S. Department of Defense and therefore the commercial credibility that earlier research languages such as Alphard (Shaw, 1981) and CLU (Liskov et al., 1981) lacked. Ada included several important features:

- the *package* mechanism, providing a convenient way to encapsulate data representations and to compartmentalize cohesive features into a single compilation unit (component);

- the separation of the *package specification* from the *package body*, recognizing a syntactic distinction between the structural part of an abstract component and a concrete component; and

- the *generic* package, providing the flexibility to parameterize a package by types, operations, and constants — but not by other packages (a feature added in Ada95).

Within a few years, an important book called *Software Components With Ada* (Booch, 1987) illustrated how one might use these new language features to design software components. The term *COTS*, for *common off-the-shelf*, components, became popular shortly afterward. Unfortunately, the language mechanisms of Ada turned out to be a bit too weak to create the best component-based software. They did not capture the crucial property that an abstract component might be compatible with multiple concrete components. This made the Booch Ada components clumsy to use; there were over 200 component implementations (hence packages), but only a few conceptually different abstractions were involved. Moreover, Ada did not offer any way to describe behavioral properties in package specifications, an important aspect of abstract components in the underlying system-thinking framework. This led some to propose Ada-based behavioral specification constructs (Luckham et al., 1987) but none were ever added to Ada itself. So the concept of substitutability that underlies system thinking remained only weakly supported by commercial programming languages.

Another approach to software components called *parameterized programming* was being pioneered at about the same time (Goguen, 1984). In the research language OBJ, some related advances appeared:

- the true separation of specifications from implementations;

- a way to place restrictions on component parameters, so the "sanity" of composition by instantiation could be checked statically; and

- constructs for writing the behavioral specifications, called *properties*, required to describe complete contracts in abstract components.

These ideas influenced later research developments in component-based software, including languages such as LILEANNA (Tracz, 1993) and RESOLVE (Sitaraman and Weide, 1994). But most of the more radical and recent innovations from the component-based software community have not yet made their way into widespread use by practitioners.

In fact, an historical perspective of how the current practice of component-based software arose — a practice based on COM, CORBA, and Java — suggests that commercial OO technology has been the primary force driving almost all software practice. There has been some cross-fertilization from the research community, however, as explained in the next section.

## Advances in Object-Oriented Software

Object technology has gone through two generations[1] and has recently entered a third, which a recent popular book (Szyperski, 1998) calls "component software" and describes as "beyond OO".

### *The First Generation of OO Technology (1970-1990)*

The first generation of OO technology began with a near infatuation with inheritance and related OO programming language mechanisms, which were different from those being introduced in languages such as Ada and OBJ. This was the exploratory phase of OO where the software engineering question was how these new mechanisms should be used (as opposed to the programming language question of how they might be used). Early OO designs resulted in software systems with deep inheritance hierarchies. *Class hierarchy* diagrams resembling Figure 3 showed how subclasses depended on, i.e., inherited features from, their superclasses. There was generally no acknowledgment of a distinction between abstract and concrete descriptions. There were only classes, which were concrete descriptions of behaviors of the objects or class instances that were created from them at run-time. Class hierarchy diagrams recorded "is a" and "has a" relationships between these concrete components.

Gradually it became evident from experience that OO software that used deep inheritance hierarchies was relatively hard to maintain when there were many and often quite long dependence chains. Following the ideas of system thinking in physical engineering, successful software designers noticed that distinguishing abstract from concrete components was a good way to help limit dependencies and thereby make software systems easier to understand and change. This realization marked the beginning of the second generation of OO.

### *The Second Generation of OO Technology (1990-2000)*

Some forward-looking designers started recommending the use of *abstract base classes* to serve as the missing abstract components. An abstract base class is a class that contains no code for its methods, and which therefore cannot serve as an implementation of behavior; this code is provided in a concrete class that implements the abstract base class. Moreover, different uses for the inheritance mechanism were teased apart (LaLonde, 1989). The use of specification inheritance was recommended to encode an important conceptual relationship between abstract components, namely, that one abstract component *extends* another. Saying $A_2$ extends $A_1$ means that $A_2$ specifies only the incremental additional behavior to be added to that described by $A_1$. In other words, for any concrete component for which $A_2$ is a valid cover story, $A_1$ is also a valid cover story (Gibson, 2000).

---

[1] Thanks to Furrukh Khan and Martin Griss for independently suggesting (in unpublished comments) the idea that object-oriented technology already has gone through two "generations".

There were two practical problems with abstract base classes and specification inheritance. First, the use of abstract base classes even to record structural (let alone behavioral) aspects of contracts was optional, and many software engineers just didn't bother to use them. Second, linguistic support for those who did use abstract base classes was weak because popular programming languages offered no syntax to describe behavior in abstract terms. Thus, compatibility between a concrete component and an abstract component it was claimed to implement was at best checked by considering structural conformance, with no regard for behavioral conformance. The term *behavioral subtype* was coined to describe the stronger, required behavioral relationship (Leavens, 1991; Liskov and Wing, 1994). A new concrete component that implements an abstract extension, it was advised, should be a behavioral subtype of an existing concrete component that implements the original abstract component. This meant that the substitution of the new concrete component for the original should not break any client program's behavior.

Unfortunately, even today the popular commercial languages do not offer even syntactic slots for writing behavioral specifications to create complete abstract components. But two other early-1990s events caused second-generation OO technology to take off even without this support.

One was the recognition of OO design patterns (Gamma et al., 1995), i.e., patterns of dependencies and detailed design decisions that were found empirically to be present in many successful OO software systems of the day. Some of these design patterns, called "paradigmatic idioms," can be viewed as software engineers' reactions to the failure of programming languages to keep up with the rapid changes in recommended design and programming techniques (Baumgartner et al., 1996). A particularly important pattern from the standpoint of component-based software systems is the *abstract factory pattern*. This is a design and coding technique that permits a software engineer to address a persistent deficiency in OO languages: In the code of a concrete component, any new object that is created must be declared with the name of a concrete component. In C++, for example, the type of an object must be the name of a concrete class; it may not be the name of an abstract base class. This implies that concrete-to-concrete component dependencies are inherently required merely as a result of coding OO designs in C++. The abstract factory pattern offers a work-around for this problem. It does not eliminate concrete-to-concrete dependencies, but at least it localizes them in the code and thereby makes it somewhat easier to change the software. Dependencies on normal concrete components are replaced by dependencies on special concrete components called "abstract" factory components — the name perhaps arising because it would have been nicer if they actually had been abstract.

A less known approach to replacing concrete-to-concrete by concrete-to-abstract component dependencies in C++ is to use the language's relatively recent parameterization mechanism, called *templates* (Sitaraman and Weide, 1994; Weide, 1996). This approach, called *decoupling*, starts with identification of the names of all other concrete components in a given concrete component. Each of these names is systematically replaced by an arbitrarily named formal template parameter, which stands for the name of a concrete component to be bound later by a client. Each such formal template parameter is restricted to be some implementation of the abstract component that describes the behavior the code really depends on. The associated actual parameter is selected not when a concrete component is designed, but only later when it is composed with other concrete components by creating an instance of the template for use in a client program. Like the abstract factory pattern, this technique localizes concrete-to-concrete dependencies in one place and simplifies software changes made by substitution. Another approach that addresses the same basic problems, also based on parameterized programming ideas but with different details, is used with the GenVoca component model (Smaragdakis and Batory, 1998).

The other important second-generation development was the introduction of the programming language Java in the early 1990s (Joy et al., 2000). The clear distinction between abstract and concrete components in the system thinking framework finally had some linguistic support. Java's interfaces (with behavioral specifications of the contract added as comments) can serve as abstract components and its classes as concrete components. Multiple classes may implement a given interface, and a class may implement multiple interfaces. Two important conceptual relationships between components — implements and extends — are encoded by meaningful keywords and not with cryptic inheritance-related and implementation-inspired keywords such as "virtual public" in C++. Finally, some early second-generation design recommendations regarding the distinction between specification inheritance and implementation inheritance, as well as recommended restrictions on when and how to use them, are enforced in Java.

However, there still was no requirement to use Java interfaces to describe structural aspects of a contract, and there still were no provisions at all for describing behavioral aspects of a contract. The claim that a Java class implements an interface can be checked only on the basis of structural conformance. And as in C++, creation of a new object at run-time requires that the name of its concrete class be written in the code. The abstract factory pattern or some other workaround is still needed to limit concrete-to-concrete dependencies. So Java gives notably more linguistic support for substitutability, but it remains incomplete by failing to be prescriptive enough to elevate abstract components (even the structural aspects, as expressed in Java interfaces) to first-class component status. Third-generation OO technologies made this next leap.

*The Third Generation of OO Technology (2000-????)*

Throughout the second generation, there were other developments that foreshadowed the recent practical transition toward component-based software systems. In the early-to-mid 1990s, Microsoft introduced and refined its *Component Object Model*, or *COM* (Box, 1998), and more or less concurrently the Object Management Group (OMG, a consortium of hundreds of information technology companies) introduced its *Common Object Request Broker Architecture*, or *CORBA* (Hoque, 1998). Both technologies offer language-neutral component models along with substantial "middleware" to support design and deployment of component-based software systems. In both cases, the structure of a component-based system is inspired by the ideas of system thinking.

These two commercial component technologies have much in common. Aside from the obvious practical benefits that come from such standards, the most important conceptual advance COM and CORBA bring is that they encourage designers to significantly reduce or even eliminate dependencies on concrete components. They do this by explicitly recognizing the distinction between abstract and concrete components and by requiring clients and implementers to use "interfaces" to record the structural parts of abstract components. They also offer mechanisms to allow a client program to locate (at run-time) a concrete component that implements a given abstract component that it depends on.

COM and CORBA, being language-neutral, permit concrete components to be written in a variety of languages. Modern development systems for COM and CORBA components typically generate the boilerplate code that deals with vendor-supplied middleware services such as support for physical distribution, transaction management, etc. The designer generally must understand the abstract factory pattern in order to use these tools, but the approach effectively leaves the component designer only with the task of supplying code that would need to be written in any case, i.e., concrete-component-specific code required to implement one or more abstract components. Both COM and CORBA have *Interface Definition Languages*, or *IDLs*, in which their abstract components are written. COM and CORBA IDLs have similar syntax. But as with their predecessors, they include syntactic slots only for describing structural aspects of a contract, not for behavioral information.

Sun Microsystems introduced Java-specific component technologies, too, called *Java Beans* (Englander, 1997), *Enterprise Java Beans*, or *EJB* (Monson-Haefel, 1999), and a distributed component technology called *Jini* (Arnold et al., 1999). Java Beans technology, unlike COM and CORBA, does not require the designer to use Java interfaces to explain the structural aspects of all classes. It is basically a second-generation OO technology that takes advantage of some Java language features to support close interactions with a development environment. Moreover, designing and using Java Beans generally involves understanding and coding yet another design pattern, the *observer* pattern, that permits communication among beans to take place using "events." Events facilitate the dynamic addition and/or removal of objects in a group of communicating objects. EJB and Jini are successors that are similar to COM and CORBA in the explicit use of interfaces to describe structural conformance, and these are better classified as third-generation OO technologies.

Another common thread linking all these modern component technologies is their support for physical distribution of software components over networked computers. This was added to COM, but it was a key motivating factor behind the other technologies. It is interesting that network computing makes the importance of contracts and abstract components all the more obvious. Perhaps 30 years ago, it was possible for a software project manager to assume that his team had access to all the source code for the entire system (or at least the portion that was not already well understood through experience, e.g., the underlying operating system). Even 15 years ago this should have been unthinkable; for example, the introduction of the Macintosh toolbox as an application programming interface should have removed all doubt that one could or should have access to source code in order to use concrete components effectively. Now, it is rare that one can hope to see the source code for any (likely, remote) concrete component. Consequently, there is simply no way to know how to use it as a client unless there is a separate abstract component that describes at least its structural, and preferably its behavioral, properties. The only real question is whether these behavioral features are expressed in English, as they were with the Macintosh toolbox "phonebooks" and with most current software documentation, or in some more formal language that might support machine-checking of conformance and offer other advantages.

## The Future

Component-based software systems are a practical reality now. COM is widely used by developers for Microsoft operating systems. Even big applications such as Microsoft® Word and Microsoft® Excel support COM interfaces to internal features that can be invoked from other programs. CORBA is developing a substantial following for platform-independent and language-independent development. EJB surely will become more popular among Java devotees as resources explaining it become easier to understand. Jini introduces attractive infrastructure concepts such as "leasing" of services that should help designers deal with nagging distributed system problems such as host failures. In short, it seems inevitable that most large software development projects will soon involve one of these component technologies or their immediate successors.

It is equally clear that all the commercial component technologies are far from complete solutions to the problem of designing and deploying component-based software systems. A review of system thinking reveals areas where more work is needed. One glaring weakness is the current inability to describe, in interfaces written using existing IDLs or Java, the behavioral aspects of contracts that are needed to create true abstract components (Williams, 1998). Of course, even if syntactic slots were available for doing this, checking the claim that a concrete component implements an abstract component would be difficult. Certainly it would not be nearly as easy as checking structural conformance to an IDL or Java interface because some sort of program verification system would be needed.

A third-generation OO technology mandates the identification and use of abstract components, and insists on recording at least the structural aspects of a contract. However, unless these abstract components are designed very carefully it is intractable to verify an implementation claim even if it is theoretically possible to do so. The primary reason is that pointer/reference aliasing significantly complicates modular automated verification — just as it significantly complicates human reasoning about, and understanding of, program behavior. This makes the difficulty in reasoning about large software systems grow at a far faster rate than their size (measured, say, in lines of code). Careful component design that hides pointers from clients can address this problem (Sitaraman and Weide, 1994; Weide, 1996; Hollingsworth et al., 2000).

Another problem is that the tools used by developers for commercial component technologies are still rather primitive. Complications associated with understanding the encodings of design patterns such as abstract factory and observer face the software engineer who uses these tools. Code-generation support helps, but tool-generated code usually is not hidden from the developer. For example, a software engineer developing a Java Bean is not presented with an abstract description of events that hides their details behind a clean interface, but with a concrete view that lays bare all the method calls that are involved in making events seem like a higher-level concept. Applying system thinking within such tools will result in significant improvements in usability of component-based software technologies.

## Bibliography

K. Arnold, et al., *The Jini Specification*. Addison Wesley Longman, Reading, MA, 1999.

C.Y. Baldwin and K.B. Clark, *Design Rules, Volume 1: The Power of Modularity*. MIT Press, Cambridge, MA, 2000.

G. Baumgartner, K. Läufer, and V.F. Russo, *On the Interaction of Object-Oriented Design Patterns and Programming Languages*. Technical Report CSD-TR-96-020, Department of Computer Sciences, Purdue University, West Lafayette, IN, 1996.

J.L. Bentley, Little languages, *Communications of the ACM* 29, 711-721 (Aug. 1986).

G. Booch, *Software Components With Ada*. Benjamin/Cummings, Menlo Park, CA, 1987.

D. Box, *Essential COM*. Addison Wesley Longman, Reading, MA, 1998.

B.J. Cox, *Object Oriented Programming: An Evolutionary Approach*. Addison-Wesley, Reading, MA, 1986.

R.Englander, *Developing Java Beans*. O'Reilly, Sebastopol, CA, 1997.

M. Fowler and K. Scott, *UML Distilled, 2nd Edition*. Addison Wesley Longman, Reading, MA, 1999.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, Reading, MA, 1995.

D.S. Gibson, B.W. Weide, S.M. Pike, and S.H. Edwards, "Toward a Normative Theory for Component-Based System Design and Analysis" in G. Leavens and M. Sitaraman, eds., *Foundations of Component-Based Systems*, Cambridge University Press, Cambridge, UK, 2000, pp. 211-230.

J. Goguen, Parameterized programming, *IEEE Transactions on Software Engineering* SE-10, 528-543 (Sept. 1984).

J.E. Hollingsworth, L. Blankenship, and B.W. Weide, Experience report: Using RESOLVE/C++ for commercial software, *Proceedings ACM SIGSOFT 2000 (Foundations of Software Engineering)*, ACM, New York, 2000, pp. 11-19.

R. Hoque, *CORBA 3*. IDG Books, Foster City, CA, 1998.

B. Joy, G. Steele, J. Gosling, and G. Bracha, *The Java Language Specification, Second Edition*. Addison Wesley Longman, Reading, MA, 2000.

W.R. LaLonde, Designing families of data types using exemplars, *ACM Transactions on Programming Languages and Systems* 11, 212-248 (Apr. 1989).

G.T. Leavens, Modular specification and verification of object-oriented programs, *IEEE Software* 8, 72-80, (July 1991).

B.H. Liskov, R. Atkinson, T. Bloom, E. Moss, J.B. Schaffert, R. Scheifler, and A. Snyder, *CLU Reference Manual*. Springer-Verlag, New York, 1981.

B.H. Liskov and J.M Wing, A behavioral notion of subtyping, *ACM Transactions on Programming Languages and Systems* 16, 1811-1841 (Nov. 1994).

D. Luckham, F.W. von Henke, B. Krieg-Brückner, and O. Owe, *ANNA: A Language for Annotating Ada Programs*. Springer-Verlag, Berlin, 1987.

M.D. McIlroy, “Mass-Produced Software Components” in J.M. Buxton, P. Naur, and B. Randell, eds., *Software Engineering Concepts and Techniques*, Petrocelli/Charter, Brussels, 1976, pp. 88-98.

B. Meyer, *Object-oriented Software Construction, 2$^{nd}$ Edition*. Prentice Hall, Upper Saddle River, NJ, 1997.

R. Monson-Haefel, *Enterprise Java Beans*. O’Reilly, Sebastopol, CA,1999.

M. Shaw, ed., *ALPHARD: Form and Content*. Springer-Verlag, New York, 1981.

M. Sitaraman and B.W. Weide, eds., Component-based software using RESOLVE, *ACM Software Engineering Notes* 19, 21-67 (Oct. 1994).

Y. Smaragdakis and D. Batory, Implementing reusable object-oriented components, *Proceedings Fifth International Conference on Software Reuse*, IEEE, 36-45 (1998).

C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley Longman, 1998.

W. Tracz, Parameterized programming in LILEANNA, *Proceedings 1993 ACM/SIGAPP Symposium on Applied Computing*, ACM, 77-86 (1993).

B.W. Weide, *Software Component Engineering*. McGraw-Hill College Custom Publishing, Columbus, OH, 1996.

T. Williams, Reusable components for evolving systems, *Proceedings Fifth International Conference on Software Reuse*, IEEE, 12-16 (1998).